Kite:

A Leiningen-like Build System for TCL/TK

William H. Duquette

Jet Propulsion Laboratory, California Institute of Technology

William.H.Duquette@jpl.nasa.gov

**Abstract**

Leiningen is a sophisticated build system for the Clojure programming language, used by both novices and skilled developers. Leiningen creates new skeleton project trees, acquires and manages external dependencies (including Clojure itself), provides build and test services, and deploys projects into the larger Clojure ecosystem. This paper examines Leiningen and describes Kite, an attempt to begin to create a similar tool for TCL/TK development.

# Overview

Kite is a new tool for TCL/TK development that is intended to provide one-stop-shopping for most development needs in the typical TCL/TK project: creation and management of project trees, acquisition and management of external dependencies, test and build automation, documentation formatting, and deployment services.

This paper will describe the vision for Kite as a series of requirements; and then for each requirement describe the challenges and obstacles, what can and has been done, what remains to be done; and what will ultimately be left undone.

Kite was inspired by the example of the Leiningen build tool [1] for the Clojure programming language [2].

## Background: Clojure, Maven, and Leiningen

In the summer of 2014 I was experimenting with Clojure, a LISP-variant for the Java Virtual Machine. Clojure integrates tightly and cleanly with its host environment (unlike most LISPs); Clojure packages can call Java packages seamlessly, and vice-versa. Further, being a LISP-variant, Clojure provides the kind of dynamic flexibility and metaprogramming capabilities I am used to finding in TCL/TK. Moreover, it is reasonably efficient with Just-In-Time compilation (as JTcl [3] is not).

I soon found that the essential development tool for Clojure development is a build tool called Leiningen. Leiningen itself was inspired by Maven [4], a Java build tool. Maven's claim to fame is that it will build your Java applications for you: use a standard directory tree, and provide a few details about your project in an XML file, and Maven will take care of the rest including downloading any needed third-party packages from the net. Maven grew along with the notion of a "Maven repository", a server to which packages can be uploaded and from which they can be downloaded given a minimal amount of the information (essentially, the package's name and version number).

Maven is known to be complex to set up, having a dizzying array of options. By contrast, Leiningen was designed from the start for maximum user simplicity.

## Getting Started with Leiningen

The following extended example shows how easy it is to get started programming in Clojure using Leiningen. The following assumes you have a working Java runtime.

First you download Leiningen itself; it is a command-line tool called lein. The first time you run it, it downloads all of its own dependencies and its latest code, and also the latest version of Clojure.

Next, you create your new project's project tree. Go to the parent directory for your project directory, and ask Leiningen to create the project for you:

$ lein new app my-project

Generating a project called my-project based on the 'app' template.

$ cd my-project

$

In this case, I've asked Leiningen to create an appropriate project tree for building a stand-alone application. Leiningen includes a number of other templates, for different kinds of application and for projects that export a library package rather than an application.

The details of my new project are defined in a Clojure file called project.clj. Here I can set the project's name, description, and version; define build products; list external dependencies; and so forth:

$ cd my-project

$ cat project.clj

(defproject my-project "0.1.0-SNAPSHOT"

:description "FIXME: write description"

:url "http://example.com/FIXME"

:license {:name "Eclipse Public License"

:url "http://www.eclipse.org/legal/epl-v10.html"}

:dependencies [[org.clojure/clojure "1.6.0"]]

:main ^:skip-aot my-project.core

:target-path "target/%s"

:profiles {:uberjar {:aot :all}})

$

As part of the project tree, Leiningen defines a Clojure package called my-project.core; this package will contain the project's code. As a start, it contains the application's main routine.

We can run the application immediately. Normally with Java applications this is a messy task involving class paths and such like; here, we just ask Leiningen:

$ lein run

(Retrieving org/clojure/clojure/1.6.0/clojure-1.6.0.pom from central)

(Retrieving org/clojure/clojure/1.6.0/clojure-1.6.0.jar from central)

Hello, World!

$

Note that as part of running the application, Leiningen retrieved the version of Clojure listed as a dependency in the project.clj file. Leiningen maintains its own local Maven repository, and can pull in JVM packages from a variety of Maven repositories on the net.

Leiningen creates the beginnings of a test suite, and will run it when asked. The initial test naturally fails.

$ lein test

lein test my-project.core-test

lein test :only my-project.core-test/a-test

FAIL in (a-test) (core\_test.clj:7)

FIXME, I fail.

expected: (= 0 1)

actual: (not (= 0 1))

Ran 1 tests containing 1 assertions.

1 failures, 0 errors.

Tests failed.

$

Like TCL, Clojure is a dynamic language; and like TCL programmers, Clojure programmers spend a lot of time interacting with their code through a shell command-line. Being LISP programmers, they refer to the shell as a "REPL" (Read/Eval/Print Loop). The plain Clojure ".jar" file provides an exceedingly primitive REPL; Leiningen provides a pleasant one, and sets it up so that your code is available:

$ lein repl

nREPL server started on port 63823 on host 127.0.0.1 - nrepl://127.0.0.1:63823

REPL-y 0.3.2, nREPL 0.2.3

Clojure 1.6.0

Java HotSpot(TM) 64-Bit Server VM 1.6.0\_65-b14-462-11M4609

Docs: (doc function-name-here)

(find-doc "part-of-name-here")

Source: (source function-name-here)

Javadoc: (javadoc java-object-or-class-here)

Exit: Control+D or (exit) or (quit)

Results: Stored in vars \*1, \*2, \*3, an exception in \*e

my-project.core=> (+ 1 1)

2

my-project.core=> (exit)

Bye for now!

$

Finally, you've written your application, or your library package, and you want to deploy it. If it's an application Leiningen will build it as a .jar file; or, if you like, as an "uber-jar" that includes all dependencies and can run standalone (given the Java Run-time):

$ lein uberjar

Compiling my-project.core

Created /Users/will/work/temp/my-project/target/uberjar/my-project-0.1.0-SNAPSHOT.jar

Created /Users/will/work/temp/my-project/target/uberjar/my-project-0.1.0-SNAPSHOT-standalone.jar

$

If your project is a library package, rather than an application, you can build the .jar file and use "lein deploy" to deploy it to a Maven repository where anyone in the world can see it, download it, and use it.

## Leiningen: The Bottom Line

In less than ten minutes, from a standing start, you can download Leiningen, create a project, add some code, add some external dependencies, run it, test it, build it for deployment, and get it out where people can see it. We have pieces of all of this in the TCL world; but not so easily, and not all in one place.

As a TCL evangelist, I want to be able to point new users at a tool that will enable them to get started and be productive from the beginning. As a TCL user, I want to be able to create new projects on a whim and make them available to others with a minimum of fuss.

In short, I want a Leiningen-like tool for TCL/TK. It should get the job done, and it should be radically simple to use.

## The Athena Build System

Athena, the project I've been working on since late 2008, has a build system constructed out of Makefiles and *ad hoc* scripts, with an insidious dependence on Subversion's svn:external capability. The current codebase has 41 individual Makefiles, some containing build targets, some containing rules or symbol definitions; most of them are boilerplate, allowing the whole project to be built by saying "make all" in the project root directory. There are perhaps four directories in which defining a Makefile makes sense; these are all C libraries. The scripts format documentation, build the on-line help database, build distribution tarballs, and manage svn:external links. It's a complex system, and one that's not easily re-used.

It all works, but I want a system that "just works" and that can be applied to multiple projects with minimal fuss, and one that is pleasant to use. A Leiningen-like tool fits the bill; but it has to be a tool that works the way we do and meets our immediate needs.

# The Kite Vision: Desired Features

These two desires—a better build system for the Athena project and a TCL build tool with the simplicity of Leiningen—arrived at the same time, and the collision resulted in the four deadly words: I Can Code That.

I chose to call this tool Kite, as a reference to starkits and zipkits, the TCL equivalent of .jar files.

Based on Leiningen's feature set and our own needs there's a particular set of features I would like Kite to possess. Some do not currently exist, but are easy to implement; others exist in a wide variety of tools that are already available and just need to be glued together; some do not exist, and will need the participation of other developers and organizations. This section lists each "requirement"; subsequent sections will go into detail about each.

**Single Download:** It should be possible to download a single distribution and get everything you need to create, test, build, document, and deploy a typical TCL/TK project, all in one place.

**Creation of Project Trees:** A well-designed project tree is the fruit of many small decisions. Kite should make creation of well-designed project trees trivially easy.

**New Project Elements:** Adding elements (i.e., new library packages or applications) to an existing project should be as easy as creating a whole new project, and shouldn't have to involve copying and modifying existing files in a text editor.

**A Project File:** Each project should include a *project file* that declaratively defines the project, its build targets, and its dependencies in a simple, concise format. Kite will use this information to perform the project's tasks.

**Provided Applications and Library Packages:** A project should be able to provide applications and library packages to external users. Kite must support building applications and packages for such external use.

**Project Version Management:** The project as a whole should have a version number, which should be available to project applications and which should be the version number of all project library packages.

**External Dependencies:** The user should be able to list required external packages in the project file; Kite should be able to pull them down from an external deployment repository (i.e., from a teapot).

**Running Project Applications:** It should be trivially easy to run the project's applications in development as TCL scripts, i.e., without a build step.

**Running Arbitrary Scripts:** It should be trivially easy to run arbitrary scripts in the context of the project's code base, without editing environment variables or manually setting the auto\_path.

**Interactive Shell:** It should be trivially easy to pop up an interactive shell with access to the project code base, and especially in the context of particular project applications.

**Test Management:** Part of creating a library package should be creating a stub test suite; and running project tests should be trivially easy and convenient from anywhere in the project tree.

**Binary Extensions:** Kite should support library packages and applications that include binary extensions.

**Documentation Formatting:** I like well-documented projects, with man pages for the public APIs and detailed user's guides. Kite should support this kind of documentation.

**Build Project Distribution Files:** Kite should be able to build distribution tarballs or ".zip" files containing the deployment products (applications, libraries, documentation).

**Automated Project Builds:** Kite should be able to do all build steps with one command (build binary extensions, run tests, format documentation, wrap applications and libraries, and build distribution files).

**Local Installation:** It should be easy to install project applications and library packages onto the local machine for general use.

**Automatic Deployment:** It should be possible to push a project release (or at least the project's provided libraries) to a deployment server (i.e., a teapot) for use by other users across the internet.

**Plugin Architecture:** It should be possible for the user to add plugins of several kinds: definition of new project tree templates; definition of new project element templates; and new tools that operate on the project tree.

# Obstacles and Solutions

This section goes through each requirement, discussions the obstacles and technical challenges, and how Kite currently implements the requirement (if it does).

## Single Download

This is not currently possible, in part because many of the tools on which Kite depends are ActiveState-proprietary (tclapp, teacup, teapot.activestate.com), but also because there's no easy way to pull down a single file TCL interpreter with all included packages for the TCL version of your choice (though TIP #430 [5] is a step in that direction). The closest we come are ActiveState's basekits; but they do not include all of the packages that come with the standard TCL core.

And then, ActiveTcl's license quite reasonably forbids using ActiveTcl to create a TCL development system that competes with ActiveTcl. Kite could conceivably download the basekit, and use it (with some glue code) to run apps in development, but that would seem to cross the line.

## Creation of Project Trees and Addition of New Elements

Kite must be able to create new project trees, and add element skeletons (e.g., for new library packages or applications) to existing project trees. This is not difficult; code generation is practically TCL's middle name. The only controversial part is what the project tree should look like.

The essence of a tool like Leiningen or Kite is picking one adequate approach out of all of the myriad possible solutions, and living with it. I chose a tree structure that matches the way we've been building our applications for some years, with some modifications for improved automation. It is described in detail in Section 5.

A project exists to provide functionality to external users in the form of applications or code libraries. Consequently, a Kite project should be able to contain at least two kinds of element:

* TCL applications (including implementation packages)
* TCL library packages (including test suites)

And Kite should be able to build or package these for external consumption.

An obvious choice for libraries is to build them as teapot packages, for installation in the user's local teapot. Teapot packages can be built using the teapot-pkg tool, provided with TclDevKit, or more simply by adding the appropriate teapot.txt manifest file to the package directory and zipping it up using the zipfile::encode package [6].

The obvious solution for delivering standalone TCL applications is as starkits or starpacks. There are a number of tools for doing that; Athena has been using TclDevKit for many years, and so Kite has standardized on using TDK's tclapp tool for building executables. Should Kite be open-sourced I'd like to look at providing other mechanisms as well; but in all honesty tclapp does many things well that I'd hate to have to implement on my own, especially its ability to pull packages directly from a teapot repository.

In addition, a project should be able to contain applications and libraries for internal use only.

Kite supports this requirement through its kite new, kite add, and kite build tools.

## A Project File

Leiningen's big advantage is that all of the information the tool needs to automate the project's development tasks is laid out declaratively in the project.clj file. In Athena, historically, this kind of metadata is scattered across the bevy of Makefiles in the project tree. In Kite projects, the required metadata—what is to be done—is abstracted out, and put in a file called project.kite; and the actual operations—how it is done—are embodied in Kite's own code.

In particular, the project file must:

* Identify the project and its version
* Identify the applications intended for external use, and how they are to be built.
* Identify the library packages intended for external use.

All other project file content is subordinate to these purposes.

The content of the Kite project file is described in Section 4.

## Project Version Management

Leiningen projects are stamped with their version number in the name of the distribution .jar file; the role of TCL's "package management" system is played by Leiningen or Maven, not by anything in the language itself.

In TCL, by contrast, every package knows precisely what its version number is, and which versions of packages that it requires. Kite must take the version number from the project file and apply it to the package ifneeded and package provide statements in each of the project's libraries, and also make it available to any project applications.

Kite does so by means of special markings in each package's pkgIndex.tcl and pkgModules.tcl files, which are updated automatically by Kite as needed.

## External Dependencies

With Leiningen, you list your project's external dependencies in your project.clj file, and Leiningen retrieves them from any available Maven repository. In the TCL world we have teapot repositories, the best known being ActiveState's teapot at teapot.activestate.com. If there's a well-supported alternative to the teapot repository for TCL code, I'm unaware of it.

Consequently, the obvious thing is to support pulling down external dependencies from a teapot repository. Athena has been developed using ActiveTcl for many years, so Kite targets ActiveState's repository.

Naturally, Kite must use the teacup executable, provided with ActiveTcl, to manage this.

Ironically, working with a local teapot has been one of Kite's biggest challenges. ActiveTcl is typically installed for all users on a machine. On Windows, this requires "admin" privileges; on Linux and OSX, it must be installed as "root". Consequently, the transparent teapot repository installed with ActiveTcl is also installed as "root"; and to install packages into it you need to have admin privileges or be "root".

The solution is to create a secondary teapot in the user's home directory. ActiveTcl supports this, but setting it up requires performing certain steps as "root". It would be a real help if ActiveTcl supported a secondary teapot as a matter of course and could set it up automatically and transparently to the user. The teacup tool could then install packages into the user's own teapot by default, and touch the installation teapot only when explicitly asked.

The kite teapot tool sets up such a local teapot; and the kite deps tool monitors and pulls down external dependencies.

## Running the Project's Applications and Arbitrary Scripts

Like Leiningen, Kite should make it easy to run the project's applications in development, and to run *ad hoc* TCL scripts in the context of the project and its packages, without requiring the developer to worry about setting environment variables or having to fiddle with the auto\_path.

This is generally straightforward. Kite installs external dependencies into a local teapot using teacup, and executes scripts by setting TCL\_LIB\_PATH to the project's lib/ directory and then passing them to the development tclsh executable. Thus, the executed script sees the external dependencies and its own project libraries.

The kite run tool executes the project's primary application, passing it any command line arguments; and if Kite's first argument is a TCL script it is executed in the context of the project's code base.

## Providing an Interactive Shell

Similarly, Kite should be able to provide an interactive TCL shell with immediate access to the project code, again without requiring the developer to fiddle with the auto\_path. In this case, tkcon [7] makes a fine helper tool; Kite uses the development tclsh to execute tkcon just as it would any project script.

The kite shell tool invokes tkcon in this way.

## Test Management

Kite should be able to run all or part of the project's test suites, which will naturally use tcltest(n) [8]. This is well understood technology; Kite's role is to help set up the relevant directories and boilerplate, and to make it easy to execute the tests from anywhere in the project tree.

The kite test tool runs all project tests; arguments can limit it to a particular library package or module within that package, and any options are passed along to tcltest. Thus, the following command entered anywhere within the project's directory tree will execute the mycode.tcl module's myproc-1.\* tests:

$ kite test mylib mycode –match "myproc-1.\*"

## Binary Extensions

Pure TCL packages are easy to "build" and deploy: just write the code, test it, and wrap it up. Binary TCL packages require much more work. Kite must make it possible to include the source for binary extensions in the project tree, and automate the process of building it at least at a high-level.

In the future, of course, it would be good if simple binary extensions could be built more easily, e.g., by incorporating Critcl [9] or something like it into Kite. But even with that, there will always be a need for full-up TEA extensions.

Consequently, Kite supports "src" directories. These are defined in the project file, e.g.,

src mybinlib

By default, Kite assumes that "src/mybinlib" contains a Makefile with "clean" and "all" targets. Alternatively, the user can specify alternative shell commands to replace "make clean" and "make clean all".

src mybinlib –cleancmd myclean –buildcmd mybuild

A "src" directory may contain anything at all. If it contains a binary extension, then "make all" should install the .dll or .so into a lib/ subdirectory, which is then provided normally.

## Documentation Formatting

Leiningen does not, by itself, format project documentation, though there are plugins that aim at that. For my part, I write and use man pages for a project's public APIs; I write user's guides; and I like them to be easy to read and easy to browse, with lots of cross-reference links. In short, I want well-done HTML documentation; and every project I've written for many years has included some mechanism to format HTML documentation from some simpler source format. Generally speaking, two kinds of documents are needed: manual pages and traditional documents with section numbers and tables of contents.

Kite includes tools for both documentation formats, using an HTML-like macro syntax based on the textutil::expander [10] package.

Documents to be formatted are placed in the docs/ tree; the kite docs tool handles the rest.

## Build Project Distribution Files

Having built all documentation, libraries, and applications, one wants to package up the distribution to give to others, usually as a .zip file or a tarball. Tcllib's zipfile::encode package makes it easy to build .zip files, and so that's what the kite dist tool does.

## Automating Project Builds

Kite should able to do the following, with a minimum of input from the user, based on the contents of the project file:

* Update required external dependencies
* Compile the project's binary extensions, if any
* Run all tests, halting on failure
* Format all documentation
* Build provided libraries for external deployment
* Build executables as starkits or starpacks.
* Build distribution .zip files or tarballs.

This is simply a matter of combining lower level tools into a single command: kite build all.

## Deployment of Packages

One of Leiningen's best features is the ability to build a library package and deploy it directly to a Maven repository on the 'net for others to see and use. So far as I'm aware, the teacup/teapot tools allow for no such easy deployment of library packages; and then, teapot.activestate.com is intended to be a repository of stable, reliable software.

All that said, it would be lovely if there were a Teapot repository on the net that allowed developers to freely upload their packages as Maven repositories do. We'd also have to handle documentation better, or at least links back to the package's home page.

In any event, this is a bridge too far at the moment; and it's academic unless Kite is open-sourced.

## Plugin Architecture

This has been a desire since day one, but has not yet been a priority. If Kite is open-sourced, it will swiftly become one.

# The Project File

The project file, or project.kite file, contains a complete description of the project from the point of view of Kite and is the basis for virtually all of Kite's operations. It is, naturally, a TCL file parsed by a safe interpreter.

## The Project Root Directory

At start-up, Kite looks in the current directory for the project.kite file, and then in the parent directory, working its way up in the directory hierarchy until it finds it or reaches "/". The directory in which project.kite is found is presumed to be the root of the project tree; most Kite tools operate relative to this directory and can only be used if Kite is "in the tree". (A few are more general, and one, kite new, can only be used outside any project tree.)

## Sample Project File

Here is an example taken from the Athena code base:

# project.kite

project athena-mars 3.0.1a0 {Mars Simulation Support Library}

poc William.H.Duquette@jpl.nasa.gov

# Applications

app mars -apptype exe

# Provided Libraries

provide marsutil

provide marsgui

provide simlib

provide Marsbin -binary

# Compiled Directories

src libGeotrans

...

src Marsbin \

-build {make -f MakeTEA clean all} \

-clean {make -f MakeTEA clean}

# External Dependencies

require snit 2.3

...

require kiteutils 0.1.2 -local

# Distribution Targets

dist install-%platform {

README.md

%apps

%libs

docs/\*.html

docs/man\*/\*.html

%get {docs/mag.docx http://...}

}

The statements in the project file are straightforward, and naturally have TCL syntax.

* project defines the project's name, version, and gives a brief description.
* poc gives a point-of-contact, for inclusion in the documentation.
* app defines an application for external use, and specifies whether it requires TK and whether it is built as a starkit or a starpack.
* provide defines a library that is exported for local use. Binary packages are so indicated.
* src defines a src/ directory, and optionally the clean and build commands for use with it.
* require identifies an external dependency. The -local flag indicates that it's a locally-built package and cannot be retrieved from a remote teapot.
* dist defines a distribution set: the files that will go into a distribution .zip file. A project can define any number of distribution sets.

The project file is usually edited by hand, but there are occasions when Kite will save a new copy with added data.

## Applying Project Metadata

Whenever Kite succeeds in loading the project file at start-up, it goes on to apply the project metadata to the project code. In particular,

* The project version number is used as the version number for all library packages in the project; and so when it changes all relevant package ifneeded, package provide, and package require commands must also be updated.
* The project file lists all of the required external packages, with the required version numbers; but this information (and especially the version numbers) must be taken into account within the project's library packages. Different library packages can package require different dependencies, but the required version numbers must be updated to match the project file.

These updates are made to the relevant project files whenever they would result in new content.

# The Project Tree

Here is an example of a project directory tree for a project with both an exported application and a provided library.

root/ Project Root Directory

project.kite Project File

README Project README

LICENSE Project LICENSE

bin/ Executable files

*appname*.tcl Loader script for application *appname*; see below.

docs/ Documentation tree

man1/... Application manual pages

man5/... File format manual pages

mann/... Package API manual pages

mani/... TCL Interface manual pages

index.ehtml Documentation root page

ug.ehtml User's guide

lib/ Library Directory

*libname*/ Provided library *libname*

pkgIndex.tcl Package index; see below

pkgModules.tcl Package loader; see below

*module*.tcl A package *module*.

...

*appname*app/ Implementation package for application *appname*.

pkgIndex.tcl Package index; see below

pkgModules.tcl Package loader; see below

main.tcl Defines command "main"; see below.

...

test/ Test directory

*libname*/ Tests for library *libname*.

all\_tests.test Test loader; see below.

*module*.test Tests for the named *module*.

*appname*app/... Tests for application *appname*.

src/ Parent of "src" directories

*name*/... "src" directory *name*.

## Application Structure

An application *appname* in a kite project consists of:

* An application loader script, bin/*appname*.tcl.
* An implementation package, lib/*appname*app.tcl.

The implementation package is a standard library package in structure (see Section 5.2); it must include a module called main.tcl that defines a command called main:

proc main {argv} {

...

}

The loader script is boilerplate. It sets up the auto\_path, requires TCL (and TK if needed be) and the application's implementation package, and calls the application's main command with any command line arguments. The final part of the script looks like this:

try {

# Allow for interactive testing

if {!$tcl\_interactive} {

main $argv

}

} trap FATAL {result} {

# A fatal application error; result is a message intended

# for the user.

puts $result

puts ""

} on error {result eopts} {

# A genuine error; report it in detail.

puts "Unexpected Error: $result"

puts "\nStack Trace:\n[dict get $eopts -errorinfo]"

}

There are several things to note. First, main is not called if the script is loaded into an interactive shell. This makes interactive testing easier. Second, if the application terminates by throwing a **FATAL** error the error message is printed to the console and no stack trace is generated; the message is presumed to be aimed at the end user. Other errors result in detailed stack trace.

This has proven to be a useful convention.

## Package Structure

A library package called *libname* resides in a subdirectory of lib/ called lib/*libname*. Every package, whether it is a provided library package or an application implementation package, will include at least three files:

* The usual pkgIndex.tcl file.
* A pkgModules.tcl file, which is loaded by pkgIndex.tcl. It:
  + Calls package provide for the package.
  + Calls package require for all required packages.
  + Sources all package modules
* At least one code module.

The body of the pkgIndex.tcl file looks like this:

# -kite-ifneeded-start DO NOT EDIT THIS BLOCK BY HAND

package ifneeded *libname* 1.2.3 [list source [file join $dir pkgModules.tcl]]

# -kite-ifneeded-end

The "-kite-ifneeded" marks allow Kite to update the package version number when the project version number changes.

The body of the pkgModules.tcl file looks like this:

# -kite-provide-start DO NOT EDIT THIS BLOCK BY HAND

package provide *libname* 1.2.3

# -kite-provide-end

# -kite-require-start REQUIRE EXTERNAL PACKAGES HERE

package require snit 2.3

package require –exact myotherlib 1.2.3

# -kite-require-end

namespace eval ::*libname*:: {

variable library [file dirname [info script]]

}

source [file join $::*libname*::library *module*.tcl]

The "-kite-provide" marks allow Kite to update the package version number when the project version number changes.

The "-kite-require" marks allow Kite to update the package require statements to match the requirements in the project file. This block should include all of the package's requirements. If a required package is one provided by the project itself, then it is required with the -exact flag; this ensures that package *libname* will pick up myotherlib from the same project tree.

## Test Structure

Tests use the tcltest(n) framework. Test targets are defined as subdirectories of the project test/ directory; they usually go hand-in-hand with lib/ subdirectories, but this is not required. Each test target directory will contain at least two files:

* all\_tests.test, which runs the other tests in the directory.
* One or more specific test modules, e.g., *module*.test.

The body of the all\_tests.test file looks like this:

if {[lsearch [namespace children] ::tcltest] == -1} {

package require tcltest 2.2

eval ::tcltest::configure $argv

}

::tcltest::configure \

-testdir [file dirname [file normalize [info script]]] \

-notfile all\_tests.test

::tcltest::runAllTests

This is a fairly standard use of tcltest(n). Each of the .test files in the directory will be executed in turn, in independent TCL interpreters, and the results will be compiled.

The individual test modules look like this:

if {[lsearch [namespace children] ::tcltest] == -1} {

package require tcltest 2.2

eval ::tcltest::configure $argv

}

namespace import ::tcltest::test

source ../../lib/*libname*/pkgModules.tcl

namespace import ::*libname*::\*

test mytest-1.1 {a description} -body {

...

} -result {...}

tcltest::cleanupTests

This is a fairly normal tcltest(n) test script. Two things are of note. First, we import only the "test" command from ::tcltest, just as a matter of course. Second, the script explicitly sources the library package to be tested, rather than doing a package require. This ensures we get the code actually in the project tree. If we used package require, the .test file would need to use the -exact flag with some mechanism for inserting the project version number.

# Kite Tools

The Kite application is a console mode application called "kite"; it provides the following subcommands:

kite add Adds a new project element (e.g, an application or library package) to an existing project tree.

kite build Builds applications as starkits or starpacks, and provided libraries as teapot .zip files. With "all", it builds the entire project from start to finish.

kite clean Cleans up build artifacts.

kite compile Compiles any or all "src/" directories, as listed in the project file.

kite deps Verifies that all external dependencies listed in the project file are available. With "update", retrieves needed dependencies from teapot.activestate.com.

kite dist Builds .zip files for any or all of the distribution sets defined in the project file.

kite docs Formats some or all of the project documentation.

kite env Kite depends on a number of external tools: tclsh, tkcon, teacup, tclapp. This tool outputs Kite's view of its environment, including the specific tools it will use.

kite help Displays a list of the subcommands, or help for a particular subcommand.

kite info Displays information about the project, largely from the project file, in human-readable form.

kite install Installs any or all of the project's applications and provided libraries for general use by the user on the user's machine.

kite new Creates a new project tree on the disk. The user may specify the template.

kite replace A simple global text search and replace tool.

kite run Runs the project's primary application, passing it any command line arguments.

kite shell Invokes tkcon, loading the project's primary application's implementation package (if any).

kite teapot Tools for verifying that the current teapot is writable by the user, and for creating and linking a new one if necessary.

kite test Invokes any or all of the project's test suites, reporting the results.

kite version Displays Kite's own version number.

In addition, if the first argument to kite is a .tcl file then Kite will execute it using the default tclsh in the context of the project's code base.

# Kite Libraries

Kite provides two library packages for use by other projects:

* kiteutils(n): a collection of utility modules, including a module for interacting with the project file. Thus, new tools using kiteutils(n) can work within a Kite project tree.
* kitedocs(n): the specific packages that support the Kite documentation formats, as used by the kite docs tool.

# Future Work

Of the requirements listed in Section 2, Kite provides all of them, at least in part, except for

* Single Download
* Automatic Deployment
* Plugin Architecture

That said, it is still a young application; we are only just starting to use it to manage Athena development. There is no doubt that as we use it more fully we will discover more "requirements" and will need to tweak what we already have. It is, however, self-hosting (and has been since day one), and has been used for two other ancillary projects.

It may be possible to open source Kite in the future; in that case, many other possibilities open up.

# References

[1] Leiningen build tool, http://leiningen.org/.

[2] Clojure programming language, <http://clojure.org/>.

[3] JTcl programming language, https://jtcl-project.github.io/jtcl/.

[4] Maven build tool, http://maven.apache.org/.

[5] Woods, Sean and Fellows, Donal, TCL Improvement Proposal (TIP) #430: Add basic ZIP archive support to TCL, http://www.tcl.tk/cgi-bin/tct/tip/430.html.

[6] Kupries, Andreas, zipfile::encode Package, found in Tcllib, http://core.tcl.tk/tcllib/doc/trunk/embedded/www/tcllib/files/modules/zip/encode.html.

[7] Hobbs, Jeff, Tkcon: Interactive TCL Console, http://tkcon.sourceforge.net/docs/index.html.

[8] tcltest(n): TCL Unit Test Framework, http://www.tcl.tk/man/tcl8.6/TclCmd/tcltest.htm.

[9] Landers, Steve, Critcl: tool to embed C code in TCL scripts, http://equi4.com/starkit/critcl.html.

[10] Duquette, William, textutil::expander, found in Tcllib, http://docs.activestate.com/activetcl/8.5/tcllib/textutil/expander.html.

# Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.

Copyright 2014 California Institute of Technology. Government sponsorship acknowledged.