ENOVIA DesignSync

Stcl Programmer's Guide

3DEXPERIENCE 2022

Table of Contents

Introduction

The **ENOVIA Synchronicity stcl Programmer's Guide** provides information about customizing or extending the built-in capabilities of the DesignSync family of products (DesignSync, ProjectSync SyncAdmin, etc.) using stcl scripts. **stcl** (Synchronicity Tcl) is the combination of the Tcl scripting language and the DesignSync command set. Use this document with the ENOVIA Synchronicity Command Reference to develop stcl scripts.

Note on using this guide: References from the *ENOVIA Synchronicity stcl Programmer's Guide* to the *ENOVIA Synchronicity Command Reference* guide always link to the ALL version of the guide, which contain information about all working methodologies for DesignSync. For more information about the available working methodologies, see ENOVIA Synchronicity Command Reference.

Why Tcl?

Tcl (tool command language) is a scripting language that provides generic programming constructs such as variables, loops, conditionals, and procedures. The DesignSync products incorporate the Tcl interpreter, giving you a powerful environment in which to create and run scripts. Because Tcl is an interpreted language, you do not have to wait for your code to compile; your results are immediate. Also, because the DesignSync API is built on top of a standard language like Tcl, you do not have to learn a proprietary language specific only to DesignSync.

What's stcl?

The DesignSync commands along with the Tcl scripting language are referred to as Synchronicity tcl (stcl). You can use the stcl commands in one of the DesignSync command shells or in scripts run on DesignSync clients or servers. For DesignSync, you can create scripts for clients and servers (SyncServers). For ProjectSync, you create server scripts.

Note that 'stcl' is also the name of a DesignSync command shell. The stcl shell, like the dss shell, supports the DesignSync text commands, sometimes called dss commands. Unlike the dss shell, the stcl shell also runs the Tcl interpreter. You can determine the version of Tcl included in your installation's stcl interpreter by using the Tcl `info tclversion` and `info patchlevel` commands within an stcl/stclc client shell. You can run client-side stcl scripts from any of the DesignSync clients (see How to Run stcl Scripts from Clients). **Note**: DesignSync software supports Tcl but not Tk.

Getting Started with stcl

Although the **ENOVIA Synchronicity stcl Programmer's Guide** provides many examples of Tcl scripting, it is not designed to be an introductory guide to the Tcl

scripting language. The following references might be of use to new Tcl scripters. Also, see stcl Scripting Tips for some Tcl guidelines that apply specifically to stcl scripting.

Tcl Scripting References

`http://www.tcl.tk`	Tcl Developer Xchange web site. The Tcl/Tk Manual Pages web page (`http://tcl.activestate.com/man`) provides Tcl command documentation. Use the `info tclversion` and `info patchlevel` commands within an stcl/stclc client shell to determine the Tcl version supported by stcl, then select the documentation corresponding to that version from the `http://tcl.activestate.com/man` web page.
`http://tcl.sourceforge.net`	Development home for Tcl.
Effective Tcl/Tk Programming by Mark Harrison and Michael Mclennan, Addison-Wesley, 1997. ISBN: 0-201-63474-0	Shows how to build effective and efficient Tcl/Tk applications. It clarifies some of the more powerful aspects of Tcl/Tk, such as the packer, the canvas widget, binding tags, and sockets. Throughout the book, the authors develop numerous applications and a library of reusable components.
Practical Programming in Tcl and Tk by Brent Welch. Prentice Hall, 1999. 3rd Ed ISBN: 0-13-022028-0.	Covers Tcl/Tk 8.2 in detail, and includes chapters on C programming for Tcl, the Tcl Web Server, and the Tcl Web Browser plugin. Included is a CD-ROM with all the sources for the examples.
Tcl and the Tk Toolkit by John K. Ousterhout. Addison-Wesley, 1994. ISBN: 0-201-63337-X.	Tcl/Tk command reference.

Tcl/Tk in a Nutshell by Paul Raines and Jeff Tranter. O'Reilly & Associates, 1999. ISBN 1-56592-433-9 — Tcl/Tk command reference.

Related Topics

Client-Side Versus Server-Side stcl

Developing Server-Side stcl Scripts

How to Run stcl Scripts from Clients

How to Run stcl Server Scripts

Introduction to the DesignSync Object Model

Working with Client stcl Scripts

Working with Server stcl Scripts

Synchronicity's stcl Environment

The stcl Environment

You can run stcl scripts on DesignSync clients and servers. The particular application you are developing determines whether it's best to create a DesignSync client script or a SyncServer script. An understanding of the client/server architecture of the ENOVIA Synchronicity tools will help you make this determination.

What Are SyncServers?

A **SyncServer**, is an HTTP server process configured to manage data for the ENOVIA Synchronicity tools, including the following:

- DesignSync vault data, which consists of the hierarchy of design objects that are under revision control. The SyncServer also contains a tags database that manages the tag names for objects under revision control.
- ProjectSync relational database, which contains the note type data, a note link table, and the predefined tables from ProjectSync, such as the user database. Each note type defined on a server is stored in a database table. Each note created is an instance of one of these tables with a set of values. Each field in the note type is likewise a field in the table. The note type table defines attachments of notes onto web objects. For example, revision control notes are notes with attachments to objects under DesignSync revision control.
- IP Gear relational database. **Note**: The scope of the **stcl Programmer's Guide** does not cover stcl scripting for IP Gear applications. See the IP Gear documentation for IP Gear administration and customization information.

SyncServers are implemented as Apache servers on UNIX and IIS servers on Windows. SyncServers automatically load DesignSync runtime libraries when needed.

What Are the DesignSync Clients?

As clients, DesignSync provides a graphical interface, DesSync, and a number of shells (dssc, dss, stclc, and stcl). These DesignSync clients are the interfaces for the revision control and configuration management commands DesignSync supports. In carrying out the commands you specify, DesignSync communicates with the SyncServer that manages the data under revision control. The local workspace corresponding to the SyncServer vault folder is located on the client, as well as local metadata used by DesignSync to manage the data.

The client for the ProjectSync tool is your HTML browser. You view the ProjectSync interface as an HTML page. As with the DesignSync client, ProjectSync carries out your commands by communicating with the SyncServer that manages its relational database.

Because the ProjectSync client is an HTML browser, you can only develop server-side stcl scripts for ProjectSync, not client stcl scripts. You can develop client stcl scripts, as well as server stcl scripts, for DesignSync applications.

How DesignSync Clients and Servers Communicate

The ENOVIA Synchronicity tools use the **http** protocol, as well as a special sync protocol, for communication between their clients, such as DesSync and stcl, and their SyncServers. The http protocol, which is a layer on top of TCP, is the standard way that web browsers (clients) communicate with web servers. A standard dialog between the DesignSync clients and servers is as follows:

1. The client sends a URL to the server.
2. The SyncServer interprets the URL and performs some action.
3. The SyncServer sends back information to the client.

The type of information passed between the clients and SyncServers depends on whether the client is DesignSync or ProjectSync, as well as the nature of the command being executed. For example, because the ProjectSync client is an HTML browser, the data sent from the SyncServer to ProjectSync must be in HTML format. Thus, the architecture used has implications for stcl scripting and the formats of the input and output of the scripts you develop.

DesignSync clients generally use the sync protocol in their URLs to communicate with the SyncServer, for example, `sync://machiavelli:2647/Projects/stclguide`. ProjectSync clients generally use the http protocol in their URLs to communicate with the SyncServer, for example, `http://bugserver:2647/Projects/rel4.4`. Synchronicity client/server communications also support secure protocols, **https** and **syncs**. See *DesignSync Data Manager Administrator's Guide*:Overview of Secure Communications for more information.

You can write stcl scripts for execution by a DesignSync client (DesSync, dssc, dss, stclc, or stcl) or a SyncServe). DesignSync clients and SyncServers run an stcl interpreter, a Tcl interpreter that also supports the DesignSync revision control commands. See Client-Side Versus Server-Side stcl for a comparison of client versus SyncServer scripts.

For further information about the DesignSync architecture, see Introduction to the DesignSync Object Model.

Related Topics

Accessing Cadence Web Objects

Client-Side Versus Server-Side stcl

Client-Side Versus Server-Side stcl

You can write stcl scripts for execution by a DesignSync client (DesSync, dssc, dss, stclc, or stcl) or by a SyncServer. However, the commands supported by the stcl interpreter differ slightly depending on whether your stcl script is running on a client or a SyncServer. For example, a client script has access to your workspace, but a server script has no knowledge about the objects in your workspace. A client script can query a SyncServer for information about vaults, but a server script cannot query clients. A client script cannot call `note` commands -- the client has no knowledge of the SyncServer notes database.

Client-Side Scripts

Client-side scripts are necessarily DesignSync client scripts. The ProjectSync client is an HTML browser with no access to an stcl interpreter; therefore, stcl is only supported for ProjectSync server scripts.

Client-side scripts have visibility into the user's workspace and environment as well as a limited visibility into the SyncServer and vaults on the server. Thus, you can create DesignSync client-side scripts to automate user tasks or implement enhancements to the built-in DesignSync command set. You can also run server-side scripts remotely from a client using the DesignSync `rstcl` command.

Server-Side Scripts

Server-side scripts have visibility into events and data within the SyncServer, but no visibility into the user's workspace and environment.

You create server-side scripts for any of the following reasons:

- To set server-wide policies (such as triggers or access controls)
- To create server customizations (such as customized ProjectSync panels or data sheets)
- To reduce the amount of client/server traffic that a client-side script accessing vault data requires
- To execute commands that are only available as server-side commands (such as `access reset` and most ProjectSync commands)

When to Use a Client-Side Script

The choice to implement a script as server-side or client-side script is sometimes not obvious. All of Tcl and most of the DesignSync command set are available for use in both server-side and client-side scripts, although some commands are server-side only.

The following are applications where client-side scripts are applicable:

- If your site or project team needs a customized version of a DesignSync command, you can create a wrapper script around the command. You include your own customizations to the command in the wrapper script.
- Create stcl utilities to report the status of design objects in your workspace, such as an stcl script to list out the design objects that have been modified.

When to Use a Server-Side Script

The following are applications where server-side scripts are applicable:

- If a client-side script is going to require many server hops, you might be better off with a server-side script.
- If the purpose of the script is to display HTML information like a data sheet, use a server-side script; the server returns script results as HTML.
- Most scripts that control or customize ProjectSync cannot be run from the client, so a server-side script is necessary.

Methods of Running Client-Side Scripts

You can run client-side scripts by:

- Using the DesignSync `run` command (to run dssc/dss or stclc/stcl scripts)
- Using the Tcl `source` command (to make dssc/dss or stclc/stcl scripts available in an stcl shell)
- Specifying a start-up script when you invoke the client
- Storing the stcl script in a directory from which it is automatically loaded (autoloaded) when a user invokes the script
- Setting up triggers to automatically run scripts based on some event

See Working with Client stcl Scripts for details.

Methods of Running Server-Side Scripts

You can run server-side scripts by:

- Passing a URL script request from your web browser to the SyncServer
- Running a remote server-side script from a client by using the rstcl command
- Setting up triggers to automatically run scripts based on some event
- Defining custom ProjectSync panels

See Working with Server stcl Scripts for details.

Related Topics

Accessing Web Objects

Developing Server-Side stcl Scripts

How to Run stcl Scripts from Clients

How to Run stcl Server Scripts

Introduction to the DesignSync Object Model

The stcl Environment

Types of Web Objects

Working with Client stcl Scripts

Working with Server stcl Scripts

Synchronicity's Object Model

Introduction to the DesignSync Object Model

The DesignSync object model (SOM) is the data model that underlies the DesignSync software. By understanding SOM and the relationships among the web object types, you can more effectively use stcl commands to access and manipulate DesignSync data.

What Are SOM Web Objects?

SOM web objects:

- Are identified by a URL with a `sync:` or `file:` protocol, and can therefore live anywhere on the World Wide Web (WWW).
- Support one or more interfaces. Interfaces are sets of commands that are valid for a given web object type.
- Can have **properties** associated with them. Properties are name/value pairs that define attributes of a web object.
- Can have **notes** attached to them. Notes are web objects that contain information of any type, for example, bug reports, change requests, revision-control data, or threaded dialogues.
- Can be **versionable**. Versionable objects are objects that can be revision-controlled with multiple versions stored in a vault. Files and collection objects such as cellviews are examples of versionable web objects. A versionable object is a client-side object with a URL such as `file:///home/projadmin/Sample/top.v`. Associated with a versionable object under revision control are its server-side objects -- its vault (for example, `sync://syncinc:2647/Projects/Sample/top.v;`) and its versions (for example, `sync://syncinc:2647/Projects/Sample/top.v;1.2`).

Related Topics

Accessing Cadence Web Objects

Client-Side Versus Server-Side stcl

The stcl Environment

Types of Web Objects

Accessing Web Objects

On a SyncServer, there are two main types of data stored:

- DesignSync vault data represented by these web objects: File, Folder, Vault, Version, Branch, and Cellview, as well as tag names stored in a tags database.
- ProjectSync relational data, represented by these web objects: Project, Configuration, User, and Note.

Client-side scripts can access some of this data; server-side scripts can access all of this data.

Each SOM web object is identified by a URL. To access web objects on a SyncServer, you use the `sync:` protocol, a protocol based on http protocol and extended for SOM web objects. To access local web objects, you use the `file:` protocol in client-side scripts. See Types of Web Objects for examples of web objects and their URLs.

Keep these points in mind when you access web objects:

- Use the `url` and `note` command sets to access web objects.

  You can access a web object by specifying its URL. Additionally, you can access web objects using the url command set. The url commands let you navigate through the web object hierarchy without explicitly specifying web object URLs. See Accessing Objects Using url Commands for details. To access ProjectSync note web objects, see Accessing Notes.

- For server-side scripts, do not specify the host and port in your `sync:` URLs.

  For example, specify:

  ```
  sync:///Projects/Asic
  ```

  and not

  ```
  sync://chopin:2647/Projects/Asic
  ```

  Because the script is run on the server itself, `host:port` information is unnecessary and is stripped out by the server, which can lead to incorrect behavior during object-name comparisons. Also, omitting the `host:port` information makes your scripts more portable.

- The url commands are available from all DesignSync client shells. However, you cannot operate on a return value in dss/dssc, so the url commands are more useful in stcl/stclc.

- Quote semicolons in URLs using quotes (" ") or curly brackets ({ }). You can also escape semicolons using the backslash (\) character.

  Version and vault URLs contain semicolons (`sync://localhost:2647/Projects/Asic/x.v;`). Because the semicolon is a Tcl command separator, you must quote semicolons in URLs with quotes or curly braces, or use the backslash (\ ) escape character:

  ```
  "sync://localhost:2647/Projects/Asic/x.v;"

  {sync://localhost:2647/Projects/Asic/x.v;}

  sync://localhost:2647/Projects/Asic/x.v\;
  ```

- Wrap commands that access web objects in Tcl `catch` statements.

  Commands that access web objects, such as `url users` and `url getprop`, should be wrapped in `catch` statements because if bad property names or bad URLs are passed into a script, they can cause exceptions to occur. See Return Values and Exception Handling.

- Avoid special characters in URLs.

  Avoid the following characters in your URL names:

  ```
  @ # \ / ? * ; & |
  ```

  These characters, while not necessarily illegal, are used for specific purposes in URLs and thus can cause problems under some circumstances.

Related Topics

Client-Side Versus Server-Side stcl

Introduction to the DesignSync Object Model

The stcl Environment

Types of Web Objects

Types of Web Objects

The most familiar web objects are files and folders (directories). The DesignSync Object Model (SOM) supports many other web object types such as projects, configurations, vaults, and versions. The SOM web objects are listed in the table below. You identify these web objects using the URL formats illustrated in the table.

Note: Web objects also have object **states**, such as whether an object is a replica or a vault reference. See DesignSync Data Manager User's Guide: Object States: Original, Replica, Reference, and Link, for details about object states. To determine the state of an object, you can use the `url fetchedstate` command.

Object Type	Description
File	A file-system file. Specify files on the client with the `file:` protocol: `file:///home/projadmin/Sample/top.v`

Folder	A file-system directory. Specify folders on the client with the `file:` protocol. Specify server-side folders with the `sync:` protocol:
	
	`file:///home/projadmin/Sample/Alu`
	
	`sync://cae22:2647/Projects/Sample/Alu`
	
	A folder that is part of a vault structure on a SyncServer is referred to as a vault folder. Folders themselves are not revision-controlled objects, but do appear within vaults to maintain the hierarchy of vault objects.

Vault	The repository of the versions checked in for a particular design object; also contains branches. Note that a vault stores versions of a single design object. A "project vault" refers to the top-level vault folder for a project and thus contains a vault for each revision-controlled object in the folder.
	
	`sync://cae22.appco.com:2647/Projects/gemini/block1/top.v;`
	
	Note that vault names must include a terminating semicolon (;). If used in stcl commands, the semicolon needs to be delimited with quotes, curly braces, or a backslash escape character:
	
	`"sync://cae22:2647/Projects/Asic/x.v;"`
	
	`{sync://cae22:2647/Projects/Asic/x.v;}`

```
sync://cae22:2647/Projects/Asic/x.v\;
```

Version

A fixed snapshot of a design object, such as a file or collection, stored in a vault. You specify a version as a URL ending with a semicolon (;) followed by a dot-numeric version identifier. The dot-numeric identifier for a version contains an even number of elements (for example, `top.v;1.2` or `top.v;1.23.2.2`); whereas, the dot-numeric identifier for a branch contains an odd number of elements (for example, `top.v;1` or `top.v;1.23.2`).

```
sync://cae22:2647/Projects/Sample/top.v;1.2

sync://cae22:2647/Projects/Sample/top.v;1.2.2.1
```

Note that version names must include a semicolon (;). If used in stcl commands, the semicolon needs to be delimited with quotes, curly braces, or a backslash escape character:

```
"sync://cae22:2647/Projects/Asic/x.v;1.2"

{sync://cae22:2647/Projects/Asic/x.v;1.2}

sync://cae22:2647/Projects/Asic/x.v\;1.2
```

A special type of version is a **branch-point version** -- a version that is the root of a new branch. For example, if you create a branch by checking out version 1.2 where versions beyond 1.2 already exist, the branch-point version is designated 1.2, the branch is designated 1.2.1, and the first version on that branch is 1.2.1.1.

Branch

A thread of development of a revision-controlled object. You specify a branch as a URL ending with a semicolon (;) followed by a dot-numeric identifier. The dot-numeric identifier for a branch contains an odd number of elements (for example, `top.v;1` or `top.v;1.23.2`); whereas, the dot-numeric identifier for a version contains an even number of elements (for example, `top.v;1.2` or `top.v;1.23.2.2`). The main branch is designated with the version number 1. A branch URL is formed by appending the branch name to the vault URL:

Vault URL: `sync://cae22:2647/Projects/Sample/top.v;`

Branch URLs:

`sync://cae22:2647/Projects/Sample/top.v;1`

`sync://cae22:2647/Projects/Sample/top.v;1.2.2`

Note that the main branch of an object (for example, `sync://.../Sample/top.v;1`) is equivalent to the object's vault (`sync://.../Sample/top.v;`). You can use the vault URL to specify the branch in this case, but in most cases it is best to explicitly specify the branch URL.

Cadence View

A file system file that is one of a collection of files that make up a Cadence Design Systems view object. A Cadence Design Systems view object represents one of several views of a cell, where cells are the building blocks of chips or systems and together make up Cadence libraries. For example, a NAND2 cell may have four views: Verilog description, schematic symbol, schematic, and layout. Views are an example of a broader category of DesignSync objects called collection objects, where multiple files are revision-controlled as a single object. DesignSync software understands how to recognize views and creates an object called `<name>.sync.cds`, where `<name>` corresponds to the name of the view folder:

`file:///home/projLeader/ttlLib/and2/symbol.sync.cds`

Cadence View Folder

A Cadence Design Systems view container. Cadence views are stored in view folders:

`file:///home/projLeader/ttlLib/and2/symbol`

Cadence View Member

One of the files stored in a Cadence Design Systems view collection object:

`file:///home/projleader/ttlLib/and2/symbol/symbol.cdb`

Project

A group of revision-controlled objects stored together as a single design effort. You can create and manage projects using ProjectSync. By convention, all projects reside under the `Projects` folder under a SyncServer's root. Projects are vault folders that have a `sync_project.txt` file in the folder, where the `sync_project.txt` file contains information about the project. A project URL is the same as a vault folder's URL:

```
sync://cae22:2647/Projects/Sample
```

Configuration — A snapshot of design object versions making up a design project. You create a configuration in ProjectSync and then identify the contents of a configuration by tagging the versions of your design objects with a tag that corresponds to the ProjectSync configuration. A configuration is identified by a project URL followed by an ampersand (@) and configuration name:

```
sync://cae22:2647/Projects/Sample@Gold
```

UserProfile — A user profile for a DesignSync software user. You manage DesignSync users using the DesignSync web interface. User profiles are stored under the `Users` folder:

```
sync:///Users/<userid>
```

The user profile contains data such as the user's name, email address, and telephone numbers.

Note — A note is an informational object that you can attach to other web objects. A note is an instance of a particular note type, such as a bug report, change request, revision control note, or a web-based discussion. Users can create notes using ProjectSync. Notes can also be generated automatically during revision control operations if you set up revision control notes. Notes can also be created and attached to objects programmatically using the `note create` and `note attach` commands.

A note is identified by a URL of this format:

```
sync:///Note/SyncNotes/notetype/id
```

For example:

```
sync:///Note/SyncNotes/SyncDefect/19830
```

Note type — A note type is a collection of properties that define a note. Examples of note types include bug reports, change requests, revision control notes, and web-based discussions. You set up note types in ProjectSync. See ProjectSync Help for a description of the built-in property types used to create note type properties. The note type name you create using the ProjectSync Note Type Manager must start with a letter followed by any number of letters, numbers, and underscores. Other special characters, such as dashes and spaces, are

not allowed in note type names.

A note type is identified by a URL of this format:

```
sync:///Note/SyncNotes/notetype
```

Related Topics

Introduction to the DesignSync Object Model

Working with Revision Control Objects

Accessing Objects Using url Commands

The `url` commands are available from all DesignSync client shells. However, you cannot operate on return values in dss shells, so the url commands are more useful in stcl shells, and especially for stcl scripting.

url Commands and Revision Control Objects

url branchid - Returns the branch number of the specified object
url configs - Returns the configurations of a ProjectSync project
url container - Returns the object containing a specified object
url contents - Returns the objects in a container object
url exists - Returns whether an object exists
url fetchedstate - Returns the fetched state of an object
url fetchtime - Returns when an object was fetched
url getprop - Get a property of an object
url inconflict - Checks if a file merge had conflicts
url leaf - Returns the leaf of the URL
url locktime - Returns when a branch was locked
url members - Returns the members of the specified collection
url mirror - Returns the URL of a local directory's mirror
url modified - Checks if an object has been modified
url notes - Returns the notes attached to the specified object; server-side-only command
url owner - Returns the owner of an object
url path - Extracts the path section of a URL
url projects - Returns a SyncServer's ProjectSync projects
url properties - Returns property/value pairs for the specified object
url registered - Checks whether an object is under revision control

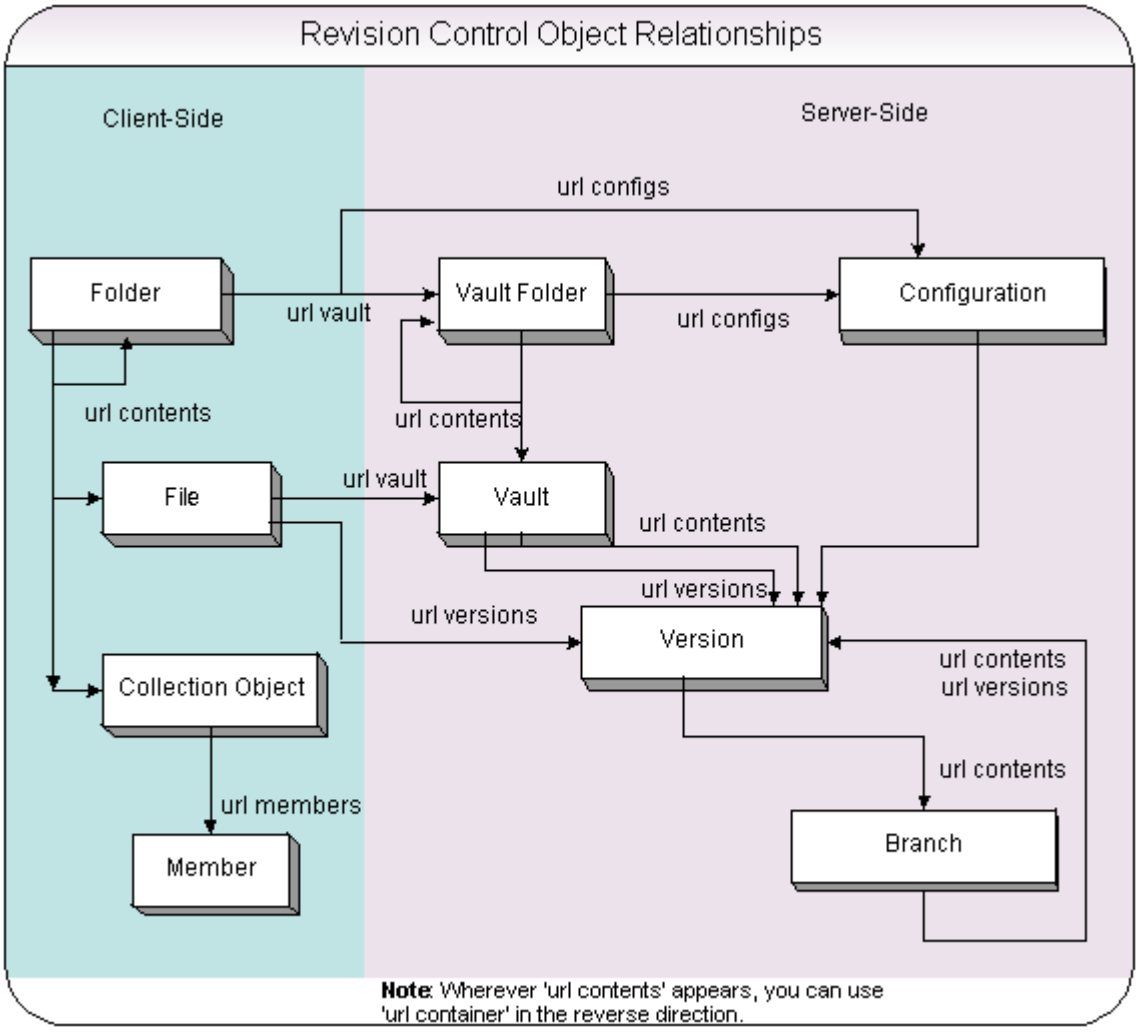
url relations - Determine collection object dependencies (assuming the relation, `dependencies`, is specified)
url resolveancestor - Returns the closest common ancestor of two versions
url resolvetag - Returns the version number associated with a selector
url retired - Returns whether a branch is retired
url selector - Returns the persistent selector list
url servers - Returns server-list definitions
url setprop - Set a property on an object
url syslock - Set a system lock on a lock name or file path
url tags - Returns the version tags of a specified object
url users - Returns all users defined for an object's server; server-side-only command
url vault - Returns the URL of an object's vault
url versionid - Returns the version number of the specified object
url versions - Returns the URLs of an object's versions

Note: `url` commands provide information about files and folders in your DesignSync workareas and their associated vaults. In general, do not use these commands to obtain information about mirrors; however, you can use the `url mirror` command to determine the mirror directory associated with your local workarea.

Relationships Between Revision Control Objects

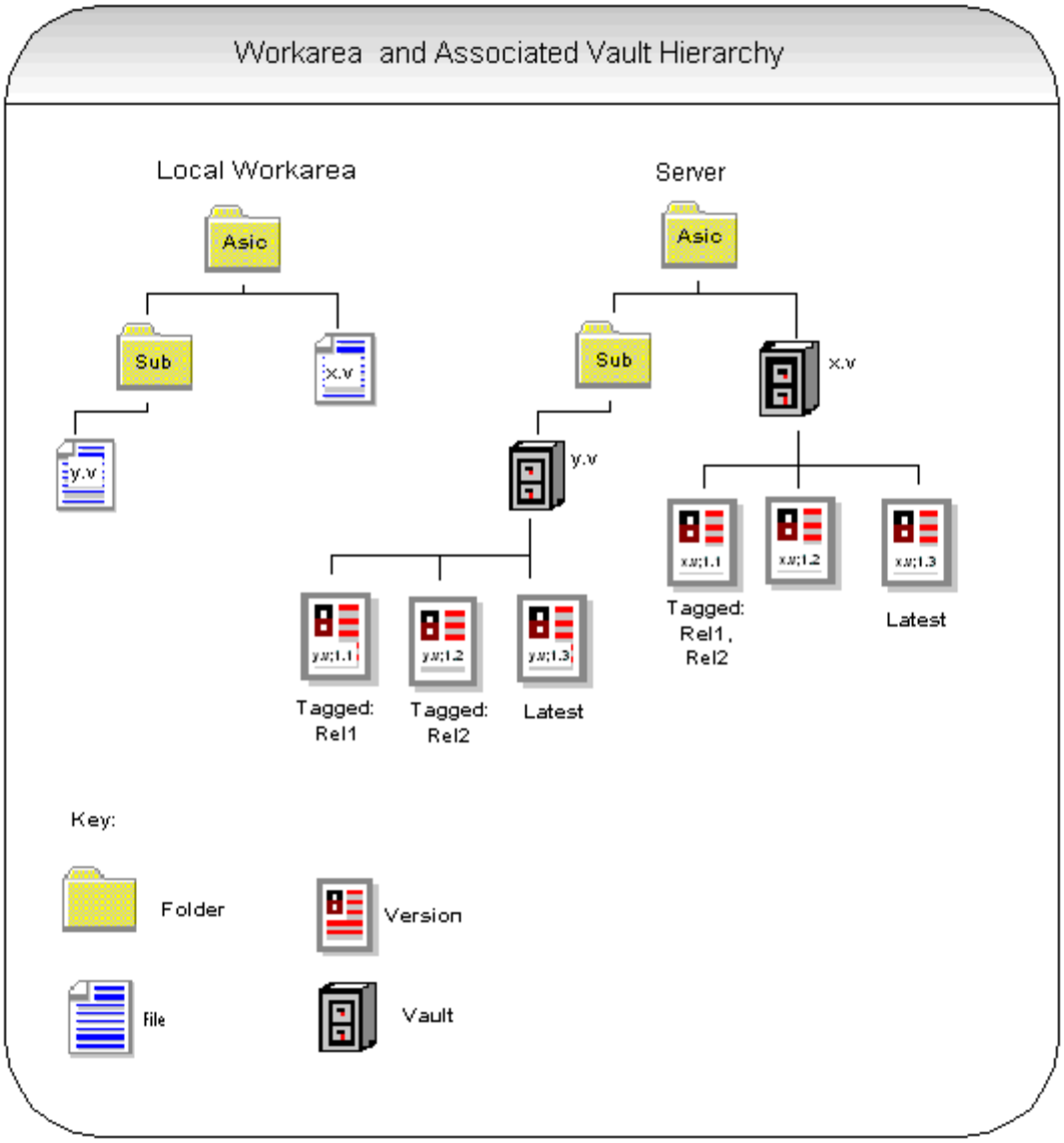
The url commands help you navigate through the SOM web objects. You can use these commands to determine relationships among web objects and to obtain their values and properties. For example, the `url contents` command returns the objects held in a container object and the `url container` command returns the container (parent) of a given object. These commands let you navigate up and down through the SOM hierarchy. Most url commands let you access web objects without having to type out URLs. In other words, you can use a relative path to specify web objects, rather than a URL. For example, the object specified by the URL, `file:///home/karen/Asic/x.v`, can also be specified as a relative path: `/home/karen/Asic/x.v`.

The following diagram illustrates the relationships among the web objects that exist in your local work area and on your team's SyncServer.



url Command Examples

The diagram below depicts a vault hierarchy and a workarea that has been associated with that vault hierarchy using the `setvault` command. The examples in the following sections refer to this diagram.



The following stcl session shows how you can use the url commands to access the Asic design objects shown in the preceding diagram. These url commands are illustrated within an stcl session, but you can use these commands in much the same way within an stcl script by using Tcl command substitution brackets, for example:

```
set filelist [url contents Asic]
```

For client-side scripts you include the SyncServer name and port number (for example, `sync://localhost:2647/Projects/Asic`) as in the session example below. For server-side scripts, you leave out the SyncServer name and port number (for example, `sync:///Projects/Asic`). Note that the return values are formatted here for easier reading, but actually are returned as one long string (no carriage returns after each value and no indentation).

stcl Session Example

```
stcl> url contents Asic
file:///home/karen/Asic/Sub
file:///home/karen/Asic/x.v
```

If you specify a clie
or file as an argume
command, the com
a list of local, client
specified with the `f`
protocol.

```
stcl> url contents Asic/x.v

stcl> url container Asic/x.v
file:///home/karen/Asic

stcl> url container Asic
file:///home/karen
```

Note that the `Asic`
not a container; thu
`contents Asic/`
command returns a

```
stcl> url contents sync://localhost:2647/Projects/Asic
sync://localhost:2647/Projects/Asic/Sub
{sync://localhost:2647/Projects/Asic/x.v;}
```

Here, the `sync:` pr
used; server-side v
folders are returned
contents' command

```
stcl> url contents sync://localhost:2647/Projects/Asic/Sub
{sync://localhost:2647/Projects/Asic/Sub/y.v;}

stcl> url contents
"sync://localhost:2647/Projects/Asic/x.v;1.1"

stcl> url contents
"sync:/localhost:2647/Projects/Asic/x.v;1"

stcl> url versions
"sync:/localhost:2647/Projects/Asic/x.v;1"

stcl> url contents
"sync://localhost:2647/Projects/Asic/x.v;"
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}
```

'url contents' of a S
folder
(`sync://.../Asi`
yields the folder's c
Note that 'url conte
version (`x.v;1.1`)
(`x.v;1`) returns an
whereas 'url conten
(`x.v;`) yields the v
in that vault.

```
stcl> url container sync://localhost:2647/Projects/Asic/x.v
```

There is no server-

```
sync://localhost:2647/Projects/Asic

stcl> url contents
sync://localhost:2647/Projects/Asic
sync://localhost:2647/Projects/Asic/Sub
{sync://localhost:2647/Projects/Asic/x.v;}
```

object, sync://.../Asi the 'url container' c returns its parent; t container' comman check for the existe object or its parent. the file, specify the file, either a relative (Asic/x.v) or usi file:// protocol (file://.../As

```
stcl> url contents sync://localhost:2647/Projects/Asic/x.v;
```

'url contents' of sync://.../x.v empty list. You mu URL containing a s

```
stcl> url contents
"sync://localhost:2647/Projects/Asic/x.v;"
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}
```

otherwise, the sem interpreted as a Tc separator. Use qu brackets, or the ba character to escap in a URL.

```
stcl> url vault Asic
sync://localhost:2647/Projects/Asic
```

'url vault' of a client (Asic) yields its va (sync://.../As

```
stcl> url vault Asic/x.v
sync://localhost:2647/Projects/Asic/x.v;
```

'url vault' of a file (A yields its vault (sync://.../As

```
stcl> url contents [url vault Asic/x.v]
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}
```

Note the use of the substitution bracke the [url vault] subst you from typing the

```
stcl> set xvault [url vault Asic/x.v]
sync://gilmour:30048/Projects/Asic/x.v;

stcl> url contents $xvault
{sync://gilmour:30048/Projects/Asic/x.v;1.1}
{sync://gilmour:30048/Projects/Asic/x.v;1.2}
```

Another way to sav typing a vault URL variable to the vaul can use the variabl in this example) to

```
{sync://gilmour:30048/Projects/Asic/x.v;1.3}

stcl> url exists ${xvault}1.1
1
```

value of the vault in
command.

```
stcl> url versions Asic/x.v
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}

stcl> url versions
"sync://localhost:2647/Projects/Asic/x.v;"
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}

stcl> url versions
"sync://localhost:2647/Projects/Asic/x.v;1"
{sync://localhost:2647/Projects/Asic/x.v;1.1}
{sync://localhost:2647/Projects/Asic/x.v;1.2}
{sync://localhost:2647/Projects/Asic/x.v;1.3}

stcl> url versions
"sync://localhost:2647/Projects/Asic/x.v"

stcl> url versions
"sync://localhost:2647/Projects/Asic@Rel1"

stcl>
```

'url versions' applie
side file (`Asic/x.v`
(`sync://.../Asi`
version
(`sync://.../Asi`
or a branch
(`sync://.../Asi`
you specify a serve
(`sync://.../Asi`
configuration
(`sync://.../Asi`
versions' returns ar
Note: If you apply '
to a branch-point v
branches are listed
example, if `x.v;1`
branch-point versio
versions' lists its ve
example, `x.v;1.1`
`x.v;1.1.2`, `x.v;`

```
stcl> url projects sync://localhost:2647

stcl> url configs Asic
```

No projects or conf
have been set up y
SyncServer. The S
administrator must
projects and config
ProjectSync. (See
User's Guide to set
and configurations.

```
stcl> url projects sync://localhost:30048
sync://localhost:30048/Projects/Thunder
sync://localhost:30048/Projects/Asic
```

The Asic project ha
created in ProjectS
as the Rel1 and Re
configurations. By

```
sync://localhost:30048/Projects/CPU

stcl> url configs Asic
sync://gilmour:30048/Projects/Asic@Rel1
sync://gilmour:30048/Projects/Asic@Rel2
```

ProjectSync projec
are implicitly assoc
ProjectSync Asic p
DesignSync Asic p
Likewise, by namin
ProjectSync config
and Rel2, we are in
associating the Rel
ProjectSync config
the Rel1 and Rel2
DesignSync shown
example.

```
stcl> tag Rel1 "[url vault Asic/Sub/y.v]1.1"

Beginning Tag operation...

Tagging:     sync://gilmour:30048/Projects/Asic/Sub/y.v;1.1
:
Added tag 'Rel1' to version '1.1'

Tag operation finished.

{Objects succeeded (1)} {}

stcl> tag Rel2 "[url vault Asic/Sub/y.v]1.2"

Beginning Tag operation...

Tagging:     sync://gilmour:30048/Projects/Asic/Sub/y.v;1.2
: Added tag 'Rel2' to version '1.2'

Tag operation finished.

{Objects succeeded (1)} {}
```

Here we use the R
tag names which c
the Rel1 and Rel2
we set up in Projec
how we can use th
command substitut
needn't type out th

```
stcl> url tags
"sync://localhost:2647/Projects/Asic/Sub/y.v;1.1"

Rel1

stcl> url tags
"sync://localhost:2647/Projects/Asic/Sub/y.v;1.2"
```

Apply 'url tags' to th
we just tagged to v

```
Rel2


stcl> url tags Asic/x.v

Latest

stcl> url tags sync://localhost:30048/Projects/Asic/x.v

SomAPI-E-101: Object does not exist at
specified URL

stcl> url tags "sync://localhost:30048/Projects/Asic/x.v;"

stcl> url tags
"sync://localhost:30048/Projects/Asic/x.v;1.1"

Rel1 Rel2
```

You apply the 'url t
to the client-side fil
or the server-side v
(`sync://../Asi`

Notice that there is
side file object,
`sync://.../Asi`
'url tags' command
error. To specify th
the client-side file,
relative specificatio
or using the file: pr
(`file:///.../As`

Do not apply the 'u
command on the va
(`sync://.../As`

```
stcl> url contents
"sync://localhost:2647/Projects/Asic@Rel1"

sync://localhost:2647/Projects/Asic/Sub@Rel1
{sync://localhost:2647/Projects/Asic/x.v;1.1}

stcl> url contents
"sync://localhost:2647/Projects/Asic/Sub@Rel1"

{sync://localhost:2647/Projects/Asic/Sub/y.v;1.1}
```

For configurations
is not recursive; yo
'url contents' on ea
within the project to
objects in a configu

```
stcl> url resolvetag -version Rel1 Asic/x.v

1.1

stcl> url resolvetag -version Rel1 Asic/Sub/y.v

1.1
```

Use 'url resolvetag'
which version of an
corresponds to a pa
selector, tag, or co
name.

The `url` commands operate on various web objects. Most `url` commands let you specify web objects using a relative path within your workarea or an absolute URL path. For example, both of the following are valid:

```
stcl> url vault .           # relative

stcl> url vault [spwd]      # absolute
```

(The `url projects` and `url users` commands are exceptions in that they require absolute paths.)

If you apply a `url` command on an object that is not applicable, an error is typically raised. For these cases, you use a Tcl `catch` statement to catch errors returned from commands in your scripts. For some objects that do not apply to a `url` command, the command returns an empty list. See Return Values and Exception Handling for details.

To help you decide whether you need to create an exception handler for a particular `url` command, consult the `url` command descriptions in the ENOVIA Synchronicity Command Reference. The `url` command descriptions list the types of web objects supported by each `url` command, along with the return values for each type of revision control object.

Tips for Accessing Revision Control Objects

Example: Traversing a Design Directory Using url Commands

You can traverse a design directory using url commands, as well as Tcl commands. The following script example, `syncRecurse.tcl`, in `<SYNC_DIR>/share/examples/doc/stclguide` shows how you can use `url contents` followed by `url getprop` to list and determine the types of objects in a design directory.

```
proc syncRecurse arg {
  foreach obj [url contents $arg] {
    if {[url getprop $obj type] == "Folder"} {
      puts [format "Folder: %s\n" [url path $obj]]
      syncRecurse $obj
    } else {
      puts [format "Object: %s\n" [url path $obj]]
    }
  }
}
```

Example: Traversing a Design Directory Using Tcl Commands

You can interchange standard Tcl commands with the url commands. The following script, `syncRecurse2.tcl`, uses the Tcl `file` command to determine the file types of objects in a design directory.

```
proc syncRecurse2 arg {
  foreach obj [url contents $arg] {
    if {[file isdirectory [url path $obj]]} {
      puts [format "Folder: %s\n" [url path $obj]]
      syncRecurse2 $obj
    } else {
      puts [format "Object: %s\n" [url path $obj]]
    }
  }
}
```

Example: Listing Object States in a Design Directory

If you want more detailed information about the objects in a design directory, use the `url fetchedstate` command. The following script example, `syncRecurse3.tcl`, shows how to traverse a design directory, listing the state of the object -- for example, whether the object is a replica or a reference.

```
proc syncRecurse3 arg {
  foreach obj [url contents $arg] {
    if {[url getprop $obj type] == "Folder"} {
      puts [format "Folder: %s\n" [url path $obj]]
      syncRecurse3 $obj
    } elseif {[url fetchedstate $obj] == "Lock"} {
      puts [format "Locked: %s\n" [url path $obj]]
    } elseif {[url fetchedstate $obj] == "Copy"} {
      puts [format "Replica: %s\n" [url path $obj]]
    } elseif {[url fetchedstate $obj] == "Mirror"} {
      puts [format "Mirror: %s\n" [url path $obj]]
    } elseif {[url fetchedstate $obj] == "Cache"} {
      puts [format "Cache: %s\n" [url path $obj]]
    } elseif {[url fetchedstate $obj] == "Reference"} {
      puts [format "Reference: %s\n" [url path $obj]]
    } elseif {[url fetchedstate $obj] == "NotFetched"} {
      puts [format "NotFetched: %s\n" [url path $obj]
    }
  }
}
```

The following stcl session shows how `syncRecurse3` traverses the Asic hierarchy:

```
stcl> source /home/karen/tclscripts/syncRecurse3.tcl
```

```
stcl> syncRecurse3 Asic

Folder: /home/karen/Asic/Sub

Replica: /home/karen/Asic/Sub/y.v

Reference: /home/karen/Asic/Sub/y.y

Locked: /home/karen/Asic/x.v
```

Example: Scanning a Workarea for Files Needing Check-In

You can use the `url modified` and `url registered` commands when you traverse your design directory to scan for design files that have been modified or new files that you might want to put under revision control. The url modified command returns both objects under revision control that have been modified and objects that are not under revision control. Thus, you can use the `url registered` command to detect new objects that you might want to put under revision control. The following script, `syncNeedCheckin.tcl`(in the `<SYNC_DIR>/share/examples/doc/stclguide` directory), checks each object in the design hierarchy and prints out those objects that are under revision control (registered) and have been modified, as well as objects not under revision control:

```
proc syncNeedCheckin arg {
  foreach obj [url contents $arg] {
    if {[url getprop $obj type] == "Folder"} {
      puts [format "Folder: %s\n" [url path $obj]]
      syncNeedCheckin $obj
    } else {
      if {[url modified $obj]} {
        if {[url registered $obj]} {
          puts [format "Registered object modified: %s\n"\
          [url path $obj]]
        } else {
          puts [format "Unregistered object: %s\n"\
          [url path $obj]]
        }
      }
    }
  }
}
```

Assume we have created a new file, `top.v`, in our `Asic` folder. Assume also that we have modifed `Sub/y.v`. Here are the results of running the `syncNeedCheckin.tcl` script on the `Asic` folder:

```
stcl> syncNeedCheckin Asic

Folder: /home/karen/Asic/Sub

Registered object modified: /home/karen/Asic/Sub/y.v

Unregistered object: /home/karen/Asic/top.v
```

See Also

Accessing Cadence Web Objects

Return Values and Exception Handling

Working with Properties of Revision Control Objects

Accessing Cadence Web Objects

Cadence web objects include cellviews, view folders, and view members. Cellviews (`<viewname>.sync.cds`) do not exist as physical files; only their members exist:

- View folder - A folder containing the set of files that make up a Cadence view
- View member - One of the set of files that make up a Cadence view

The following example illustrates stcl commands you can use to access Cadence web objects. In this example, the cellview, `symbol.sync.cds`, is represented by the following directory structure:

Cell folder: `file:///home/projleader/ttlLib/and2` contains:

- View folder: `file:///home/projleader/ttlLib/and2/symbol`
- Cellview: `symbol.sync.cds`

View folder: `file:///home/projleader/ttlLib/and2/symbol` contains:

- View member: `symbol.cdb`
- View member: `master.tag`
- View member: `pc.db`
- View member: `prop.xx`

Cadence Web Object Session Example

```
stcl> cd /home/projlead/Projects/ttlLib/and2

stcl> ls

Time Stamp Status  ...   Name
---------- ------  ----  ----
...  ...   ...     ...   symbol
...  ...   ...     ...   symbol.sync.cds
```

The cell folder (`and2`) has two objects, a view folder (`symbol`) and a view (`symbol.sync.cds`).

```
stcl> url getprop symbol.sync.cds type
Cadence View

stcl> url contents symbol.sync.cds

stcl> url contents symbol
file:///home/projlead/Projects/ttlLib/and2/symbol
/symbol.cdb
file:///home/projlead/Projects/ttlLib/and2/symbol
/master.tag
file:///home/projlead/Projects/ttlLib/and2/symbol
/pc.db
file:///home/projlead/Projects/ttlLib/and2/symbol
/prop.xx

stcl> url getprop symbol/symbol.cdb type
Cadence View Member
```

Notice that the property type of the symbol.sync.cds cellview is Cadence View. 'url contents' of the cellview returns the empty string because a cellview is not a physical object. 'url contents' of the symbol directory, a container object, returns the Cadence view members (for the `symbol` view, in this case).

```
...
stcl> url members symbol.sync.cds
file:///home/projlead/Projects/ttlLib/and2/symbol
/symbol.cdb
file:///home/projlead/Projects/ttlLib/and2/symbol
/master.tag
file:///home/projlead/Projects/ttlLib/and2/symbol
/pc.db
file:///home/projlead/Projects/ttlLib/and2/symbol
/prop.xx

....
```

Because a cellview is not a container object, you use the 'url members' command to determine its members. Note that this command is similar to 'url contents symbol' above that we used to list the contents of the `symbol` view folder. A view folder can contain objects that are not members of the cellview; thus, these commands are not

necessarily identical.

```
stcl> url container symbol/symbol.cdb
file:///home/projlead/Projects/ttlLib/and2/symbol

...
```

Notice that the container of a view member is the view folder--not the cellview.

```
stcl> cd
/home/projlead/Projects/adderLib/add16/schematic

stcl> url relations schematic.sync.cdb
dependencies
file:///home/projlead/Projects/ttlLib/and2/symbol
ttlLib:and2/symbol.sync.cds
file:///home/projlead/Projects/ttlLib/nor2/symbol
ttlLib:nor2/symbol.sync.cds

...
```

Use 'url relations' to list views on which this `add16 schematic.sync.cdb` view is dependent.

Related Topics

Accessing Objects Using url Commands

Working with Properties of Revision Control Objects

Working with Properties of Revision Control Objects

Revision control objects can have the following built-in properties that you can access with commands such as the `url properties` command:

Property	Description
name	The name of the specified object.
description	The generic description for the object, or an empty string if none exists.
type	The type of the specified object. Examples are File, Folder, Vault, Version, Branch, Project, and Project Configuration.

owner	The owner of the object. The following object types have owners: projects, project configurations, vaults, and branches. If owner is the only property you are interested in, use 'url owner'.
locked	The name of the user who has the object locked, or '0' if it is unlocked. A nonzero value can be expected only for files, vaults, branches, and versions. If a vault is specified, the default branch is examined. Specifying a file has the same effect as calling 'url locked' on the file's branch.
locktime	The time, in time_t format, that the object was locked (if the object is locked -- value of 'locked' property is nonzero), otherwise '0'.
citime	The time, in time_t format, that a version was created in the vault.
log	The log information for the specified object.
selector	The selector (tag) list associated with a ProjectSync project configuration that identifies the versions of DesignSync data that are part of the configuration.
exposure	The list of team members (usernames) associated with a project configuration.

Accessing Revision Control Object Properties

You can access these built-in revision control object properties using the `url properties` command. To determine user-defined properties, use the `url getprop` command. These commands are illustrated in the stcl session below:

```
stcl> url properties [url vault Asic/x.v]
 vaultprops
```

The 'url properties' command stores the array of property/value pairs in the supplied argument (`vaultprops`)

```
stcl> puts $vaultprops(name)
x.v;

stcl> puts $vaultprops(description)

stcl> puts $vaultprops(type)
Vault

stcl> puts $vaultprops(owner)
karen

stcl> puts $vaultprops(locked)
0

stcl> puts $vaultprops(locktime)
0

stcl> puts $vaultprops(citime)
980882754

stcl> puts $vaultprops(log)
Fixed syntax error.

stcl> puts $vaultprops(selector)

stcl> puts $vaultprops(exposure)
```

For this vault, the properties name, type, owner, locked, locktime, citime, and log are applicable. Properties that are not applicable return an empty list.

```
stcl> url properties [url vault
Asic]@Rel1 confprops

stcl> puts $confprops(name)
Rel1

stcl> puts $confprops(type)
Project Configuration

stcl> puts $confprops(description)
First release

stcl> puts $confprops(owner)
karen

stcl> puts $confprops(selector)
Rel1

stcl> puts $confprops(exposure)
sal alex karen
```

'url properties' of a configuration object stores values in the name, type, description, owner, selector, and exposure properties. The selector property indicates which tags are associated with the configuration created in ProjectSync. In this case, the tag name, Rel1, matches the configuration name. The exposure property is the list of users associated with the configuration.

```
stcl> url properties Asic folderprops

stcl> puts $folderprops(name)
Asic

stcl> puts $folderprops(locked)

can't read "folderprops(locked)":
no such element in array

stcl> puts $folderprops(locktime)

can't read "folderprops(locktime)":
no such element in array

stcl> puts $folderprops(citime)

can't read "folderprops(citime)":
no such element in array

stcl> puts $folderprops(log)

can't read "folderprops(log)":
no such element in array

stcl>
```

Notice that a folder cannot be revision-controlled so the locked, locktime, citime, and log properties are not applicable.

```
stcl> co -lock Asic/x.v -comment
"Fix bug"

Beginning Check out operation...

Checking out: Asic/x.v:
Success - Checked Out
version: 1.3 -> 1.4

Checkout operation finished.

{Objects succeeded (1)} {}

stcl> url properties [url vault
Asic/x.v]1.4 versionprops

stcl> puts $versionprops(name)
x.v;1.4

stcl> puts $versionprops(type)
Version

stcl> puts $versionprops(owner)

stcl> puts $versionprops(locked)
karen

stcl> puts $versionprops(locktime
981664986

stcl> puts $versionprops(citime)
0

stcl> puts $versionprops(log)
Fix bug
```

User karen has locked `x.v`. In this example, 'url properties' is applied to the pending version, 1.4. Notice that the locktime is accessible, but the citime returns 0 as the object has not yet been checked in. Notice that the owner property returns an empty list; owner only applies to projects, project configurations, vaults, and branches.

```
stcl> clock format
$versionprops(locktime)

 Thu Feb 08 15:43:06 EST 2001
```

You can use the Tcl `clock format` command to convert the `time_t` format to a date string.

```
stcl> url setprop [url vault Asic/x.v]
completed 1
1
```

You can create new properties using 'url setprop'. Use 'url getprop' to extract the

```
stcl> url setprop [url vault
Asic/Sub/y.v] completed 0
0

stcl> url getprop [url vault Asic/x.v]
completed
1

stcl> url getprop [url vault
Asic/Sub/y.v] completed
0
```

value of a user-defined property.

```
stcl> url getprop Asic/x.v type
File

stcl> url getprop Asic type
Folder
```

You can use 'url getprop' to return an object's type; however, 'url getprop' is generally used to obtain user-defined properties, not built-in properties.

Adding User-Defined Properties

You can create properties for revision control objects using the `url setprop` command. To extract a user-defined property, use the `url getprop` command. These commands are illustrated in the stcl session below, in which a new property, `completed`, is defined. **Note**: There is currently no method of listing all user-defined properties. If you want to be able to list user-defined properties, you can create an additional property, for example, `customprops`, to which you can add the new property names as you create them. Then use `url getprop` on the `customprops` property to obtain the list of properties for an object.

```
stcl> url setprop [url vault Asic/x.v]
completed 1
1

stcl> url setprop [url vault
Asic/Sub/y.v] completed 0
0
```

You can create new properties using 'url setprop'.

```
stcl> url getprop [url vault Asic/x.v]
completed
1
```

Use 'url getprop' to extract the value of a user-defined property.

```
stcl> url getprop [url vault
Asic/Sub/y.v] completed
0
```

Adding User-Defined Properties in Server-Side Scripts

You can also set properties in server-side scripts as in the following `syncSetPriority.tcl` script (in the `<SYNC_DIR>/share/examples/doc/stclguide` directory):

```
foreach project [url projects sync:///] {
  url setprop $project priority low
}
```

After running this script, the Asic project has property, priority, set to low:

```
stcl> url getprop sync://localhost:30048/Projects/Asic priority

low
```

Related Topics

Accessing Cadence Web Objects

Accessing Objects Using url Commands

Working with Notes

Accessing Notes

Notes are created and modified in the following ways:

- Manually with ProjectSync when a team member adds or modifies a note.
- Automatically when a DesignSync operation occurs and attaches a revision control note to a DesignSync object. (See DesignSync Data Manager User's Guide: RevisionControl Notes Overview to learn how to set up these types of notes.)

Likewise, you create and modify note types using the ProjectSync NoteType Manager. From there, you can create, modify, rename, and delete note types.

However, if you want to make changes to a number of notes or note types, you can update the note and note type databases programmatically using the `note`, `notetype`, `ptype`, and `url` commands.

The commands that access and update notes and note types are server-side commands. **Note**: When you develop scripts that work with notes and note types, always use the `sync:///` protocol syntax (no `<host>:<port>` specification).

Tips for Accessing Notes and Properties

Example: Using url Commands to Extract Note Values

You can use the `url leaf`, `url path`, and `url container` commands to extract values from the pathnames of note objects. For example, the note directory hierarchy contains the note type and note IDs for notes. Here is a sample note URL:

```
sync:///Note/SyncNotes/HW-Defect-1/1
```

The following commands extract the note type and note ID:

```
puts "Notetype: [url leaf [url container $noteURL]]"
puts "Note Id: [url leaf [url path $noteURL]]"

Notetype: HW-Defect-1
Note Id: 1
```

Although these `url` commands can be used on the client side, if you are using them to access notes or note types, you must incorporate them in a server-side script. See url Commands to Access Notes for more information about `url` commands and notes.

Example: Using note Commands to Extract Note Values

You can use the `url notes` command to obtain all of the notes in a ProjectSync project. Then you can use `note getprop` to access specific properties (fields) of the note.

The following server-side script, `lsNoteProps.tcl`, lists the titles of all notes in all the SyncServer projects:

```
foreach project [url projects sync:///] {
  foreach note [url notes $project] {
    puts <pre>
    puts "Project: $project"
    puts "NoteURL: $note"
    puts "Notetype: [url leaf [url container $note]]"
    puts "Note Id: [url leaf [url path $note]]"
```

```
    puts "Note Title: [note getprop $note Title]"
    puts </pre>
  }
}
```

See note Commands to Access Notes for more information about `note` commands.

Example: Using note query to List Notes

You can use the `note query` command to obtain a group of notes based on criteria you set. The following example displays only notes of type Note that are attached to the Asic project:

```
puts [note query -type Note -attached sync:///Projects/Asic]
```

Example: Using note links to List Objects and Attached Notes

You can use the `note links` command to list the attachments between notes and web objects. A note link is a list that pairs a web object and the note attached to the web object.

Example: To list the objects to which a note is attached:

Use the `-note` option of the `note links` command:

```
puts [note links -note sync:///Note/SyncNotes/Note/6697]
```

This command generates a list of the web objects to which note 6697 is attached (formatted here to make it easier to read):

```
sync:///Projects/Asic
sync:///Projects/Asic@Rel1
{sync:///Projects/Asic/x.v;}
{sync:///Projects/Asic/x.v;1.4}
{sync:///Projects/Asic/x.v;1}
sync:///Notes/SyncNotes/Note/6677
```

Example: To list the notes attached to an object:

Use the `-object` option of the `note links` command:

```
puts [note links -object sync:///Projects/Asic]
```

This command generates a list of the notes attached to the Asic project (formatted here):

```
sync:///Note/SyncNotes/SyncDefect/1
sync:///Note/SyncNotes/SyncDefect/2
sync:///Note/SyncNotes/Note/3
sync:///Note/SyncNotes/Note/6697
```

Example: To list all note attachments:

Apply the `note links` command with no arguments:

```
puts [note links]
```

This command generates note links of web objects and their attached notes for all web objects on the SyncServer. You cannot rely on the order of these lists; for example, notice that the note links for the `sync:///Projects/Asic` project are not grouped together:

```
{{sync:///Projects/Asic} {sync:///Note/SyncNotes/Note/1}}
{{sync:///Projects/CPU} {sync:///Note/SyncNotes/Note/1}}
{{sync:///Projects/CPU} {sync:///Note/SyncNotes/Note/2}}
{{sync:///Projects/Asic} {sync:///Note/SyncNotes/Note/2}}
{{sync:///Projects/Asic} {sync:///Note/SyncNotes/Note/3}}
{{sync:///Projects/Asic@Rel1} {sync:///Note/SyncNotes/Note/4}}
{{sync:///Projects/Asic@Rel1} {sync:///Note/SyncNotes/Note/5}}
{{sync:///Projects/Asic/x.v;} {sync:///Note/SyncNotes/Note/5}}
...
...
```

note Commands to Access Notes

You can use `note` commands within server-side scripts to access notes. Use `notetype` commands and `ptype` commands to work with the properties of notes (reflected in a note's fields). See Working with Note Types to work with note types and properties.

The following commands access notes and their properties:

- `note counts` - Computes statistics about notes and the frequency of values
- `note getprop` - Retrieves a property of a note
- `note links` - Returns note link data
- `note query` - Queries the note system and returns note URLs
- `note setprops` - Sets field values on a note
- `note types` - Returns a list of defined note types
- `notetype enumerate` - Returns a list of defined note types
- `notetype getdescription` - Returns a brief description of a specified note type
- `notetype schema` - Extracts information about a note type's schema

url Commands to Access Notes

Some of the url commands are useful for accessing notes. The following subset of the `url` commands let you access and work with notes. To use these `url` commands with notes, include them in server-side scripts.

url Command	Examples

url container - Returns the object containing a specified object	`puts "Note type URL: [url container \ sync:///Note/SyncNotes/SyncDefect/40]"`
	Returns:
	`Note type URL: sync:///Note/SyncNotes/SyncDefect`

url exists - Returns whether an object exists	`puts "Note exists? [url exists \ sync:///Note/SyncNotes/SyncDefect/40]"`
	Returns:
	`Note exists? 1`

url getprop - Get a property of an object	`puts [url getprop $newnoteURL Title]`
	Returns:
	`Proposal for Parallel Architecture`

url leaf - Returns the leaf of the URL	`puts "Note Id: [url leaf [url path \ $noteURL]]"`
	Returns:
	`Note Id: 40`

url notes - Returns the notes attached to the specified object (server-side-only command)	`foreach note [url notes \ "sync:///Projects/Asic"] {`
	`  puts "$note"`
	`}`

Returns:

```
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697
```

url path - Extracts the path section of a URL

```
puts "Note Id: [url leaf [url path \
$noteURL]]"
```

Returns:

```
Note Id: 40
```

url projects - Returns a SyncServer's ProjectSync projects

```
puts [url projects sync:///]
```

Returns:

```
sync:///Projects/BestProj
sync:///Projects/NotSoGoodProj
sync:///Projects/Lightening
sync:///Projects/Thunder
```

url properties - Returns properties for the specified object

```
url properties \
"sync:///Note/SyncNotes/Note/6677" \ Props

foreach prop [array names Props] {
  puts "Prop $prop=$Props($prop)<BR>"
}
```

Returns:

```
Prop Id=6677
Prop Body=Main portion of a note.
Prop CCList=
Prop DateCreate=37759.567801
Prop Title=Hello, World!
Prop Author=karen
```

url servers - Returns server-list definitions

```
puts [url servers -user]
```

Returns:

```
{{My Server} {sync://localhost:2647} {}}
{{Source} {sync://src.myco.com:3001} {The
company-wide source repository.}}
```

url setprop - Set a property on an object; to set multiple note properties, use note setprops

Note: See Updating Notes for examples.

url users - Returns all users defined for an object's server (server-side-only command)

```
foreach user [url users sync:///] {
  puts "[url getprop $user name]<br>"
}
```

Returns:

```
Charles Dent
Asic Developers
Jean Boswell
Anabel Blythe
```

Related Topics

Creating and Attaching Notes

Typically notes are created manually by ProjectSync users or automatically by DesignSync revision control operations. You can create server-side scripts to programmatically generate notes using the `note create` and `note attach` commands:

- `note create` - Creates a new note
- `note attach` - Creates a link between a note and an object

There are two steps involved in generating notes. You first create the note, then, you attach it to a web object. You can attach notes to any server-side web object, including revision control objects and notes themselves.

The following server-side script (in the `<SYNC_DIR>/share/examples/doc/stclguide` directory) illustrates how you can create a note and attach it to any type of web object.

genNotes.tcl Script

```
set newnoteURL [note create -type Note
\
-date 1234.5678 \
{Title "Hello, World!"} \
{Body "Main portion of a note."} \
{Author karen}]
```

Create a note of type Note with Title, Body, and Author properties.

```
note attach $newnoteURL \
sync:///Projects/Asic

note attach $newnoteURL \
sync:///Projects/Asic@Rel1

note attach $newnoteURL \
"sync:///Projects/Asic/x.v;"

note attach $newnoteURL \
"sync:///Projects/Asic/x.v;1.4"

note attach $newnoteURL \
"sync:///Projects/Asic/x.v;1"

note attach $newnoteURL \
"sync:///Projects/Asic/nothing.v;"

note attach $newnoteURL \
sync:///Notes/SyncNotes/Note/6677
```

Attach the new note to a project, a configuration, a vault, a version, a branch, a nonexistent file, and a note. The 'note attach' command does not verify whether the object exists, so no error message is raised when the note is attached to `nothing.v`, a nonexistent file. Quotes are used if the web object's URL contains a semicolon or a space.

```
puts <PRE>

puts "Project Asic Notes:"
foreach note [url notes \
"sync:///Projects/Asic"] {
  puts "$note"
}

puts "Config Asic@Rel1 Notes:"
foreach note [url notes \
```

The lists of notes attached to all web objects are printed. Because genNotes.tcl is a server-side script, the output format is HTML--the <PRE> tag specifies preformatted text.

```
"sync:///Projects/Asic@Rel1"] {
  puts "$note"
}

puts "Vault x.v; Notes:"
foreach note [url notes \
"sync:///Projects/Asic/x.v;"] {
  puts "$note"
}

puts "Version x.v;1.4 Notes:"
foreach note [url notes \
"sync:///Projects/Asic/x.v;1.4"] {
  puts "$note"
}

puts "Branch x.v;1 Notes:"
foreach note [url notes \
"sync:///Projects/Asic/x.v;1"]
  puts "$note"
}

puts "Vault y.v -- No Notes:"
foreach note [url notes \
"sync:///Projects/Asic/Sub/y.v"] {
  puts "$note"
}

puts "Nonexistent file Notes:"
foreach note [url notes \
"sync:///Projects/Asic/nothing.v;"] {
  puts "$note"
}

puts "Note attached to Note:"
foreach note [url notes \
sync:///Note/SyncNotes/Note/6677] {
  puts "$note"
}

puts </PRE>
```

The vault y.v has no notes attached, so 'url notes' of vault y.v outputs an empty list. The file nothing.v does not exist, but 'note attach' does not verify this; thus, 'url notes' lists attachments for nothing.v.

Sample Results from the genNotes.tcl Script

```
Project Asic Notes:
```

The lists of notes

```
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697

Config Asic@Rel1 Notes:
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697

Vault x.v; Notes:
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697

Version x.v;1.4 Notes:
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697

Branch x.v;1 Notes:
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697
```

attached to all web objects are printed.

```
Vault y.v Notes:
```

The vault `y.v` has no notes attached, so 'url notes' of vault `y.v` outputs an empty list.

```
Nonexistent file Notes:
sync:///Note/SyncNotes/Note/6694
sync:///Note/SyncNotes/Note/6695
sync:///Note/SyncNotes/Note/6696
sync:///Note/SyncNotes/Note/6697

Note attached to Note:
sync:///Note/SyncNotes/Note/6697
```

The file `nothing.v` does not exist, but 'note attach' does not verify this; thus, 'url notes' lists attachments for `nothing.v`.

Related Topics

Accessing Notes

Updating Notes

Working with Note Types

Working with Note Types

For most work with note types, it is best to use ProjectSync's Notetype Manager. You can use the Notetype Manager's graphical user interface to create, delete, and rename note types. You can also change properties of existing note types using the Notetype Manager. If you want to programmatically create, delete, or rename note types, use the `notetype create`, `notetype delete`, and `notetype rename` commands. See ProjectSync User's Guide for the list of predefined property types you use to create properties for your note types. You can also use the `ptype` commands to query existing property types.

Following are the commands that let you work with note types and query property types:

`notetype create` - Creates a new note type

`notetype delete` - Deletes the specified note type

`notetype rename` - Renames an existing note type

`ptype choices` - Returns the choice list of an enumerated type

`ptype class` - Returns the class of a property type

`ptype enumerate` - Returns a list of all property types

`ptype is` - Tests whether a prop type is of a certain class

`ptype strwidth` - Returns the maximum width for strings of this type

`ptype transitions` - Returns valid next states for a state machine

Determining Properties of a Note Type

Before you begin making changes to notes, you can determine the properties that currently exist for a note type by using the `notetype schema` command. Include the following line in a server-side script:

```
puts [notetype schema HW-Defect-1]
```

Here is the schema, or list of properties, that is output for the built-in HW-Defect-1 note type :

```
Submod Manager Product Title Doc Tools Platform DateCreate
Author Severity Id Info CCList Body Keywords Status Foundby
Waiting Resp
```

Related Topics

Accessing Notes

Creating and Attaching Notes

Updating Notes

Updating Notes

You can use ProjectSync to update the values of note properties (fields). However, if you want to make programmatic changes to your notes database, you can use the `note` and `url` commands. The following note commands help you access and update notes:

- `note delete` - Deletes a note and associated note links
- `note detach` - Deletes the link between a note and an object
- `note getprop` - Retrieves a property of a note
- `note links` - Returns note link data
- `note query` - Queries the note system and returns note URLs
- `notetype schema` - Extracts information about a note type's schema
- `note setprops` - Sets field values on a note

Determining Properties and Values of a Note

Before you begin making changes to notes, you can view the properties and values that currently exist for your notes by using the `url properties` command in a server-side script. The following lines in a server-side script print out the properties and values for all notes in the Asic project:

```
foreach note [url notes sync:///Projects/Asic] {
  puts "Note URL: $note"
  url properties $note Props
  foreach prop [array names Props] {
    puts "Prop $prop=$Props($prop)<BR>"
```

```
   }
}
```

Updating Values for Particular Properties

You can use the `url setprop` command to update a single property and value pair on a note. Use the `note setprops` command to update several properties at once. To determine the property names of the properties you want to change for a note, use the `notetype schema` command:

```
puts [notetype schema "Note"]
```

Returns:

```
Id DateCreate Title Body CCList Author
```

The `modNotes.tcl` script (in the `<SYNC_DIR>/share/examples/doc/stclguide` directory) updates the `CCList` and `Title` properties of the notes attached to the "`x.v;`" vault. Because two properties are updated, the script uses the `note setprops` command rather than the `url setprop` command.

modNotes.tcl Script

```
puts <PRE>
foreach note [note query -type Note \
-attached "sync:///Projects/Asic/x.v;"] {

  puts "NoteURL: $note <BR>"
  puts "Title: [note getprop \
  $note Title] <BR>"
  puts "CCList: [note getprop \
  $note CCList] <BR>"
  puts "Body: [note getprop \
  $note Body] <BR>"



  set newCCList ""
  set newTitle ""

  append newCCList "asicdev " \
  [note getprop $note CCList]

  append newTitle "x.v Note: " \
  [note getprop $note Title]
```

Query for the notes attached to the `x.v;` vault and print out properties.

Set temporary variables to an empty list. Append new string "asicdev " to the current CCList. Append new string "x.v Note: " to the

current Title.

```
  note setprops $note CCList \
  $newCCList Title $newTitle
```

Set the new properties for CCList and Title. Use 'note setprops' rather than 'url setprop' because more than one property is being updated.

```
  puts "New CCList: [note getprop \
  $note CCList] <BR>"
  puts "New Title: [note getprop \
  $note Title] <BR>"
}
puts </PRE>
```

Print the new properties to verify.

Sample Results from the modNotes.tcl Script

```
NoteURL: sync:///Note/SyncNotes/Note/6712

Title: Inefficient verilog code

CCList: sal karen

Body: *** Original text on Feb 23 2001,
13:40:48 (GMT 5:00) by  karen ***

Module x.v needs to be rewritten.  The
code would be faster if the behavioral
statements are replaced by component
instances.

New CCList: asicdev sal karen

New Title: x.v Note: Inefficient verilog
code
```

Notice the new CCList and Title properties in Note 6712.

```
NoteURL: sync:///Note/SyncNotes/Note/6713

Title: Incorrect behavior

CCList: karen

Body: *** Original text on Feb 23 2001,
13:42:36 (GMT 5:00) by  karen ***

The logic is incorrect in the fork
statement.

newCCList: asicdev karen

newTitle: x.v Note: Incorrect behavior
```

The CCList and Title properties are updated in Note 6713, as well.

Updating Notes Based on Particular Criteria

You can use the `note query` command to collect notes matching particular criteria, then operate on the generated list of notes using commands such as `note setprops`, `url setprop`, `note detach`, and `note delete`.

The `delNotes.tcl` script (in the `<SYNC_DIR>/share/examples/doc/stclguide` directory) uses this method to delete all notes attached to the `sync:///Projects/Asic` project:

delNotes.tcl Script

```
puts <PRE>
puts "sync:///Projects/Asic notes: <BR>"
puts "[note links -object \
  sync:///Projects/Asic]<BR>"                    puts
"sync:///Projects/CPU notes: <BR>"
puts "[note links -object \
  sync:///Projects/Asic]<BR>"
```

Print out the notes attached to the Asic and CPU projects.

```
puts "Notes to be deleted: <BR>"
foreach note [note query -attached \
  sync:///Projects/Asic] {
  puts "$note<BR>"
  note delete $note
}
puts "Asic notes deleted. <BR>"
```

Use the 'note query' command to obtain the notes attached to the Asic project. Delete those notes.

```
puts "sync:///Projects/Asic notes: <BR>"
puts "[note links -object \
  sync:///Projects/Asic]<BR>"
puts "sync:///Projects/CPU notes: <BR>"
puts "[note links -object \
  sync:///Projects/Asic]<BR>"
puts </PRE>
```

Again, print out the notes attached to the Asic and CPU projects.

Sample Results from the delNotes.tcl Script

```
sync:///Projects/Asic notes:

sync:///Note/SyncNotes/Note/3
sync:///Note/SyncNotes/Note/6697

sync:///Projects/CPU notes:

sync:///Note/SyncNotes/Note/3
sync:///Note/SyncNotes/Note/6697
sync:///Note/SyncNotes/Note/6677
```

Print the notes attached to the Asic and CPU projects.

```
Notes to be deleted:

sync:///Note/SyncNotes/Note/3
sync:///Note/SyncNotes/Note/6697

Asic notes deleted.
```

Deleting all notes attached to the Asic project

```
sync:///Projects/Asic notes:

sync:///Projects/CPU notes:

sync:///Note/SyncNotes/Note/6677
```

Notice that the deleted notes no longer appear in the CPU note links list, as well as the Asic note links list.

Related Topics

Accessing Notes

Creating and Attaching Notes

Working with Note Types

The stcl Environment for Client Scripts

Working with Client stcl Scripts

Once you decide that the script you are developing is appropriate as a client-side script as opposed to a server-side script (see Client-Side Versus Server-Side stcl for comparisons), you can decide how to set up your stcl scripts for use by a site, project team, or individual user. You must decide whether to source the script from a startup script or to include it as an **autoloaded** procedure. DesignSync supports an autoloading mechanism for site and project installations by which the stcl procedure is loaded only when a DesignSync user invokes the procedure. You can also decide to set up a trigger to run the stcl script or procedure.

Here are the basic steps for developing client stcl scripts:

Develop the code for your stcl script.

As you develop your stcl scripts, you might need to access DesignSync Object Model (SOM) data or client environment information from within your stcl script. The following sections will help you develop your stcl scripts:

- Introduction to the DesignSync Object Model
- Accessing Environment Information from Client Scripts
- stcl Scripting Tips
- command defaults command line topic

Test your stcl script.

During development of your script, you can test your script using the Tcl `source` command or the DesignSync `run` command with the pathname of the script as an argument.

Set up your script.

Once you have tested your script, however, it is best to include it in the user, site, or project environment using a startup script or the stcl autoloading mechanism.

The following table delineates the methods of setting up client scripts.

If you want ...	**Set up your client script using**
All users at your site, server, or project to have access to the script without performing set-up steps	Autoloaded stcl procedures

To run an stcl script only when an event occurs, for example, when a revision control operation occurs on an object of a particular project.	Autoloaded stcl procedures and client triggers
To create stcl scripts for your individual use without having to create Tcl procedures	A user startup script
To run a script whenever any users at your site start DesignSync	A site startup script (although you might find it more efficient to implement stcl scripts as autoloaded procedures instead)
To run an stcl script as an executable OS shell script	Create a shell script that runs the stcl shell using '`exec`'
To run an stcl script from dss or dssc	Create an alias for the stcl script

Run the script.

Once you have developed your stcl script and stored it in the desired directory, it is either automatically fired by a trigger, sourced upon startup, or, if it is an autoloaded procedure, it is automatically loaded when you or a team member invokes it. For details about running scripts from clients, see How to Run stcl Scripts from Clients.

Related Topics

Client-Side Versus Server-Side stcl

How to Run stcl Scripts from Clients

stcl Scripting Tips

Setting Up stcl Client Scripts

Accessing Environment Information from Client Scripts

To access environment information from client or server scripts, use the `syncinfo` command instead of using the Tcl global array, `env`. For example, instead of using `$env(SYNC_DIR)` to access the `SYNC_DIR` directory, you use the DesignSync command, '`syncinfo syncDir`', the `syncinfo` command with the `syncDir` argument. The reason to avoid using the global array, `env`, is that DesignSync does not always obtain its values from environment variable settings; instead, DesignSync obtains some values from registry settings. Because you cannot be sure how DesignSync obtains particular values, the safest means of getting these values is by using the `syncinfo` command.

The `syncinfo` command is a client- and server-side command; however, some arguments are supported for client scripts and others are supported for server scripts. The following table shows the arguments supported for client scripts. For usage details, see the `syncinfo` command.

syncinfo Arguments Available from Clients

General Information	
helpFileDir	Returns the directory that contains the help (documentation) files.
isServer	Returns a Tcl boolean value (0 or 1) indicating whether the software executing the `syncinfo` command is acting as a server (1) or client (0).
syncDir	Returns the root directory of the SyncServer installation.
version	Returns the version of the DesignSync software.
Registry Information	
clientRegistryFiles	Returns a comma-separated list of registry files used by the DesignSync clients.
portRegistryFile	Returns the port-specific registry file.
projectRegistryFile	Returns the project-specific registry file.
serverRegistryFiles	Returns a comma-separated list of registry files used by the SyncServer.
siteRegistryFile	Returns the site-specific registry file.
enterpriseRegistryFile	Returns the enterprise-specific registry file.
syncRegistryFile	Returns the DesignSync-supplied standard registry file.
userRegistryFile	Returns the user-specific registry file.
usingSyncRegistry	Returns a Tcl boolean value (0 or 1) indicating whether the DesignSync software is using the text-based registry (1) or the native Windows registry (0).
Customization Information	
customDir	Returns the root directory of the "custom" branch of the SyncServer installation hierarchy, which contains all site- and server-specific customization files.
customEntDir	Returns the directory that contains the enterprise-specific

	customization files.
customSiteDir	Returns the directory that contains site-specific customization files.
siteConfigDir	Returns the directory that contains site-specific configuration files.
userConfigDir	Returns the directory that contains user configuration files.
userConfigFile	Returns the user configuration file.
Client Information	
connectTimeout	Returns the number of seconds the client will wait per communication attempt with the server.
commAttempts	Returns the number of times client/server communication is attempted before failing.
defaultCache	Returns the default cache directory for the client as specified during installation or using SyncAdmin.
fileEditor	Returns the default file editor as specified during installation or using SyncAdmin.
htmlBrowser	(UNIX only) Returns the default HTML browser as specified during installation or using SyncAdmin.
proxyNamePort	Returns the <name>:<port> of a proxy, if one is defined in a client registry file or using the ProxyNamePort environment variable.
somTimeout	Returns the number of milliseconds after an unsuccessful server connection attempt during which the client does not try to connect again.
User Information	
home	Returns the home directory of the user running `syncinfo` (HOME on UNIX, or as defined in your user profile on Windows platforms).
userName	Returns the account name of the user running `syncinfo`.

Related Topics

Autoloaded Site and Project stcl Procedures

Client Triggers

Client-Side Versus Server-Side stcl

How to Run stcl Scripts from Clients

Startup Scripts

Working with Client stcl Scripts

Startup Scripts

As you are developing your script, you can use the stcl source command or the dss run command as described in How to Run stcl Scripts from Clients. Once you have debugged your script however, you can use a startup script to make your stcl scripts accessible from your DesignSync clients. This type of startup script is most suitable for use by a single user. For your site or project team, it's more efficient to set up stcl procedures as autoload procedures. In this way, DesignSync only loads procedures when they are invoked. See Autoloaded Site and Project stcl Procedures to set up stcl procedures for a site or project team.

Startup scripts can contain general environment variables or aliases. You can also include a Tcl source statement in a user startup script to:

- Read in Tcl procedure declarations so that you can invoke the procedures as commands
- Read in Tcl scripts that you want to run upon startup of the client, such as using the scd command to change to a particular project directory

Setting Up stcl Procedures Using Startup Scripts

To set up stcl procedures as commands that are readily available, follow these steps. **Note**: These steps assume a UNIX shell, but you can create a Windows startup script as well.

1. Develop stcl procedures and include these in a file with a .tcl extension, for example, `tcl/mytclprocs.tcl`.

2. Create a startup script that sources the stcl script you have developed. For example, create a startup script named `.startup.tcl` and include the following `source` command:

```
source tcl/mytclprocs.tcl
```

3. Specify the startup script using the **Startup** tab of the **DesSync Tools=>Options** dialog box.

   If you do not specify a path for the script, DesignSync searches for the script in these directories in the following order:

   `$SYNC_USER_CFGDIR` (resolves to `<HOME>/.synchronicity` by default)

   `$SYNC_SITE_CUSTOM (resolves to <SYNC_CUSTOM_DIR>/site by default)`

`$SYNC_ENT_CUSTOM` (resolves to `<SYNC_CUSTOM_DIR>/enterprise` by default)

If the startup script that you specify has a `.tcl` extension, DesignSync automatically interprets it as an stcl script. See DesignSync Help for details.

Related Topics

Accessing Environment Information from Client Scripts

Autoloaded Site and Project stcl Procedures

Client Triggers

Client-Side Versus Server-Side stcl

How to Run stcl Scripts from Clients

Working with Client stcl Scripts

Autoloaded Site and Project stcl Procedures

Autoloaded procedures are stcl procedures the DesignSync clients load on demand, when a user invokes them. As a system administrator or project leader, you can set up stcl procedures your site or project team can access. Team members can then use the procedures within their DesignSync sessions without having to perform any set-up steps. You can set up stcl procedures for your whole site or for a single project. After you have set up autoloaded stcl procedures, users can call the procedures directly or you can create client triggers to execute the stcl procedures based on criteria you define using **SyncAdmin**, the ENOVIA Synchronicity Administrator tool. See Client Triggers for a discussion about setting up client triggers.

As system administrator or project leader, you place the startup files and scripts in designated site or project directories. DesignSync then loads an stcl procedure only when a team member invokes it.

The autoload mechanism works with an index file, `tclIndex`, which indexes the stcl procedures. After you place the startup files and scripts in their designated directories, DesignSync generates the index file automatically upon invocation. You can also force the index file to be generated using the `auto_mkindex` procedure.

Follow these steps to set up the stcl autoload capability:

1. To set up aliases and environment variables, set up autoload.tcl startup scripts.
2. To store the procedures, set up stcl procedure files.
3. To register the stcl procedures, create the tclIndex file.

4. Invoke the stcl procedures.

Set Up autoload.tcl Startup Scripts

To set up aliases and environment variables for use with your stcl procedures, store them in a startup script named `autoload.tcl` in the site-wide or project-level tcl directory listed below. DesignSync sources the `autoload.tcl` scripts in the order shown below.

- ENOVIA Synchronicity-provided startup script:

  `<SYNC_DIR>/share/client/tcl/autoload.tcl`

- Enterprise-wide startup script:

  `<SYNC_ENT_CUSTOM>/share/client/tcl/autoload.tcl`

- Site-wide startup script:

  `<SYNC_SITE_CUSTOM>/share/client/tcl/autoload.tcl`

- Project-level startup script:

  `<SYNC_PROJECT_CFGDIR>/tcl/autoload.tcl`

`<SYNC_PROJECT_CFGDIR>` has no default; no project information is loaded if this environment variable is not set. `<SYNC_SITE_CUSTOM>` resolves to `<SYNC_CUSTOM_DIR>/site`. **<**`SYNC_SITE_CUSTOM>` is equivalent to `<SYNC_CUSTOM_DIR>/site`; if **<**`SYNC_SITE_CUSTOM>` is not set, but **<**`SYNC_CUSTOM_DIR>` is set, DesignSync can still access the site-wide `autoload.tcl` startup script.

Note: Do not put custom stcl files in the `<SYNC_DIR>/share/client/tcl` or the `<SYNC_ENT_CUSTOM>/share/client/tcl` directories; custom scripts belong in the site-wide and project-level directories specified above.

The startup mechanism described here supports site-wide and project-level configurations. To learn about creating user-level startup scripts, see the DesignSync Data Manager User's Guide: Running a Script at Startup topic. Aliases and variables defined in the user-level startup scripts have precedence over those defined in the project-level and site-wide startup scripts.

Set Up stcl Procedure Files

You can create any number of stcl procedure files denoted by a `.tcl` extension. Each file can contain multiple stcl procedures. For your stcl procedures to be autoloaded, you must store the procedure files in one of the site-wide or project-level tcl directories:

- For site-wide stcl files:

  `<SYNC_SITE_CUSTOM>/share/client/tcl`

- For project-level stcl files:

  `<SYNC_PROJECT_CFGDIR>/tcl`

`<SYNC_PROJECT_CFGDIR>` has no default; no project information is loaded if this environment variable is not set. <`SYNC_SITE_CUSTOM`> resolves to `<SYNC_CUSTOM_DIR>/site`.

Note: <`SYNC_SITE_CUSTOM`> is equivalent to `<SYNC_CUSTOM_DIR>/site`; if <`SYNC_SITE_CUSTOM`> is not set, but <`SYNC_CUSTOM_DIR`> is set, DesignSync will still access the site-wide stcl files.

Create the `tclIndex` File

The `tclIndex` file generates automatically when a DesignSync client is invoked. You must have write access to the tcl directory for the file to be generated.

You can also force the generation of the `tclIndex` file using the `auto_mkindex` procedure. Using the `auto_mkindex` procedure to manually generate the index file is helpful when you are testing your Tcl procedures. To use `auto_mkindex`, set your DesignSync mode to stcl and provide the absolute directory path of the site-wide or project-level tcl directory:

```
stcl> auto_mkindex
"<INSTALL_DIR>\custom\\site\\share\\client\\tcl"
```

Note: This example shows the syntax of a Windows client where an extra backslash is required as an escape character in a pathname.

For more information, see the `auto_mkindex` command.

The newly indexed procedures are not visible to currently running DesignSync clients. If a DesignSync client is already running when the index is generated, use the `auto_reset` command to have DesignSync reread the index file:

```
stcl> auto_reset
```

For more information, see the `auto_reset` command.

Invoke the stcl Procedures

When a team member invokes an stcl procedure, DesignSync searches the DesignSync, enterprise, site, and project Tcl index files in this order:

- For DesignSync-provided stcl files:

  `<SYNC_DIR>/share/client/tcl/tclIndex`

- For enterprise-wide stcl files:

  `<SYNC_ENT_CUSTOM>/share/client/tcl/tclIndex`

- For site-wide stcl files:

  `<SYNC_SITE_CUSTOM>/share/client/tcl/tclIndex`

- For project-level stcl files:

  `<SYNC_PROJECT_CFGDIR>/tcl/tclIndex`

Thus, if an stcl procedure is overloaded, existing in both the project-level and site-wide directories for example, the project-level version of the procedure is autoloaded when a team member calls the procedure.

Note: Do not put custom stcl files in the `<SYNC_DIR>/share/client/tcl` or the `<SYNC_ENT_CUSTOM>/share/client/tcl` directories; custom scripts belong in the site-wide and project-level directories specified above.

You can use the `parray` command with `auto_index` as its argument to list the procedures available:

```
stcl> parray auto_index
```

For more information, see the `parray auto_index` command.

Client Triggers

You can create client triggers, watchpoints that cause your stcl script or procedure to fire in response to an action. Client triggers are documented fully in the DesignSync Data Manager Administrator's Guide: Triggers Overview. This topic highlights the ways you can set up your stcl scripts with client triggers.

You can set up client triggers using the DesignSync `trigger create` command or using the SyncAdmin Client Triggers tab. You can set up client triggers so that they fire

for an individual user, for all users at a site, or for all users on a project team. To set up stcl scripts with client triggers, you can choose to:

- Load a set of commands that get executed when the trigger fires.

  If you choose this option, you must reload the entire set of commands if you wish to make a change. This choice maps to the **Tcl Commands** action type in the SyncAdmin Client Triggers tab, as well as the DesignSync `trigger create -tcl_script` option. **Note**: You must be in an stclc or stcl shell to use this option.

- Load an stcl script that gets executed when the trigger fires.

  If you choose this option, you must recreate the trigger and reload the stcl script if you wish to make a change. This choice maps to the **Tcl Stored Procedure** action type in the SyncAdmin Client Triggers tab, as well as the DesignSync `trigger create -tcl_store` option.

- Provide the path to an stcl script so that the script is reloaded each time the trigger fires.

  Use this method so that you can continue editing the stcl script without having to recreate the trigger. This is a good method to use while you are developing your stcl script and the trigger to fire it. This choice maps to the **Tcl File** action type in the SyncAdmin Client Triggers tab, as well as the DesignSync `trigger create -tcl_file` option.

- Invoke an autoloaded stcl procedure each time the trigger fires.

  This is the most effective method for setting up triggers for your site or project team. First, you develop your stcl script and store it in the appropriate location so that it is automatically autoloaded, then you create a trigger that invokes the command. This choice also maps to the **Tcl Commands** action type in the SyncAdmin Client Triggers tab, as well as the DesignSync `trigger create -tcl_script` option. **Note**: You must be in an stclc or stcl shell to use this option. See Autoloaded Site and Project stcl Procedures to learn how to set up for autoloading.

Running stcl Scripts from Clients

How to Run stcl Scripts from Clients

There are many different DesignSync clients from which you can run stcl scripts. See DesignSync Data Manager User's Guide:Comparing the DesignSync Shells for details about these shells. In general, the clients are divided into dss clients and stcl clients.

Use the stcl shell when you need the scripting constructs of Tcl, such as conditionals (`if`/`then`/`else`), loops (`while`, `for`, `foreach`), and variable assignment (`set`). If you do not need Tcl constructs, dss provides a simpler command environment.

Note: You can run **server** scripts from DesignSync clients using the `rstcl` "remote stcl" command. See How to Run stcl Server Scripts or the `rstcl` command for more information.

The following sections describe the clients listed below and illustrate how to invoke scripts within these clients. These sections provide steps to invoke sample stcl scripts located in the <`SYNC_DIR`>`/share/examples/doc/stclguide` directory.

- stclc and stcl Clients
- dssc and dss Clients
- The DesSync Client (supports both the dssc and the stclc shells)
- OS Shell Scripts

dssc and dss Clients

The dssc and dss (DesignSync shell) clients are the command-line interfaces from which you can invoke revision control commands. You can execute dss commands directly from your terminal window or you can integrate commands into makefiles or OS shell scripts. By using dss instead of stcl, you can perform basic DesignSync operations without having to understand the details of Tcl syntax.

The dssc and dss clients support all DesignSync commands, but not Tcl commands directly. However, you can run client, as well as server, stcl scripts in the dssc and dss shells. Unlike dss, dssc does not use a DesignSync daemon process (syncd). Because syncd handles requests serially, using dssc eliminates a potential bottleneck when you have multiple shells communicating with a SyncServer. To decide whether to use a dss or concurrent dssc shell, see DesignSync Data Manager User's Guide:Comparing the DesignSync Shells.

To run stcl client scripts from the dssc and dss clients, you use the dss `run` command. To run a server script on a dssc/dss client, you use the `rstcl` command; see the `rstcl` command to run a server script remotely on your client.

Invoking the dssc and dss Shells

- To enter the dssc environment, you enter the `dssc` command at the OS prompt. To enter the dssc Windows environment, select **DesignSync Shell** from the **Dassault Systems DesignSync** program group on the Windows Start menu.
- To enter the dss environment, enter the `dss` command at the OS prompt.

You remain in the dssc or dss shell until you enter the `exit` command, which returns you to your OS shell. To learn about the types of editing supported in the dssc and dss shells, see DesignSync Data Manager User's Guide:Command-Line Editing.

Using the dss `run` command

The `run` command does not let you pass arguments to the script being invoked. Use the `run` command if you have an stcl script that performs its action immediately rather than defining procedures and variables to be called later. If you want to access a procedure or variable defined in an stcl script from a dss shell, use the alias command as described in Using the alias Command to Access stcl Procedures below.

You can use the dss run command from a dss or dssc shell as follows:

```
dss> run $SYNC_DIR/share/examples/doc/stclguide/unlockall.tcl
```

If the script name has a `.tcl` extension, the `run` command runs the script using the stcl interpreter. You can also invoke the dss `run` command directly from your OS shell also, in which case you are returned to the OS shell:

```
% dss run $SYNC_DIR/share/examples/doc/stclguide/unlockall.tcl

Beginning Unlock operation...

Unlocking: stack_pointer/inc_dec_sp.gv : Not locked

Unlocking: stack_pointer/reg6.v : Unlocked.

Unlocking: top.f : Not locked

Unlocking: top.v : Not locked

Unlock operation finished.

%
```

The only difference between using the dss `run` command and the stcl `source` command is that the `run` command does not pass the script's return value.

Using the `alias` Command to Access stcl Procedures

You can run an stcl script from dss using the `run` command. However, once the script has finished running, none of the script's environment is available to you.

You can make Tcl procedures defined in a script accessible from dss by using the DesignSync `alias` command. For example, the `syncGo.tcl` script defines a procedure `syncGo` that is made available as a dss command named `go`:

```
proc syncGo {dir} {

  scd $dir

  ls

}

alias -args 1 -- go syncGo \$1
```

Then from dss:

```
dss> run $SYNC_DIR/share/examples/doc/stclguide/syncGo.tcl

Alias go created.

dss> go /Sportster

Directory of: file:///home/karen/Sportster


Time Stamp Status Version Locked By Name

---------- ------ ------- --------- ----

07/14/2000 14:40 - code

07/14/2000 14:40 - synth

07/14/2000 14:41 - test

07/14/2000 14:42 - top
```

This script has one required argument, the directory where you want to go, and performs an `scd` (to change directories within the DesignSync environment only) and an `ls`. The `alias` command then defines a DesignSync command, `go`, that executes the `syncGo` procedure.

Related Topics

How to Run stcl Scripts from Clients

stclc and stcl Clients

The stclc and stcl shells are the DesignSync clients that support all DesignSync commands and commercial Tcl commands. You can run client, as well as server, stcl scripts in the stclc and stcl shells. Unlike stcl, stclc does not use syncd. Because syncd handles requests serially, using stclc eliminates a potential bottleneck when you have multiple shells communicating with a SyncServer. The stclc also provides more command-line editing capabilities than the stcl shell (see DesignSync Data Manager User's Guide: Command-Line Editing). To decide whether to use an stcl or concurrent stclc shell, see DesignSync Data Manager User's Guide:Comparing the DesignSync Shells.

Invoking the stclc and stcl Shells

- To enter the stclc environment, you enter the `stclc` command at the OS prompt. To enter the stclc Windows environment, select **DesignSync Tcl Shell** from the **Dassault Systems DesignSync** program group on the Windows Start menu.
- To enter the stcl environment, enter the `stcl` command at the OS prompt.

You remain in the stclc or stcl shell until you enter the `exit` command, which returns you to your OS shell. To learn about the types of editing supported in the stclc and stcl shells, see DesignSync Data Manager User's Guide:Command-Line Editing.

Running stcl Scripts

To run stcl client scripts from the stclc and stcl clients, you can use these methods:

- The Tcl source command
- Startup scripts
- Tcl procedure autoloading
- Client triggers

To run a server script on a stclc/stcl client, you use the `rstcl` command; see the `rstcl` command to run a server script remotely on your client.

Using the Tcl `source` Command

The Tcl `source` command executes a Tcl script. The default return value is the value of the last command executed in the script. You can also return a value explicitly using the

Tcl `return` command. The `source` command does not accept additional parameters; if you want to specify additional parameters to a Tcl command you are creating, you define a Tcl procedure using the `proc` command. When you run an stcl script (using either the `source` command or the DesignSync `run` command), the variables and commands defined in the script are now available for use within your stcl session.

Because the `source` command is a Tcl command, you cannot use it in dss mode. However, DesignSync provides a `run` command that you can use to execute an stcl or dss script from the dssc and dss clients. (See dssc and dss Clients.)

To source an stcl script within an stclc or stcl shell:

You use the stcl `source` command from an stclc or stcl shell as follows:

```
stcl> source
$SYNC_DIR/share/examples/doc/stclguide/syncIsLocked.tcl
```

Now that you have sourced the `syncIsLocked.tcl` stcl script, the variables and procedures defined in the script are available for use within your stcl session. For example, the `syncIsLocked.tcl` script defines a procedure called `syncIsLocked`, so you can run the `syncIsLocked` command after you've sourced the script:

```
stcl> syncIsLocked [pwd]

invalid command name "syncIsLocked"

stcl> source
$SYNC_DIR/share/examples/doc/stclguide/syncIsLocked.tcl

stcl> syncIsLocked [pwd]
```

This example checks the current directory and all subdirectories and reports the name and owner of any locked files. The script assumes that you are running in a directory associated with a DesignSync vault folder.

To source an stcl script from the OS:

You can also source an stcl script directly from your OS shell, in which case you are returned to the OS shell. The `stcl` command assumes its argument is a script so you do not need the source command:

```
% stcl $SYNC_DIR/share/examples/doc/stclguide/unlockall.tcl
```

Note: Because the `stcl` command assumes its argument is a script, you need to use the `-exp` argument to call other `stcl` commands from your OS shell. Delimit the

command using single quotes; you can also use double quotes, but if you use single quotes, you can still use double quotes within the command line. See the `stcl` command or `stclc` command for more information.

```
% stcl -exp 'source
$SYNC_DIR/share/examples/doc/stclguide/unlockall.tcl'
```

Because this method returns you directly to the OS, you cannot later access procedures and variables defined in the `.tcl` file. Use this method only for scripts that carry out their action immediately.

Using a Start-Up Script

If you or your DesignSync administrator has sourced an stcl script file from a startup file, you can type in the stcl procedure name in your OS shell or your stcl/stclc shell to run the stcl procedure. See Startup Scripts to configure your startup file.

Invoking an Autoloaded Procedure

Your DesignSync administrator or project leader can store stcl scripts in a directory from which they are automatically loaded (autoloaded) when you invoke the script. The Autoloaded stcl Procedures topic describes how autoloaded procedures are set up.

If your DesignSync administrator or project leader has set up autoloaded stcl procedures, you can type in the name of the stcl procedure in the stclc or stcl shells, or in the DesSync GUI output window if **Tcl Mode** has been selected from the **Options** pulldown.

To list the available autoloaded stcl procedures, use the `parray` command with `auto_index` as its argument:

```
stcl> parray auto_index
```

For more information, see the `parray auto_index` command.

Using Client Triggers

You or your DesignSync administrator can set up client triggers to automatically run scripts based on some event. You can use also force a trigger that has already been set up to fire by using the `trigger fire` command. This command is useful to use to test new triggers you are developing. See Client Triggers for more information, as well as the DesignSync Data Manager User's Guide:Triggers Overview.

Related Topics

The DesSync Client

DesSync is the DesignSync graphical user interface (GUI), which provides users with a simple interface to the revision-control commands. The menus and dialog boxes make it easy to choose from these commands and their associated options. DesSync also provides access to the dss and stcl command-line interfaces through the command window at the bottom of the DesSync window. Having the command window within the GUI both provides a means for users to enter commands or options that aren't available in the GUI and makes the transition from GUI to command line (for those who prefer command-line environments) easier.

To Run an stcl Script from the DesSync Client:

1. On the right side of the DesSync output window, click the **Options** pulldown and select **Tcl Mode**.
2. At the prompt, type the following:

   ```
   source
   $SYNC_DIR/share/examples/doc/stclguide/syncIsLocked.tcl
   ```

The variables and commands defined in the script are now available for use within your stcl session.

OS Shell Scripts

You can also run an stcl script as an executable OS shell script using one of the following methods:

Include the stcl shell pathname as the shell script's first line:

In general, you can create shell scripts that use different shells than the one you are running in. To specify the shell that a script should run under, the first line of the script file should look like:

```
#![full path to shell executable]
```

For example, a common shell executable path is `/bin/sh`. You can even specify options with which to invoke the shell. So, in order to specify stclc as the command shell for a particular script, you might use something like:

```
#!/net/mymachine/usr1/syncinc/bin/stclc
```

There are some complications to this method however. The total length of the line used to specify the path to the command shell, including any arguments, cannot exceed 30

characters for some operating systems. On these operating systems, you can use links to specify the executable.

Use the exec command to call stcl from an OS shell script:

To avoid the issue of long pathnames to the stclc shell, you can use a trick posted to the `comp.lang.tcl` newsgroup. Because the syntax for comments is different in the Bourne shell versus the stcl/stclc shell, the following sh script can call stclc on itself:

```
#!/bin/sh
# the next line restarts using stcl \
exec stclc $0 ${1+"$@"}
# from this line below is the tcl script
# for eg
puts "hello world"
```

The `/bin/sh` interpreter runs this script and calls `exec stclc`. The `$0` option calls the script on itself and the `${1+"$@"}` passes the rest of the arguments to stclc. The trick is that in tcl and stcl shells, the \ character continues the comment line to the next line; hence, the stclc interpreter skips over the `exec stclc` command line. The stclc interpreter then continues interpreting the rest of the noncomment lines of the script.

Related Topics

dssc and dss Clients

How to Run stcl Scripts from Clients

stclc and stcl Clients

The DesSync Client

The stcl Environment for Server Scripts

Working with Server stcl Scripts

Once you decide that the script you are developing is appropriate as a server-side script as opposed to a client-side script, set up your stcl script on your server so that users of the server can run the script.

Server stcl scripts run on SyncServers and are invoked using either a URL script request on a web browser, the `rstcl` command on a client, or a server-side trigger. These invocation methods are explained in How to Run stcl Scripts from Servers.

DesignSync features security safeguards to limit stcl script access on SyncServers. Users cannot install scripts on a SyncServer; however, they can run scripts on the SyncServer once the scripts have been installed. The owner of the server installation, typically the DesignSync administrator, has sole permission to the locations where server-side stcl scripts are stored. See Developing Server-Side stcl Scripts and Server Scripts and SyncServer Security for more information.

Related Topics

Client-Side Versus Server-Side stcl

Developing Server-Side stcl Scripts

How to Run stcl Server Scripts

stcl Scripting Tips

Setting Up stcl Server Scripts

Developing Server-Side stcl Scripts

To set up server-side stcl scripts, you must be a DesignSync tools administrator. If you are not the tools administrator, work through your tools administrator to set up stcl server-side scripts. In some cases, project leaders own and manage SyncServers for their projects and can set up their own server-side stcl scripts. For more information, see Server Scripts and SyncServer Security.

Setting Up a Server-Side Script

The following procedure shows how you set up a server-side script. You can try out the steps by setting up a sample server-side stcl script from the `<SYNC_DIR>/share/examples/doc/stclguide` directory. The following table lists some server-side scripts for you to try out.

Server-Side Sample Scripts

`chEmailAddr.tcl`	Use the DesignSync web interface **Admin Menu**=>**User Profiles**=>**Add** command to add new users to a SyncServer. Then, you can use the `chEmailAddr.tcl` script to change all email addresses matching the parameter you pass to the script. See Passing Parameters in Server-Side stcl Scripts below for details. Make sure you have permission to edit users by checking the ProjectSync `EditUser` access control in your `AccessControl` files: `<SYNC_DIR>/share/AccessControl`, `<SYNC_ENT_CUSTOM>/share/AccessControl`, `<SYNC_SITE_CUSTOM>/share/AccessControl`, or `<SYNC_CUSTOM_DIR>/servers/<host>/<port>/share/AccessControl`
`lsUserProps.tcl`	Lists the ProjectSync user profile properties.
`lsProjProps.tcl`	Lists the ProjectSync project properties if you have set up a project. To set up a project, use the DesignSync Web interface **ProjectSync Menu**=>**ProjectSync Projects**=>**Create** command.

To Set Up a Server-Side stcl Script:

1. Write your stcl script.

   The following sections provide some pointers for developing server-side stcl scripts. See also stcl Scripting Tips.

2. Store your stcl script in one of the `tcl` directories in your DesignSync installation.

   You can set up the stcl script so that all users at your site or all users of a SyncServer can access the script. See Where to Place Server Scripts for details.

3. Test the script and then inform your users of the script's availability.

   See How to Run stcl Scripts from Servers for the various methods of invoking server-side scripts.

Important: If you make modifications to the script, use the ProjectSync Reset Server button to force the SyncServer to reread your script.

Where to Place Server Scripts

There are four `tcl` directories the SyncServer searches when a user invokes a server-side stcl script:

- Server-specific script directory (UNIX servers only):

  `<SYNC_CUSTOM_DIR>/servers/<host>/<port>/share/tcl`

- Site script directory:

  `<SYNC_SITE_CUSTOM>/share/tcl`

  Note: By default, `$SYNC_SITE_CUSTOM` resolves to `$SYNC_CUSTOM_DIR/site`.

- Enterprise script directory:

  `<SYNC_ENT_CUSTOM>/share/tcl`

- Default script directory:

  `<SYNC_DIR>/share/tcl`

You set up server-side stcl scripts by storing the scripts in either the server-specific or site `tcl` directory on your SyncServer. To learn more about the directory structure of SyncServers, see the DesignSync Data Manager Administrator's Guide.

When a user runs a script either by specifying a URL script request or by calling the `rstcl` command, the SyncServer searches for the script in the order shown above. For example, if you have a script of the same name in both the server-specific script directory and the site script directory, the server-specific script has precedence. For more information on running server scripts, see How to Run stcl Scripts from Servers.

IMPORTANT: Do not put custom scripts in the default DesignSync script directory (`<SYNC_DIR>/share/tcl`); this directory is reserved for scripts and samples included with the product and might be overwritten when you upgrade your DesignSync software. Also, do not put custom stcl files in the `<SYNC_ENT_CUSTOM>/share/client/tcl` directory; the enterprise directory is intended for enterprise-wide customizations and might also be overwritten.

Passing Parameters in Server-Side stcl Scripts

To Pass Parameters to a Script:

Users pass parameters into a server-side stcl script using one of the following methods:

- In URL script requests, users specify the parameters as part of the URL.

  The parameter list takes the form of name/value pairs following the file parameter with the following syntax:

  ```
  <param1>=<value1>&<param2>=<value2>...
  ```

  You can pass any number of parameters using this method. Separate each parameter name/value pair with an ampersand (&). Separate each name and value with an equal sign (=). The parameter names and values cannot contain spaces; instead, represent spaces with a plus sign (+) or enclose the entire parameter list with quotes ("). The following is an example of a URL script request containing two parameters, `oldemail` and `newemail`:

  ```
  http://<machine>:port/scripts/isynch.dll?panel=TclScript&fi
  le=chEmailAddr.tcl&oldemail=mycompany.co.uk&newemail=mycomp
  any.com
  ```

- In rstcl commands, users specify the parameters using the -`urlparams` argument.

  The parameter list takes the form of name/value pairs with the same syntax as the parameter list in a URL script request:

  ```
  rstcl -server sync://<machine>:<port> -script
  chEmailAddr.tcl -urlparams
  oldemail=mycompany.co.uk&newemail=mycompany.com
  ```

To Access Parameters within the Script:

Within the script, you access the parameter values using the `SYNC_Parm` Tcl array. The names (keys) in this array are the names of the parameters. To access the value of a parameter, use the following syntax:

```
$SYNC_Parm(param1)
```

The following sample code from the `chEmailAddr.tcl` script in `<SYNC_DIR>/share/examples/doc/stclguide` references the parameters in the URL script request and the `rstcl` call above:

```
if {[string match "*$ SYNC_Parm(oldemail)" $ props(EmailAddr)]}
{
```

```
  regsub "@$ SYNC_Parm(oldemail)" $ props(EmailAddr) \
          "@$ SYNC_Parm(newemail)" changedemail
  url setprop $user EmailAddr $ changedemail
...
```

This sample code checks whether the `oldemail` parameter matches the `EmailAddr` property of the ProjectSync user profile. If there's a match, the code substitutes the value of the `newemail` parameter into the `EmailAddr` property of the user profile.

Commands Supported for Server-Side stcl Scripts

As you write your server-side stcl scripts, you will need to know the types of commands supported for server scripts:

- The DesignSync Object Model (SOM) provides commands for directly accessing the databases supporting the DesignSync products. See Introduction to the DesignSync Object Model for an overview.
- For information about the commands, see the ENOVIA Synchronicity Command Reference.
- To access server environment information from within your stcl script, see Accessing Environment Information from Server Scripts.

Using Procedures in Server-Side stcl Scripts

Unlike the stcl autoloading mechanism for calling client-side Tcl and stcl procedures, you cannot explicitly call a Tcl procedure on the server-side. You must call Tcl procedures from within a script on the server-side. To call a procedure on the server-side, include the procedure and any auxiliary procedures in a `.tcl` script file, then as the last statement in the `.tcl` file, call the procedure as in the following example:

```
proc showParams {} {
  global SYNC_Parm
  foreach param [array names SYNC_Parm] {
    puts "$param = $ SYNC_Parm($param)"
  }
}

showParams
```

Specifying the sync Protocol in Server-Side Scripts

When you specify `sync:` protocol URLs within scripts that run on a SyncServer, do not specify the host and port. For example, specify:

```
sync:///Projects/Asic
```

and not

```
sync://chopin:2647/Projects/Asic
```

Because the script is run on the server itself, `host:port` information is unnecessary and is stripped out by the server. Including the host and port can lead to incorrect behavior during object-name comparisons. Also, omitting the host and port makes your scripts more portable.

Note: Even if you are writing a script for a secure server, you must use the `sync:///` protocol rather than specifying the `syncs:` SSL protocol. If your DesignSync administrator has set up a secure SyncServer using access controls, the SyncServer automatically redirects communications through the SSL port. For more information about secure communications, see DesignSync Data Manager User's Guide:Overview of Secure Communications.

Error Messages and Server-Side Scripts

Error messages resulting from server-side stcl scripts display in the web browser where you've invoked the URL script request. If there are messages about access controls, the server writes these messages to the error_log file, `<SYNC_CUSTOM_DIR>/servers/<host>/<port>/logs/error_log`.

In addition to server error messages, you can provide explicit informational messages to users in your scripts. See Output Formatting for details.

Related Topics

Accessing Environment Information from Server Scripts

Client-Side Versus Server-Side stcl

How to Run stcl Server Scripts

Server Scripts and SyncServer Security

Working with Server stcl Scripts

Server Scripts and SyncServer Security

Keep in mind the following important server security recommendations as you develop your server-side stcl scripts.

SyncServer Permissions

Make sure your custom hierarchy has adequate permission protections so that only select individuals can place scripts in the DesignSync script directories. There is no explicit protection against stcl scripts; a script running on the server has complete and unrestricted access to the server's resources including the file system, even beyond the DesignSync software installation. A script running on the server has exactly the same rights as the system user account that is running the server. For example, if the server is installed and running as `syncmgr`, the script is allowed to do whatever the `syncmgr` user is allowed to do.

SyncServers do not run stcl scripts unless they are in one of the three supported `share/tcl` directories (described in Developing Server-Side stcl Scripts) or in the `share/panels` ProjectSync stcl script directory. (See ProjectSync User's Guide for details about using stcl scripts for panel customization.) The UNIX permissions for these directories must prevent users other than the designated DesignSync administrator or project leader from placing scripts in these script directories— even in the custom hierarchy.

The *ENOVIA Synchronicity DesignSync Administrator's Guide* contains server setup scenarios and information about using SUID to set up servers with effective permissions.

End User Input

To protect your server, make sure that any server-side Tcl scripts you install do not use strings entered by users as input to Tcl `exec`, `eval`, or `subst` commands. If input from end users is included in calls to these commands, a user can run arbitrary commands on the server.

Access Controls for Server Scripts

Verify access controls explicitly in server-side scripts. Access controls are generally ignored in server-side scripts; it is up to the script itself to call the `access verify` command for access controls it wishes to honor. This explicit call to access verify is not needed for most DesignSync commands, which honor their access controls even on the server. The call to access verify is needed for ProjectSync commands, such as `note` and `user`, which do not perform any access checks on their own.

Note: If you make modifications to an access control file or a server-side stcl script, use the ProjectSync Reset Server button to force the SyncServer to reread your file.

Example of access verify

The following sample script, `<SYNC_DIR>/share/examples/doc/stclguide/deleteUser.tcl`, deletes a user profile, but first verifies the DeleteUser access control to ensure that the user

running the script has permission to delete a user on the server. Notice that the script uses HTML formatting for its output; see Output Formatting for more information.

```
# deleteUser.tcl
#
# This script deletes the user, $SYNC_Parm(terminate_user).
# The script first calls access verify to ensure that the
# user who calls the script has permission to delete a
# user (the DeleteUser access control).
#
set isSelf [expr {$SYNC_User==$SYNC_Parm(terminate_user)}]
if {[access verify DeleteUser $SYNC_User
  $SYNC_Parm(terminate_user) $isSelf]} {
  user delete $SYNC_Parm(terminate_user)
} else {
  puts "<h1>Permission denied: you may not delete this \
  user</h1>"
}
```

Custom Access Controls for Server Scripts

In some cases, the DesignSync administrator must create access control definitions in addition to calling access verify from server-side scripts.

For example, to prevent ProjectSync users from deleting user profiles, the DesignSync administrator might include the following access control in the `$SYNC_SITE_CUSTOM/share/AccessControl` file:

```
  access allow DeleteUser only users $admin
```

Note: By default, `$SYNC_SITE_CUSTOM` resolves to `$SYNC_CUSTOM_DIR/site`.

If your server-side script operates on RevisionControl notes, you need to protect the integrity of your data by blocking access to the server while the script runs:

1. Edit your custom AccessControl file to deny all actions that operate on RevisionControl notes, for example:

   ```
   access deny Checkin everyone
   access deny Tag everyone
   ```

2. Reset access controls using the ProjectSync Access Reset menu item.
3. Run your server-side script.
4. Edit your custom AccessControl file and remove the `access deny` commands.
5. Reset access controls using the ProjectSync Access Reset menu item.

See the ENOVIA Synchronicity Access Control Guide for more information about customizing AccessControl files. See *DesignSync Data Manager Administrator's Guide*: RevisionControl Notes Overview to learn about RevisionControl notes, the notes that DesignSync creates when revision control operations occur.

Related Topics

Accessing Environment Information from Server Scripts

Client-Side Versus Server-Side stcl

Developing Server-Side stcl Scripts

How to Run stcl Server Scripts

Working with Server stcl Scripts

Accessing Environment Information from Server Scripts

To access environment information from server-side scripts, you can use environment variables as well as the `syncinfo` command. The following sections describe the types of information you can access with each of these methods.

Server-Side Environment Variables

For server-side scripts, DesignSync supports the following environment variables:

Variable	Description
`noteURL`	A namespace variable that determines the notes that are modified or attached in trigger scripts. (In earlier releases, this was a global varial called `SYNC_NoteURL`.)
`objURL`	A namespace variable that determines the objects to which notes are attached in trigger scripts. (In earlier releases, this was a global varial called `SYNC_NoteURL`.)
`oldProps`	A namespace variable that determines the previous value for any not properties that have been changed. This variable lists the name/value pairs for properties that change during an event. The `oldProps` varia is defined as a list, but is in array format. (In earlier releases, this was global variable called `SYNC_OldProps`).
`SYNC_ClientInfo`	A global Tcl array that is available only to server-side scripts that are from a browser. You cannot use `SYNC_ClientInfo` with client-side scripts, server-side trigger scripts, or server-side scripts run using the command. This array contains the following parameters:

`AgentName` - The name of the browser that sent the request, as repo
by the browser. This encodes information such as browser name, ver
and OS. The format differs depending on the browser.

`Locale` - Time zone information, passed in as the number of minutes
be subtracted from GMT. For example, five hours after GMT would be
represented as 300 (5 times 60).

`IPAddress` - The IP address of the client.

`UserName` - The name of the user. This information is also available
`$SYNC_User`. Use `$SYNC_User` if you only need user name informa

You access these parameters as follows:

```
$SYNC_ClientInfo(AgentName)
```

`SYNC_Parm` A global Tcl array used to determine parameters passed into server-s
Tcl scripts from the URL invoking the scripts. For example, if the URL

```
http://<host>:<port>/scripts/isynch.dll?panel=TclSc
&file=filename&name=Joe&age=30&weight=160
```

then in order to access the name, age, and weight, you would use
`$SYNC_Parm(name)`, `$SYNC_Parm(age)`, and `$SYNC_Parm(weig`

Note: The parameter name `command` is a special-purpose parameter
used to load special modes of a panel. Do not name your parameter
`command` unless you are using this advanced panel programming
technique.

`SYNC_User` A global variable that determines the ProjectSync user in all Tcl
applications.

`SYNC_ClearPort` A global variable that returns the cleartext port number.

`SYNC_Protocol` A global variable that returns the cleartext protocol -- the standard `ht`
protocol or the SSL encrypted `https` protocol.

`SYNC_ClientHttpHeaders` A global Tcl array that lets you access client HTTP header information
such as agent type, language setting, and the client's host. The head
information is passed as name and value pairs that you can access fr
server-side Tcl script as follows:

```
foreach {name val} [array get \
  SYNC_ClientHttpHeaders] {
  puts "$name=$val<br>"
}
```

Note: When you make changes to your registry settings, environment variables, or revision control note generation settings, you need to "`stop_sync_server`" and then "`start_sync_server`" instead of using the ProjectSync Reset Server button.

Accessing the Server Environment Using syncinfo

To access other environment information from client or server scripts, use the `syncinfo` command instead of using the Tcl global array, `env`. For example, instead of using `$env(SYNC_DIR)` to access the `SYNC_DIR` directory, you use the dss/stcl command, '`syncinfo syncDir`', the syncinfo command with the `syncDir` argument. The reason to avoid using the global array, `env`, is that DesignSync does not always obtain its values from environment variable settings; instead, DesignSync obtains some values from registry settings. Because you cannot be sure how DesignSync obtains particular values, the safest means of getting these values is by using the `syncinfo` command.

The `syncinfo` command is a client- and server-side command; however, some arguments are supported for client scripts and others are supported for server scripts. The following table shows the arguments supported for server scripts. For usage details, see the `syncinfo` command.

Arguments to syncinfo Available from Servers

General Information	
helpFileDir	Returns the directory that contains the help (documentation) files.
isServer	Returns a Tcl boolean value (0 or 1) indicating whether the software executing t command is acting as a server (1) or client (0).
syncDir	Returns the root directory of the SyncServer software installation.
version	Returns the version of the SyncServer software.
Registry Information	
clientRegistryFiles	Returns a comma-separated list of registry files used by DesignSync clients.
portRegistryFile	Returns the port-specific registry file.
projectRegistryFile	Returns the project-specific registry file.
serverRegistryFiles	Returns a comma-separated list of registry files used by a SyncServer.
siteRegistryFile	Returns the site-specific registry file.
enterpriseRegistryFile	Returns the enterprise-specific registry file.
syncRegistryFile	Returns the DesignSync standard registry file.
userRegistryFile	Returns the user-specific registry file.
usingSyncRegistry	Returns a Tcl boolean value (0 or 1) indicating whether DesignSync is using the registry (1) or the native Windows registry (0).
Customization Information	
customDir	Returns the root directory of the "custom" branch of the SyncServer installation which contains all site- and server-specific customization files.
customEntDir	Returns the directory that contains enterprise-specific customization files.

customSiteDir	Returns the directory that contains site-specific customization files.
siteConfigDir	Returns the directory that contains site-specific configuration files.
userConfigDir	Returns the directory that contains user configuration files.
userConfigFile	Returns the user configuration file.
Server Information	
serverMetadataDir	Returns the directory that contains the server metadata (such as relational datal
serverDataDir	Returns the directory that contains vault (repository) data that is stored by a ser
serverMachine	Returns the name of the server as returned by gethostname(). This value is retu when `syncinfo` is run from a server-side script.
serverName	Returns the name of the server as it was specified in the URL used to contact th This value is returned only when `syncinfo` is run from a server-side script.
serverPort	Returns the port number used by the server to respond to the `syncinfo` reques value is returned only when `syncinfo` is run from a server-side script.

Related Topics

Client-Side Versus Server-Side stcl

Developing Server-Side stcl Scripts

How to Run stcl Server Scripts

Server Scripts and SyncServer Security

Working with Server stcl Scripts

Running stcl Scripts from Servers

How to Run stcl Server Scripts

DesignSync leverages its client/server communication protocol to run stcl scripts on a SyncServer and return information back to your web browser or DesignSync client:

1. The web browser or DesignSync client sends a URL containing a request that a SyncServer run an stcl script.
2. The SyncServer interprets the URL, in this case, an instruction to load an stcl script.
3. The SyncServer finds the script in the site or server tcl directory and runs the script. See Developing Server-Side stcl Scripts for the location of these tcl directories.

Important: If you make modifications to the script, use the ProjectSync Reset Server button to force the SyncServer to reread your script. See ProjectSync User's Guide: Resetting the SyncServer for details.

There are a number of methods of sending a URL script request to the SyncServer:

- URL script requests -- Users enter a URL script request in a web browser. See URL stcl Script Requests.
- `rstcl` command -- Users apply the `rstcl` command in a DesignSync client to specify the stcl script and its arguments, then the DesignSync client sends the URL script request to the SyncServer. See the `rstcl` command.
- Server-side triggers -- stcl scripts run when a server-side trigger fires, sending the URL script request to the SyncServer. See Server Triggers.

To illustrate how to invoke client stcl scripts in these clients, the sections mentioned above provide steps to invoke the following sample stcl script: `<SYNC_DIR>/share/examples/doc/stclguide/deleteUser.tcl`. You can use the ProjectSync **User Profiles=>Add** command to add new users to a SyncServer, so that you can delete them using the `deleteUser.tcl` script. Make sure you have permission to delete users by checking the ProjectSync `DeleteUser` access control in your `AccessControl` files: `<SYNC_DIR>/share/AccessControl`, `<SYNC_ENT_CUSTOM>/share/AccessControl`, `<SYNC_SITE_CUSTOM>/share/AccessControl`, or `<SYNC_CUSTOM_DIR>/servers/<host>/<port>/share/AccessControl`. See the ENOVIA Synchronicity Access Control Guide for more information.

Related Topics

Developing Server-Side stcl Scripts

Server Triggers

URL stcl Script Requests

URL stcl Script Requests

You instruct a SyncServer to execute an stcl script by issuing a URL from your web browser, for example:

```
http://myserver:2647/scripts/isynch.dll?panel=TclScript&file=del
eteUser.tcl&terminate_user=hal
```

The SyncServer interprets this URL, invokes the stcl interpreter, and loads the file specified with the `file` argument.

URL Script Request Synopsis

```
http://<host>:<port>/scripts/isynch.dll?panel=TclScript&file=<sc
ript>&<parmlist>
```

where

<host> and <port> are the server's machine name and port number (for example: myserver.myco.com:2647)

<script> is the name of the stcl script that the server will execute. The script name must have a .tcl extension (for example: deleteUser.tcl).

<parmlist> is an optional list of parameters with name/value pairs to pass into the script (for example: parm1=value1&parm2=value2&parm3=value3). Separate each parameter from the previous one using an ampersand (&). Separate the name from the value using an equal sign (=). These parameters and values cannot contain spaces. **Note**: Certain characters must be encoded. For example, because these parameters and values cannot contain spaces; you must replace spaces by the + character or enclose the entire parameter list in quotes ("). If you are not sure how a character in the parameter list should be encoded, you can use the formatQuery command from the Tcl http package to encode your parameter list, for example:

```
http::formatQuery author georgia title "This is the title" url
sync:///Projects/chip

=>
author=georgia&title=This+is+the+title&url=sync%3a%2f%2f%2fProj
ects%2fchip
```

To Run a Script by Specifying a URL stcl Script Request:

The following example shows how to run the sample script: <SYNC_DIR>/share/examples/doc/stclguide/deleteUser.tcl. You must first use the ProjectSync **User Profiles=>Add** command to add new users to a SyncServer, so that you can delete them using the deleteUser.tcl script. Make sure you have permission to delete users by checking your AccessControl files.

To run the deleteUser.tcl script, follow these steps:

1. Copy the <SYNC_DIR>/share/examples/doc/stclguide/deleteUser.tcl file into your <SYNC_SITE_CUSTOM>/share/tcl directory.

   To create a server-specific script, you can instead copy the script to: <SYNC_CUSTOM_DIR>/servers/<host>/<port>/share/tcl. **Note**: By default, <SYNC_SITE_CUSTOM> resolves to <SYNC_CUSTOM_DIR>/site.

2. Run the deleteUser.tcl script by entering the following URL in your web browser:

```
http://<host>:<port>/scripts/isynch.dll?panel=TclScript&fil
e=deleteUser.tcl&terminate_user=username
```

where `username` is the name of the user you are deleting.

Server Triggers

Another way that server-side stcl scripts can be invoked is through triggers. You can set up triggers to fire when certain events take place such as the modification of a note, the attachment of a note to an object, or DesignSync revision control commands such as `ci`, `co`, and `tag`. To set up a trigger, you develop a trigger script, place it in one of the custom `tcl` script directories, and register the trigger using the ProjectSync user interface. See Developing Server-Side stcl Scripts for general scripting help, including where to place your server-side trigger scripts.

The stcl environment passes in several environment variables to the trigger script depending on the reason the trigger was fired. See Accessing Environment Information from Server Scripts for descriptions of trigger-specific variables, as well as general environment variables available to all server scripts.

You register note-related triggers using the ProjectSync Add Trigger panel. You can define multiple triggers for different purposes and each trigger can have filters to control the circumstances under which triggers fire. These filters include wildcards to identify the NoteURLs and the attachments for which the triggers should fire. See *ENOVIA Sychronicity DesignSync Data Manager Administrator's Guide*: Creating Note Object Triggers to learn how to set up server-side triggers.

Note: The ProjectSync Add Trigger panel registers note-related trigger scripts. If your trigger script is not a note-related trigger script, register the script by including the `trigger create` command in a server-side script and invoking the script using a URL stcl script request from your web browser or the `rstcl` (remote stcl) command from a DesignSync client.

Related Topics

Developing Server-Side stcl Scripts

How to Run stcl Server Scripts

URL stcl Script Requests

stcl Scripting Tips

stcl Scripting Tips

The **ENOVIA Synchronicity stcl Programmer's Guide** is a guide to stcl scripting rather than a general purpose Tcl scripting guide. For general Tcl scripting instruction, see the list of Tcl scripting references in the Introduction.

Following are the stcl scripting guidelines. Also see Hints for First-Time Scripters for some tips to get you started with stcl scripting.

Use the Reset Server button if you update server scripts or access controls.

If you make modifications to a server-side script or an access control file, use the ProjectSync Reset Server button to force the SyncServer to reread your files.

To determine the Tcl version, use the info tclversion command.

You can determine the version of Tcl included in your DesignSync installation's stcl interpreter by using the `info tclversion` and `info patchlevel` commands within a stcl/stclc client shell.

Use the url and note commands to access DesignSync and ProjectSync web objects.

You can use the `url` and `note` commands within your Tcl commands. For example:

```
if {[access verify EditUser $SYNC_User \
    [url leaf $user] 0]} {
  url properties $user props
}
```

The `url leaf` command returns the leaf of the `$user` URL and the `url properties` command returns an array of property name/value pairs. Notice how the `url leaf` command is used as an argument to another stcl command, `access verify`. The `url` commands are available from all DesignSync client shells. However, you cannot operate on a return value in dss/dssc, so the `url` commands are more useful in stcl/stclc.

See Accessing Web Objects for details.

Use command-line editing within stcl shells.

The stcl shells provide command-line editing support. See DesignSync Help: Command-Line Editing.

Add exception and data handling to your stcl scripts.

You must ensure that your scripts terminate gracefully and that data returned by scripts and commands is handled appropriately. To do so, review the guidelines in Return Values and Exception Handling.

Use appropriate commands to format output.

Client- and server-side scripts have different requirements for output formatting. Client-scripts are processed by the stcl interpreter; thus, you can use standard Tcl commands like `puts` and `format` to format your output. The results of server-side scripts invoked as URL script requests are sent to an HTML browser; thus, you format these types of results using HTML. See Output Formatting for details.

You can abbreviate DesignSync commands in stcl scripts.

Within stcl scripts as well as dss scripts, you can abbreviate DesignSync commands. For example, you can abbreviate '`populate`' as '`pop`'. You might want to test out a command abbreviation within an stclc shell before using it in a script to make sure there is only one command with that abbreviation; you need to provide enough characters for the abbreviation to resolve to a unique command.

Hints for First Time Scripters

- Running a Clean Environment
- Using Commands
- General Formatting
- Delimiting Strings and Whitespace

As a new stcl scripter, review the following list of potential stcl 'gotchas':

Running a Clean Environment

When a DesignSync tool, such as stclc or DesSync launches, it may change or set environment variables, or underlying default paths as part of creating the environment it needs to operate. This may result in the unpredictable results if the tcl script or a user manually running commands is unaware of these changes.

To create a clean environment, that does not use the changes introduced by the DesignSync in which the command or script is being run, use the `clean_exec` command instead of the `exec` command to allow any commands that use exec to use the system level path and environment settings, rather than the DesignSync client modified path or environment variables.

Using Commands

- Within stcl scripts as well as dss scripts, you can abbreviate DesignSync commands. For example, you can abbreviate '`populate`' as '`pop`'. You might want to test out an abbreviation within an stclc shell before using it in a script to make sure there is only one command with that abbreviation; you need to provide enough characters for the abbreviation to resolve to a unique command.
- You can call stcl commands as arguments to other stcl commands using Tcl command substitution, for example, `url contents [url vault Asic/x.v]`. See Accessing Objects Using url Commands for more examples.
- If you want to run a single DesignSync command from an OS shell, you can precede the command with 'dss' or 'dssc'. The syntax for specifying a single command is more complex for stcl. You must specify the `-exp` option to the `stcl` command to execute a DesignSync command from the OS shell. Because stclc is primarily a scripting shell, an argument specified without `-exp` is assumed a script. See stclc and stcl Clients for details.

General Formatting

- You must quote objects that contain a semicolon (;), such as vaults, branches, and versions.

  Version and vault URLs contain semicolons (`sync://localhost:2647/Projects/Asic/x.v;`). Because the semicolon is a Tcl command separator, you must quote semicolons in URLs using quotes or curly braces. You can also escape semicolons using the backslash (\ ) character:

  ```
  "sync://localhost:2647/Projects/Asic/x.v;"

  {sync://localhost:2647/Projects/Asic/x.v;}

  sync://localhost:2647/Projects/Asic/x.v\;
  ```

- You must use spaces as separators; do not rely on brackets or quotes to separate sections of code. For example, the following statement is incorrect because a space is required as a separator before the `url container` command substitution:

  ```
  INCORRECT: set notetypename [url leaf[url container $note]]

  CORRECT: set notetypename [url leaf [url container $note]]
  ```

- You can use the backslash character to continue a long statement onto the next line:

  ```
  puts [format "Registered object modified: %s"\
    [url path $obj]]
  ```

- For comments in Tcl code, the pound # character must be the first character in the line. You can, however, append a comment to the end of a line if you include a semicolon directly before the # character to explicitly terminate the command:

```
set users {jack jamie stan}  ;#  Create user list
```

Delimiting Strings and Whitespace Tips

You use curly braces { } and quotes " " to delimit strings and to handle whitespace within strings. A question that can cause confusion for first-time Tcl scripters is when to use quotes and when to use curly braces. The following are some differences between curly braces and quotes:

- Tcl performs variable substitution (also called 'variable interpolation') on expressions within quotes, but not within curly braces.
- Tcl allows nesting of curly braces, but not quotes.

Here are suggestions for quotes and curly braces:

	Argument contains whitespace:	**Argument contains no whitespace:**
Argument requires variable substitution:	Use quotes: `puts "The time is $time"`	Use neither quotes nor curly braces: `puts time=$time`
Argument does not require variable substitution:	Use curly braces: `puts {Now is the time...}`	Use neither quotes nor curly braces: `puts done`

Note that the dss shell does not support the curly brackets for delimiting whitespace; only quotes are supported in dss shell and scripts.

For example, the following works in both dss and stcl shells and scripts:

```
ci -comment "Fixed defect 1234" top.v
```

Using curly braces works only in stcl shells and scripts:

```
ci -comment {Fixed defect 1234} top.v
```

Return Values and Exception Handling

Tcl commands return a string value. When a Tcl command completes successfully, the return string contains the result of the command. In many cases, you must add Tcl code to process the results of stcl commands -- both to handle particular return value data types and to handle errors raised by stcl commands and scripts.

Return Values - dss Versus stcl

The return values of the DesignSync commands differ depending on whether you invoke them from a dss script (or shell) or from an stcl script (or shell). Return values of commands in a dss shell or script are not intended to be operated upon programmatically; unlike Tcl and stcl, the dss scripting language does not support programming constructs such as loops and conditionals, needed to process the return values effectively. Whereas stcl commands return string values, empty strings, or error messages, dss commands do not return values. Instead, each dss command returns an exit code of '0' or a non-zero integer. The returned exit code of a dss command is always '0' unless its corresponding stcl command would have returned an error message (thrown an exception), in which case, the dss exit code is non-zero. For revision control commands that operate on multiple objects, the dss exit code is '0', unless the operation fails on **all** objects, in which case the exit code is non-zero. In stcl, you can operate on the results of these commands to discern the number of successes and failures. The following examples illustrate these return values. In these examples, file `x.v` exists but file `top.v` does not. Also assume that there are no connection failures, so the checkin of `x.v` will succeed.

```
stcl> catch {ci -new -noc x.v top.v} error

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d1/top.v

Checking in: x.v: Success - New version:
1.1 on New branch: 'Trunk' (1)

Checkin operation finished.

##### WARNINGS and FAILURES LISTING #####

#

# Unable to find :
file:///home/rsmith/d1/top.v
```

A Tcl exception is not raised because at least one of the checkins was successful.

```
#

########################################

0

stcl> set error
{Objects succeeded (1)} {Objects failed
(1)}
```

```
% dss ci -new -noc top.v x.v

Logging to
/home/rsmith/dss_11142011_153804.log

V6R2013

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d2/rop.v

Checking in: x.v : Success - New version:
1.1 on New branch: 'Trunk' (1)

Checkin operation finished.

##### WARNINGS and FAILURES LISTING #####

#

# Unable to find :
file:///home/rsmith/d2/rop.v

#

########################################

{Objects succeeded (1)} {Objects failed
(1)}

% echo $?

0
```

The same command in dss returns an error code of 0.

```
stcl> catch {ci -new -noc top.v} error

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d1/top.v

Checkin command exiting.

1

stcl> set error

{} {Objects failed (1)}

stcl>
```

This checkin has no successes, so an exception is raised.

```
% dss ci -new -noc top.v

Logging to
/home/rsmith/dss_11142011_154504.log

V6R2013

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d1/top.v

Checkin operation finished.

{} {Objects failed (1)}

% echo $?

1
```

The same command in dss returns a non-zero error code.

See Return Values of Revision Control Commands for other examples. **Note**: To add exception handling for dss scripts, you can create an stcl wrapper script that invokes the command and provides exception handling. Then, you use the `alias` command to expose the stcl script (see dssc and dss Clients for an example).

Return Values and Error Codes

The stcl commands yield strings that represent different types of return vales. Some commands return strings that represent boolean values of "0" and "1". Some commands return empty lists. Consult the **ENOVIA Synchronicity Command Reference** for the types of return values associated with the commands. The `url` command descriptions in the **ENOVIA Synchronicity Command Reference** also show the return values associated with each type of web object.

Note that return values are not the same as return codes. Return codes are codes set by stcl or Tcl procedures to indicate that an exception has been raised. In addition to returning values, an stcl command called in an stcl script or shell can set an error code, thus throwing an exception (raising an error). Note that this error code is not a returned value; if an exception is raised, no value is returned, as in the following example:

```
stcl> set return [ci -new -noc x.v top.v]

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d3/top.v

Checking in: x.v
                            : Success - New
version: 1.1 on New branch: 'Trunk' (1)

Checkin operation finished.

##### WARNINGS and FAILURES LISTING #####

#

# Unable to find :
file:///home/rsmith/d3/top.v

#

########################################

{Objects succeeded (1)} {Objects failed
(1)}

stcl> set return

{Objects succeeded (1)} {Objects failed
(1)}
stcl> set return ""
```

A Tcl exception is not raised because at least one of the checkins was successful.

An exception occurs because

```
stcl> set return [ci -new -noc top.v]

Beginning Check in operation...

Unable to find :
file:///home/rsmith/d3/top.v

Checkin operation finished.

{} {Objects failed (1)}

stcl> set return

stcl>
```

the ci command had no successes. Notice that the `return` variable is not set because the ci command did not return a value. Because an exception is raised, no value is returned.

Many commands that return error codes also display error messages. Error messages resulting from client-side stcl scripts display in the client, whereas error messages from server-side stcl scripts display in the web browser where you've invoked the URL script request. If there are messages about access controls, the server writes these messages to the `error_log` file, `<SYNC_CUSTOM_DIR>/servers/<host>/<port>/logs/error_log`.

You can detect exceptions using the Tcl `catch` command. In terms of errors, some commands throw exceptions, whereas other commands return empty lists to reflect an unsuccessful command. The commands that throw exceptions can cause your script to fail. In these cases, you need to provide exception handling code, typically using the Tcl `catch` statement described in Handling Errors Raised by stcl Commands and Scripts.

Return Values of Revision Control Commands

Most revision control commands operate on multiple objects. For these commands, the return value is a list containing two strings:

```
{Objects succeeded (3)} {Objects failed (1)}
```

The first string indicates the number of objects processed successfully. The second string indicates the number of objects whose processing failed. If, there are no failures, the second string is an empty string:

```
{Objects succeeded (4)} {}
```

Likewise if there are no successes, the first string is an empty string. If all objects fail, an exception occurs (the return value is thrown, not returned). The meaning of a

successful operation varies by command. For example, if you attempt to check in a file that has not changed, the checkin does not occur, yet the operation is counted as a success. See the command descriptions in the **ENOVIA Synchronicity Command Reference** to understand what a successful operation is for each command.

The following revision control commands yield a return value format of the type shown above:

- `cancel` - Cancels a previous checkout operation
- `ci` - Checks in the specified objects
- `co` - Checks out the specified objects
- `mkbranch` - Creates a new branch
- `populate` - Creates or updates a local work area
- `setselector` - Sets the persistent selector list
- `tag` - Assigns a tag to a version or a branch

Note: If you call these commands from a dss shell or script, the exit code is '0', unless the operation fails on **all** objects, in which case the exit code is a non-zero integer.

You can test non-empty lists returned by a revision control command for successes and failures by parsing the return value. This method is not encouraged, as the format of these returned values is subject to change in future releases. The following example shows how to use the Tcl `lindex` command to parse these results. **Note**: You can also capture the output of a command to parse it for other information using the stcl `record` command (see Capturing Output of Commands).

```
stcl> set ret [co *.txt]

Beginning Check out operation...

Checking out: log-Windows_NT-201-042700-182419.txt : Failed:som:
Error 62: Not under revision control

Checking out: prefixes.txt : Success - Already fetched, and
still unmodified version 1.12.2.2

Checking out: ReadMe.txt : Success - Already fetched, and still
unmodified version 1.1.26.2

Checking out: telephones.txt : Success - Already fetched, and
still unmodified version 1.150.2.69

Checkout operation finished.

{Objects succeeded (3)} {Objects failed (1)}
```

```
stcl> lindex $ret 0

Objects succeeded (3)

stcl> lindex $ret 1

Objects failed (1)
```

Return Values of Note-Related Commands

Server-side commands that operate on notes do not necessarily return valid values; thus, you must trap and test the values returned. Note properties (fields) can contain an initial null value (""). Because you cannot rely on properties of numeric or Boolean types to return values of 0 or False, some operations on Boolean or numeric properties can fail.

A script with the following call to url getprop terminates if `NumHits` is unset because the `incr` command expects an integer rather than an empty string (""):

```
incr hits [url getprop $note NumHits]
```

The following code traps and tests the `NumHits` property:

```
set numhits [url getprop $note NumHits]
if {$numhits != ""} {
  incr hits $numhits
}
```

Handling Data Returned by stcl Commands

Although a Tcl return value is always a string, the data represented by the string can be of a variety of data types, such as a boolean, a list, or an array. You must ensure that the code processing a return value determines the type of data the string represents so that it can perform a data type conversion, if necessary.

All of the Tcl rules for data types apply. For example, a common return value type is a list. In processing a returned list, use the family of list commands such as `llength`, `lindex`, and `lappend` to manipulate the list. See the Tcl references listed in the Introduction to learn more about these commands.

To help you determine whether you need to provide data handling such as converting the results of an stcl command, see the `url` command descriptions in the **ENOVIA Synchronicity Command Reference**, which provide the return values corresponding to each type of revision control object.

Return values that represent strings sometimes contain braces { }, a Tcl grouping operator. An element of a list that contains separators (such as spaces or semicolons) is enclosed by braces to clarify that it is one element. The braces are part of the list structure and not part of the value of the list element. In the following example, the second element in the return value is enclosed in braces because the semicolon might otherwise be interpreted as a separator:

```
stcl> url contents sync://localhost:2647/Projects/Asic

sync://localhost:2647/Projects/Asic/Sub

{sync://localhost:2647/Projects/Asic/x.v;}
```

You do not need any special commands to detect these braces because the Tcl list commands you use to operate on these returned lists handle the braces automatically.

Handling Errors Raised by stcl Commands and Scripts

If a command fails (for example, if you specified the incorrect command syntax), then the command aborts and returns a string containing an error message. If a command fails within a script, all commands that follow the failed command are skipped. However, Tcl supports exception handling, which lets you catch errors and handle them more gracefully, by stopping the script execution and providing a customized error message.

It is good coding practice to add exception handling to your calls to stcl commands. Doing so can provide additional debugging information when your command fails, and also allows any commands that follow your command (if you are running a script) to execute.

Note: The `rmfile`, `rmversion`, `rmvault`, and `rmfolder` commands do not throw exceptions if objects you are trying to delete do not exist. If you are removing a group of objects, the operation continues even if an object does not exist. To ensure the correct use of the remove commands in your scripts, perform a check ahead of time to make sure the objects exist before you invoke one of the remove commands.

Using the catch Command to Handle Errors

The following Tcl statement shows how you can use the catch command to handle exceptions.

```
stcl> if {[catch {syncNeedCheckin .} msg ]} {puts "Error: $msg"}

Connect failure. Server 'alserv' may have reset the connection.

Folder: /home/karen/Asic/Sub

Connect failure. Server 'alserv' may have reset the connection.
```

```
Error: som-E-11: Communication Connect Failure.
```

The catch example shows how you can use a script or procedure as an argument to the `catch` command to handle potential exceptions in your scripts or procedures. In this case, the `url contents` stcl command could not access the server and thus returned an error code. This error code was passed back to the catch command. The `catch` command checks the error code of the script or procedure, not the output or return value. Notice from the output that the procedure runs as normal and prints to standard output before terminating.

You can also use a `catch` statement as a wrapper script for your procedures. The `syncNeedCheckin2` procedure below uses a `catch` statement as a wrapper to catch exceptions in the `syncNeedCheckin` script.

```
proc syncNeedCheckin2 arg {
  if {[catch {
    foreach obj [url contents $arg] {
      if {[url getprop $obj type] == "Folder"} {
        puts [format "Folder: %s\n" [url path $obj]]
        syncNeedCheckin2 $obj
      } else {
        if {[url modified $obj]} {
          if {[url registered $obj]} {
            puts [format "Registered object modified: %s\n"\
                [url path $obj]]
          } else {
            puts [format "Unregistered object: %s\n"\
                [url path $obj]]
          }
        }
      }
    }
  } msg ]} {puts "syncNeedCheckin2 Error: $msg"}
}
```

The following stcl session example shows how the `catch` statement handles an exception:

```
stcl> syncNeedCheckin2 sync://localhots:2647

DNS Lookup failure on Server 'localhots'

DNS Lookup failure on Server 'localhots'

DNS Lookup failure on Server 'localhots'
```

```
syncNeedCheckin Error: som-E-4: Communication Failure.
```

For more details on catch statements and exception handling, refer to one of the Tcl references in the Introduction.

Capturing Output of Commands

Sometimes you might want to tailor the output for users of your scripts by customizing the output of the stcl commands. You can use the stcl `record` command to trap a command's output so that you can customize what the user sees. You can then parse this output to create your own messages. **Important**: This method of parsing the recorded output, although useful, is discouraged because the stcl command output messages are subject to change. Thus, you might need to update your parsing code upon upgrading your DesignSync software.

The following are some examples where you might use the `record` command to capture output:

- If you create an stcl script that calls a number of stcl commands, you might not want the output of each command to display for users. Instead, you can use the record command to capture the output of each command, then you can create your own message that summarizes the success or failure of the entire task rather than that of each stcl command.
- Some stcl commands provide no return values, yet print informational messages. You can create a wrapper script by calling the stcl command within the record command. The record command captures the informational messages, and you can parse the captured message to create a customized return value for the wrapper script.
- If you find the default output of a particular stcl command inadequate for your users' purposes, you can record it and tailor the output message.

The following script shows how you can use the record command to process the results of a populate command to ensure that a successful populate of an entire workspace has occurred. The script uses Tcl string processing commands such as `split` and `string` to parse the output saved in the `popOutput` variable.

```
if [catch {record {populate -recursive } popOutput} msg] {
  puts "Populate failed: $msg"
}

# now scan for fail
set failOutput ""
foreach line [split $popOutput \n] {
  if [string match *Fail* $line] {
    set failOutput "$failOutput\n$line"
    set hasError 1
```

```
  }
}
```

The following stcl session shows how the script captures the output of a recursive populate command, then parses the output for failures. The script is sourced and the `failOutput` and `hasError` variables store the error information:

```
stcl> source ../tclscripts/custompop.tcl

stcl> set hasError

1

stcl> set failOutput

Sub/y.v  : Failed:som: Error 135: Can't overwrite local changes.
 Use force switch.

stcl>
```

Output Formatting

You format output in stcl scripts differently depending on whether the script is a client- or server-side script. The output from server-side scripts is typically displayed on the user's browser, whereas, the output from client-side script displays in the client--the dss, stcl, or DesSync window.

Server Script Output Formatting

For server-side scripts, you can format output using HTML, for example:

```
puts "<h2>Permission denied: you cannot delete this user</h2>"
```

A method of formatting your output text is to include your script between the following two HTML preformatted text calls:

```
puts "<PRE>"

...

puts "</PRE>"
```

Important: If your server-side script will be invoked from a client or a client script using the `rstcl` (remote stcl) command, do not include HTML formatting; the HTML formatting commands are not processed and HTML commands such `<PRE>` appear in the output text.

To Format Lists

Many of the `url` and `note` commands return unformatted lists which are difficult for users to read. You can format these lists using HTML ordered or unordered lists, shown in the following excerpt of a server-side script:

```
puts "<PRE>"
puts "All users: "
set userlist [url users sync:///]
puts "<OL>"
puts "<LI>[join $userlist "<LI>"]"
puts "</OL>"
puts "</PRE>"
```

This stcl code returns:

```
All users:
1. sync:///Users/karen
2. sync:///Users/alex
3. sync:///Users/sal
4. sync:///Users/asicdev
5. sync:///Users/cdent
6. sync:///Users/jboswell
7. sync:///Users/anabel
```

If your users will be invoking the script using the `rstcl` command, leave out the HTML formatting and handle the return values using the Tcl split command. For example, the following client-side script, `lsUsersFromClient.tcl` uses the `rstcl` command to call a server-side script, `lsUsers.tcl`. To process the return value of the `lsUsers.tcl` script, the client script uses the split command to split the return value list into strings separated at the carriage return characters (`\n`):

Example: lsUsersFromClient.tcl Client Script

```
proc lsUsersFromClient {} {
  set users [rstcl -server sync://localhost:30048 \
            -script lsUsers.tcl]
  set users [split $users \n]
  foreach user $users {puts $user}
}
```

Here's sample output of the `lsUsersFromClient.tcl` script:

```
stcl> source tclscripts/lsUsersFromClient.tcl

stcl> lsUsersFromClient
```

```
Karen Mendola

Alessandro Christiani

jerry

Asic Developers

Charles Dent

Jean Boswell

Anabel Blythe
```

lsUsers.tcl Server Script Called by the lsUsersFromClient.tcl Client Script

```
foreach user [url users sync:///] {
  puts [url getprop $user name]
}
```

Client Script Output Formatting

For client-side scripts, you can use the Tcl format command as a command substitution within a puts command. The format command is similar to the C printf function:

```
  puts [format "Folder: %s" [url path $obj]]
```

See a Tcl language reference manual for details.

Getting Assistance

Using Help

ENOVIA Synchronicity DesignSync Data Manager Product Documentation provides information you need to use the products effectively. The Online Help is delivered through WebHelp® , an HTML-based format provided by eHelp™ Corporation that is displayed in your Web browser.

Note: Use SyncAdmin to change your default web browser, as specified during ENOVIA Synchronicity DesignSync Data Manager tools installation.

To bring up the online help from the tool you are using, do one of the following:

- Select **Help => Help Topics** from the tool you are using. The help system opens in your default browser. The Contents tab displays in the left pane and the corresponding help topic displays in the right pane.
- Click **Help** on forms. The help system opens to the topic that describes the form.
- Press the **F1** key. The help system opens to the topic that describes the current form or window you have open.

To bring up stand-alone Online Help, do one of the following:

- Enter the correct URL from your Web browser:

  `http://<host>:<port>/syncinc/doc/<docname>/<docname.htm>`

- where <host> and <port> are the SyncServer host and port information. Use this server-based invocation when you are not on the same local area network (LAN) as the DesignSync installation.

  For example:

  `http://<host>:<port>/syncinc/doc/stclguide/stclguide.htm`

  `http://<host>:<port>/syncinc/doc/DesSync/dessync.htm`

- Enter the following URL from your Web browser:

  `file:///$SYNC_DIR/share/content/doc/<docname>/<docname.htm>`

  For example:

  `file:///$SYNC_DIR/share/content/doc/stclguide/stclguide.htm`

```
file:///$SYNC_DIR/share/content/doc/DesSync/dessync.htm
```

where $SYNC_DIR is the location of the DesignSync installation. Specify the value of SYNC_DIR, not the variable itself. Use this invocation when you are on the same LAN as the installation. This local invocation may be faster than the server-based invocation, does not tie up a server process, and can be used even when the SyncServer is unavailable.

When the Online Help is open, you can find information in several ways:

- Use the **Contents** tab to see the help topics organized hierarchically.
- Use the **Index** tab to access the keyword index.
- Use the **Search** tab to perform a full-text search.

Within each topic, there are the following navigation buttons:

- **Show** and **Hide**: Clicking these buttons toggles the display of the navigation (left) pane of WebHelp, which contains the Contents, Index, and Search tabs. Hiding the navigation pane gives more screen real estate to the displayed topic. Showing the navigation pane givens you access to the Contents, Index, and Search navigation tools.
- **<<** and **>>**: Clicking these buttons moves you to the previous or next topic in a series within the help system.

You can also use your browser navigation aids, such as the **Back** and **Forward** buttons, to navigate the help system.

Related Topics

Getting a Printable Version of Help

Getting a Printable Version of Help

The *ENOVIA Synchronicity DesignSync Data Manager CD User's Guide* is available in book format from the ENOVIA Documentation CD or the DSDocumentationPortal_Server installation available on the 3ds support website (http://media.3ds.com/support/progdir/). The content of the book is identical to that of the help system. Use the book format when you want to print the documentation, otherwise the help format is recommended so you can take advantage of the extensive hyperlinks available in the DesignSync Help.

You must have Adobe® Acrobat® Reader™ Version 8 or later installed to view the documentation. You can download Acrobat Reader from the Adobe web site.

Contacting ENOVIA

For solutions to technical problems, please use the 3ds web-based support system:

http://media.3ds.com/support/

From the 3ds support website, you can access the Knowledge Base, General Issues, Closed Issues, New Product Features and Enhancements, and Q&A's. If you are not able to solve your problem using this information, you can submit a Service Request (SR) that will be answered by an ENOVIA Synchronicity Support Engineer.

If you are not a registered user of the 3ds support site, send email to ENOVIA Customer Support requesting an account for product support:

enovia.matrixone.help@3ds.com

Related Topics

Using Help

Index