

---

# TACTIC Developer

## Table of Contents

Introduction .....	4
Development Concepts .....	4
Architecture Overview .....	4
The TACTIC Editor .....	7
Client API .....	8
Client API Setup .....	8
Client API Structure .....	11
Basic Operations .....	13
Checkin / Checkout Operations .....	17
Snapshot Dependency .....	23
Widgets .....	25
TACTIC Widgets .....	25
Custom Interfaces .....	29
Expression Language .....	36
Using Expressions in Scripting .....	36
Validation .....	39
Validation Set-up .....	39
TACTIC Python Client API Reference .....	40
__init__ .....	41
abort .....	42
add_config_element .....	43
add_dependency .....	45
add_dependency_by_code .....	46
add_directory .....	47
add_file .....	49
add_group .....	52
add_initial_tasks .....	53
build_search_key .....	54
build_search_type .....	55
checkout .....	56
clear_upload_dir .....	57
create_search_type .....	58
create_snapshot .....	59
create_task .....	60
delete_object .....	61
directory_checkin .....	62
download .....	63
eval .....	64
execute_cmd .....	65
execute_pipeline .....	66
execute_python_script .....	67
finish .....	68
get_all_children .....	69
get_all_dependencies .....	70
get_all_paths_from_snapshot .....	71
get_by_search_key .....	72
get_child_types .....	73

get_client_api_version .....	74
get_client_dir .....	75
get_client_version .....	77
get_column_info .....	78
get_column_names .....	79
get_config_definition .....	80
get_dependencies .....	81
get_expanded_paths_from_snapshot .....	82
get_full_snapshot_xml .....	83
get_handoff_dir .....	84
get_home_dir .....	85
get_info_from_user .....	86
get_login .....	87
get_md5_info .....	88
get_parent .....	89
get_parent_type .....	90
get_path_from_snapshot .....	91
get_paths .....	93
get_pipeline_processes .....	94
get_pipeline_processes_info .....	95
get_pipeline_xml .....	96
get_pipeline_xml_info .....	97
get_preallocated_path .....	98
get_project .....	100
get_protocol .....	101
get_related_types .....	102
get_resource_path .....	103
get_server_api_version .....	104
get_server_name .....	105
get_server_version .....	106
get_snapshot .....	107
get_ticket .....	109
get_types_from_instance .....	110
get_unique_subject .....	111
get_virtual_snapshot_path .....	112
get_widget .....	114
group_checkin .....	115
insert .....	116
insert_update .....	118
log .....	119
query .....	120
query_snapshots .....	122
reactivate_subject .....	123
redo .....	124
retire_subject .....	125
set_current_snapshot .....	126
set_login_ticket .....	127
set_project .....	128
set_protocol .....	129
set_server .....	130
simple_checkin .....	131
split_search_key .....	133
start .....	134
undo .....	135

update ..... 136

update\_config ..... 137

update\_multiple ..... 138

upload\_file ..... 139

# Introduction

## Development Concepts

### Introduction

The term "asset" is used often, and has many different meanings in different industries and even in different areas of the same production facility. In TACTIC, an asset is an *atomic entity* with metadata and files associated with it. To avoid confusion, the TACTIC assets are called "searchable objects," shortened to *sObjects*.

### sObjects

sObjects are the atomic entities (or assets) that TACTIC uses to manipulate data and check in files. An sObject can be any entity required in a production. Examples of sObjects include shots, textures, users, tasks, production notes, and so on.

Every sObject must belong to a search type, also known as sType. *Search types* are a set of unique string entities that serve to classify all variations of sObjects. Search types are registered in the "search\_object" table in the "sthpw" database. This table defines the properties for each search type, and is used to ensure that sObjects adheres to their search type properties. For instance, in a custom project, you may have a custom/shot sType created for shot. Once it's registered, you can add shot entries in the shot table that it generates. The shot entries are the shot sObjects.

It is technically possible to store data on assets anywhere, but the TACTIC approach is to use an SQL database so sObject data can be tracked in the database and rules can be enforced. In TACTIC, each sObject is represented as a table in the database. All sObjects for your project are stored in a project-wide database and cross-project sObjects (for example, those related to users) are stored in the main TACTIC database "sthpw."

## Architecture Overview

The TACTIC architecture is an MVC architecture with the following major components:

<b>SObject - Model(M)</b>	Provides the data model. All interactions with the data model use SObjects and their derived classes.
<b>Widget - View(V)</b>	Provides the display model, which determines the user interface and how users interact with the web application. The display architecture is built upon hierarchical widgets that are SObject-aware (that is, they use SObjects to define the interface).
<b>Command - Command(C)</b>	Provides higher-level interactions with the data model. All actions affecting the data model or the filesystem must go through a command layer so that the changes can be tracked and completely undoable should something go wrong.
<b>Search</b>	Provides a search model so widgets can obtain the SObjects they need to complete the interface display. Each type of SObject has a registered name which is used in the search engine to identify which SObject type to search. This provides a consistent interface to access all SObjects regardless of the location of the SObject in the database or table.

In summary, widgets make use of the Search, get SObjects, and use commands to change persistent data. The SObject communication unit binds the view layer with the data model.

## Main Data Objects

SObjects (searchable objects) are atomic, self-contained units that contain attributes. A particular SObject can be uniquely identified by two parameters: a search type and a search ID. Often these two parameters are combined into a "search key" defined as <search\_type>|<search\_id> (joined with the "|" character). Search keys allow you to uniquely identify any SObject using a single string.

Particular SObjects are obtained using the search engine, which generally returns a list of SObjects. The search engine is flexible enough to allow arbitrary bits of SQL code to be used for a search, although that approach is discouraged. (To maximize code reuse, it is better to put SQL code inside the low-level business objects that provide static functions to higher level parts of the framework.)

Widgets are the atomic drawing units. Typically, widgets are SObject-aware and can perform and affect searches and draw SObjects. Widgets can contain children, and many function calls will traverse down to their children. For example, a widget can be assigned a search object. It will perform this search and pass the results to all of its children widgets, who will make use of the result as necessary.

One important widget function is the `get_display()` function, which draws widgets and can generate HTML. This function can be as simple as just drawing something that has nothing to do with SObject data, or can be a complicated function retrieving and displaying SObjects and all of their child SObjects.

## Widgets

Widgets determine how users interact with the web application. They have a number of useful properties that allow for the rapid development of web applications. For example, they can have a search assigned to them to locate and retrieve SObjects. They can typically perform actions across the search results, affecting multiple SObjects.

Widgets call events and listen to events, allowing for inter-widget communication. They interact with each other in the web application by registering events. For example, one widget, on initialization, may register itself as a listener for a named event. Another widget may call the named event upon an arbitrary action, at which point all widgets that are registered listeners for that event will be executed. This type of interaction allows for multiple actions to occur as a result of a user interaction, such as the click of a single button.

Checkin/checkout is the framework for filesystem interaction. All interaction within the checkin/checkout framework is done through the SObjects themselves so that they can determine their own checkin/checkout conditions and mechanisms. The checkin framework creates a 'snapshot' SObject that is related to the original SObject through a `search_id`. It assigns a unique file ID for every transaction, and creates snapshot attributes for the SObjects.

Engineering requirements for a particular application must be gathered and translated into widgets, including definitions of the widgets' relationships to each other.

### AJAX Widgets

TACTIC's widget hierarchy falls naturally within the AJAX paradigm, where widgets are capable of redrawing themselves. Instead of refreshing the entire page, AJAX widgets actively gather the required information from the page and send only that information to the web server (as opposed to the entire contents of the page). The widget then processes the information and updates itself. This technique makes a much more interactive application because the web server only has to draw the individual widget element instead of the entire page. In addition to a faster and more interactive experience, AJAX widgets significantly reduce the overall load on the web server, making TACTIC far more scalable with the same resources.

TACTIC's interface runs on top the the client API, therefore all interaction between the client and the server run on an XMLRPC layer resting on top of AJAX. This is very convenient for complex interactions between the client and the server.

### Web Drawing Engine

This drawing engine is based on numerous interface platforms generally geared towards traditional application design. However, it has been adjusted to accommodate the unique web environment. A typical application would define a number of predefined widgets and assemble them in a hierarchical relationship.

Specialized widgets must be created to serve specific functions: for example, checkin/checkout widgets, download widgets, upload widgets, and navigation widgets.

## Persistent Store

All metadata is stored in an industry-standard SQL database. The database tables and rows are clearly marked and readable, so it is easy to access the data directly. In today's fast-changing environment, it is essential to be able to quickly read and understand the underlying data store to be able to maintain proper support for diagnosing and fixing problems.

All data is accessed through SObject entities, which provide the object relational mappings to the database tables. In general, a single SObject is represented by a row in the table of a database. The table defines the type of SObjects stored in it, and there is usually a one-to-one relationship between the attributes of each SObject and the columns in the database.

## Directory and File Naming Conventions

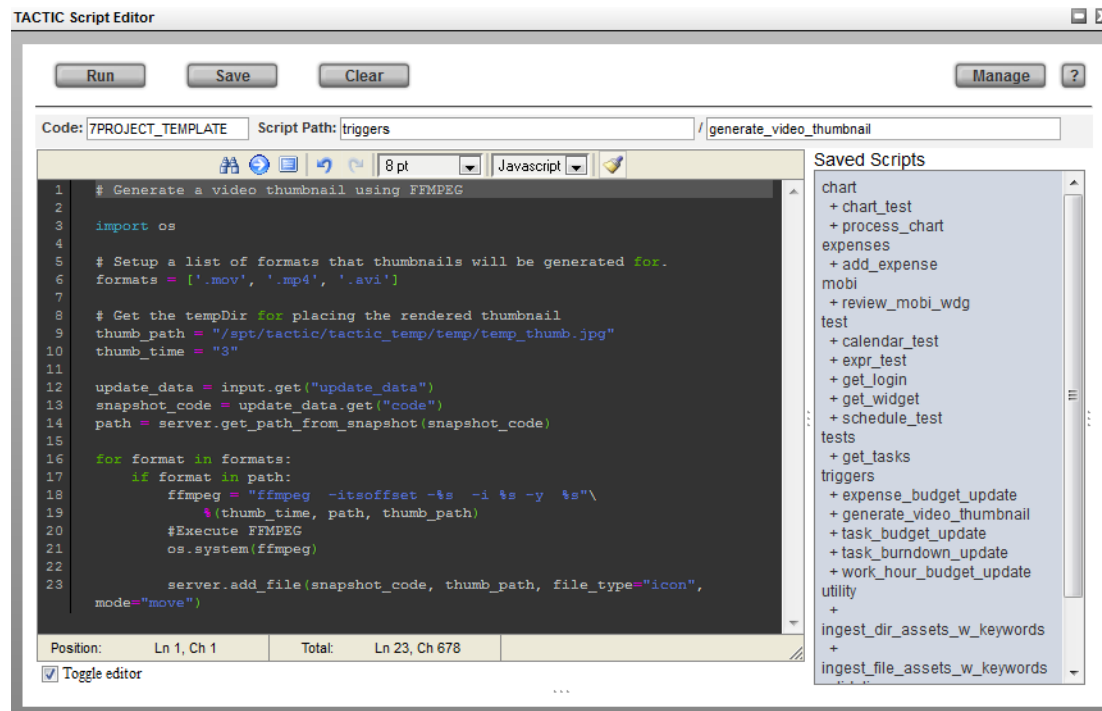
It is just as critical to be able to navigate the filesystem and understand what is located there. Therefore, all naming conventions are filtered through naming classes, which use clear procedures to create filenames based on metadata in the database.

Directories and file naming are handled slightly differently. TACTIC builds filenames procedurally and then stores them in the database. On the other hand, TACTIC never stores directory names directly in the database, but always builds them up procedurally. This additional level of abstraction provides the opportunity to reorganize your asset structure as needed (because the directory structure isn't hard-coded). Note that there may be other dependencies that are outside the control of TACTIC, so great care must be taken should you decide to reorganize the directory structure of your assets.

## The TACTIC Editor

The TACTIC Script editor allows for Javascript and Python based scripts to be written and stored in a "custom script" SObject. These scripts harness the power of Javascript in the web browser along with the power of the Python TACTIC Client API. They can be structured to run on a general execution, by a trigger or, they can be attached to a button to execute for a specific SObject.

One of the main benefits with using this method of custom scripting in TACTIC is that the script writer does not have to have direct access to the server's filesystem.



# Client API

## Client API Setup

### Important Note

Visit the Southpaw support site for more examples and tutorials on the API and its usage. The Support site is the place to go for wikis, forums, examples, and more.

### Setup

The easiest way to interact with the server from the client using the Client API is to use the provided server stub code. This code includes a class and a utility that are very useful for handling many of the details around client/server interaction and authentication.

The server stub code is housed in a client folder and can be found in the TACTIC installation in the directory:

```
<tactic_install_dir>/src/client
```

The first step is to copy the entire client folder over to the client machine (the machine that will be running the scripts) to a directory that will be visible to the user. Most facilities would likely put this folder in a centralized location so that every computer would be able to execute its scripts. The path to this folder must be specified in the PYTHONPATH environment variable on client machines so that it can be found by the scripts. For instance, if PYTHONPATH = L:/custom\_python, you would put the client folder in L:/custom\_python. Please refer to the Python documentation for more information.

### Settings

There are three important parameters for setting up the TacticServerStub to connect correctly :

- **server:** specifies the server that the server stub will connect to. This server can be a domain name ("localhost") or an IP address ("127.0.0.1"). It can even be a port number ("localhost:9000"). This setting allows you to switch between various TACTIC servers in your facility.
- **project:** specifies the current project. In TACTIC, the project is a state under which interactions occur.
- **ticket:** specifies the authentication ticket, a long alpha-numeric string that encrypts the login and password so that these values remain secure.

There are a number of methods to set these parameters.

The **first method** is to set the following parameters directly in the server stub reference:

```
server = TacticServerStub()  
server.set_server(tactic_server)  
server.set_project(project)  
# this is not needed if you have run python get_ticket.py  
server.set_ticket(ticket)
```

These settings override all settings obtained elsewhere. This method ensures that these values are set up correctly based on some external information.

To set up a server stub, you can insert the stub information in your script (described in the client API documentation as part of the get\_ticket() function). Or, you can run the script **get\_ticket.py**, which is included with the client API example set (located in <TACTIC\_INSTALL\_DIR>/src/client/bin). When the stub is run, it creates a ticket file on the user's machine which will be used each time any API script is run to authenticate which user is running the script.

The **second method** is through environment variables set up across the studio:

- TACTIC\_SERVER: sets the server that the server stub will connect to.
- TACTIC\_PROJECT: sets the project that the server stub will connect to.
- TACTIC\_TICKET: sets the authentication ticket.

This method can be used by programs that set up user environments, and has other advantages. It is easy to switch the settings using a shell variable. The program that sets up the environment does not have to be written in Python. It can even be simple to set up by using a shell command line to set the environment variables.

The **third method** makes use of a resource file located in the user's home directory. This resource file has a simple format:

```
login=joe
server=localhost
ticket=97d2bec3d73da71c14fb724a47af5053
project=bar
```

The login tag doesn't actually do anything here, since the user name is encapsulated in the ticket itself.

The **fourth method** is described below:

## Alternative way of using TacticServerStub without a ticket file

## Alternative way of using TacticServerStub without a ticket file

If you have written a GUI or have some means of retrieving the user's password on individual session instead, you can use the following construct to set the ticket. The server's IP and project should be set beforehand.

```
server = TacticServerStub.get()
server_IP = '10.10.50.100'
my.set_server(server_IP)
my.set_project('sample3d')

ticket = my.get_ticket(login, password)
my.set_ticket(ticket)
```

Once you have set up the environment for the client API to run correctly, you can try a sample script. The following simple script illustrates the structure of a TACTIC Client API program:

```
import sys
from tactic_client_lib import TacticServerStub

def main(args):
    server = TacticServerStub()
    server.start("Ping Test")
    try:
        print server.ping()
    except:
        server.abort()
        raise
    else:
        server.finish()

if __name__ == '__main__':
    executable = sys.argv[0]
    args = sys.argv[1:]
    main(args)
```

This simple program will ping the server and return "OK". If everything is set up correctly, you should be able to run this program from a shell as follows:

```
# python ping.py  
OK
```

If you see "OK", then you have successfully connected to the TACTIC server using the client API.

If you need to run `python get_ticket.py` first, it can be found under: `client/bin/get_ticket.py`.

# Client API Structure

## Directory Structure

The client API files are located in the directory <tactic\_install\_dir>/src/client. This directory contains all the files need for the client API. Typically you would copy all of the files in this directory to a location visible to the client machine.

There are a number of directories in this Client API directory:

- **bin:** contains useful supported scripts.
- **test:** contains unit tests for the client API.
- **examples:** contains a number of small examples to be used for reference.
- **tactic\_client\_lib:** the main directory for the Client API.

The main directory "tactic\_client\_lib" is the base module that you will use to access all of the TACTIC client APIs. Typically, you would import this module when working with the client API:

```
from tactic_client_lib import TacticServerStub
```

There are a number of subdirectories under tactic\_client\_lib:

- **tactic\_server\_stub.py:** contains the main server class "TacticServerStub". This class encapsulates all interactions to the TACTIC server and is generally the primary class used with the client API.
- **(ALPHA) application:** contains all the classes that deal with interaction with third-party applications. It provides an abstraction layer for applications and allows you to set data that can be used by TACTIC's introspection (verification).
- **common:** contains a number of convenience functions that are commonly used.
- **interpreter:** contains the client-side pipeline interpreter. This interpreter executes pipelines defined on the TACTIC server. These pipelines can be used to create highly complex modular client-side processes. Typical uses are for the checkin and checkout pipelines.
- **test:** contains a number of test classes used by the unit tests.

You should point to the Client API by having the directory src/client/tactic\_client\_lib stored somewhere accessible to client machines. Import the Tactic\_Server\_Stub with the following line in your script from tactic\_client\_lib:

```
import Tactic_Server_Stub
```

(For more details, visit the Southpaw Support site.)

## tactic\_server\_stub.py

This module contains the TacticServerStub class, which encapsulates all interactions with the TACTIC server. This class lets you make full use of the TACTIC architecture in your custom applications. Although the TacticServerStub can be instantiated, it is often preferable to use it as a singleton so you can set up the server once and make use of it from various locations in your applications:

```
from tactic_client_lib import TacticServerStub
server = TacticServerStub.get()
```

Once you have a reference to the TacticServerStub, you must set it up using three essential parameters: server, ticket, project. These parameters are described in more detail in the client API setup documentation.

## **interpreter**

This directory contains all the code needed to execute pipelines on the client. Pipelines in TACTIC are arbitrary process flow graphs. These pipelines have a number of advantages over other methods:

- They promote reusability, with each process handler having a consistent interface from which it can extract information. Typically, handlers are like mini programs which for the most part are compartmentalized and have little to do with each other.
- They can be visualized. Using the pipeline editor, the entire flow of the pipeline can be graphically visualized
- They can be specialized. Each aspect of the pipeline can be written by those team members most suited for the task.
- They lower the bar to creating complex pipelines. With a large library of well-written handlers, it becomes possible for non-developers to create pipelines by graphically piecing processes together.

## **application**

This directory handles all of TACTIC's interaction with third-party applications.

NOTE: this section is still in active development.

## Basic Operations

### Note

If you haven't done so, please review the Client API Setup doc.

### Simple Ping

The following is a skeleton script interacting with the Client API:

```
from tactic_client_lib import TacticServerStub

def main():
    server = TacticServerStub()
    server.start("Ping Test")
    try:
        print server.ping()
    except:
        server.abort()
        raise
    else:
        server.finish()

if __name__ == '__main__':
    main()
```

Executing this script will give the following output:

```
$ python examples/ping.py
OK
```

If you haven't had a ticket in the user directory, please run `python get_ticket.py`. Otherwise, you will get an error like this:

```
File "G:\TSI\3.0_client\client\tactic_client_lib\tactic_server_stub.py",
line 2789, in _setup raise TacticApiException(msg)
tactic_client_lib.tactic_server_stub.TacticApiException:
[C:/sthpw/etc/<someuser>.tacticrc] does not exist yet. There is not enough
information to authenticate the server. Either set the appropriate environment
variables or run get_ticket.py
```

The first line imports the `TacticServerStub` class. This class is a stub to the server and relays function calls between the TACTIC server and the client API code. It handles all the details of how to connect to the server. It also maintains status information, including the current project and whether or not the session is authenticated.

All client API scripts should run within a transaction. This requirement is achieved using `server.start("Ping Test")`, which initiates a new transaction on the server. All subsequent server interactions are grouped in the same transaction until `server.finish()` is executed. The function `server.abort()` is used to abort the transaction should any error occur in the body of the code.

### Querying data

The most fundamental operation in the Client API is the query function, which enables access to direct information on an SObject

The following example illustrates the use of the query function:

```
# define the search type we are searching for
```

```
search_type = "prod/asset"

# define a filter
filters = []
filters.append( ("asset_library", "set") )

# do the query
assets = my.server.query(search_type, filters)

print "found [%d] assets" % len(assets)

# go through the asset and print the code
for asset in assets:
    code = asset.get("code")
    print(code)
```

Executing this example will give the following output:

```
$ python examples/query.py
found [3] assets
chr001
chr002
chr003
```

In this example, a `search_type` is first defined. This search type is a uniquely named identifier for a class of `SObjects`.

A list of filters is next defined. These filters allow you to narrow the search to specific `SObjects`. In this example, only assets of the `asset_library = "set"` will be found.

Next, the assets are retrieved using the `query()` function, which returns a list where each element is a serialized dictionary of an `SObject`. In this example, the code for each asset is retrieved and printed.

Filters are very important in the `query` function because they narrow down searches to find the specific `SObjects` you are looking for. The filters are very flexible and support a wide range of different modes. A sample of the supported modes is shown below:

```
# simple search filter
filters = []
filters.append( ("name_first", "Joe") )
results = my.server.query(search_type, filters, columns)

# search with 'and': where name_first = 'Joe' and name_last = 'Smoe'
filters = []
filters.append( ("name_first", "Joe") )
filters.append( ("name_last", "Smoe") )
results = my.server.query(search_type, filters, columns)

# search with 'or': where code in ('joe','mary')
filters = []
filters.append( ("code", ("jo e", "mary")) )
results = my.server.query(search_type, filters, columns)

# search with 'or': where code in ('joe','mary') order by code
filters = []
filters.append( ("code", ("joe", "mary")) )
order_bys = ['name_first']
results = my.server.query(search_type, filters, columns, order_bys)

# search with like: where code like 'j%'
filters = []
filters.append( ("code", "like", "j%") )
```

```
results = my.server.query(search_type, filters, columns)

# search with regular expression: code ~ 'ma'
filters = []
filters.append( ("code", "~", "ma") )
results = my.server.query(search_type, filters, columns)

# search with regular expression: code !~ 'ma'
filters = []
filters.append( ("code", "!~", "ma") )
```

## Insert and Update

It is essential to insert SObjects and update their values.

The following code creates a new asset in the database.

```
# define a search type for which to add a new entry
search_type = 'prod/asset'

# build a data structure which is used as data for the new sobject
data = {
    'code': 'chr001',
    'name': 'Bob',
    'description': 'The Bob Character'
}

server.insert(search_type, data)
```

The following code snippet updates an existing asset in the database:

```
# define the search key we are searching for
search_type = "prod/asset"
code = 'vehicle001'
search_key = server.build_search_key(search_type, code)

# build a dataset of updated data
data = {
    'description': 'This is a new description'
}
# do the update
asset = my.server.update(search_key, data)

print asset.get("description")
```

Note that the search key is used to identify the precise sObject being updated. This search key uniquely identifies an sObject in TACTIC. With this search key, TACTIC is able to precisely update the correct sObject.

## Javascript Client API

The TACTIC Client API can be accessed in Javascript as well as Python. One can deduce its usage from the Python Client API doc. Here are some examples:

1. Using the eval() function, we want to find all the anim snapshots checked in with the asset chr001.

```
var server = TacticServerStub.get();
var exp = "@SOBJECT(sthpw/snapshot['context','anim'])";
var result = server.eval(exp, {search_keys:['prod/asset?project=sample3d&code=chr001']})
log.critical(result)
```

2. Display the notes written for the selected assets in the UI.

```
var server = TacticServerStub.get();
var search_keys = spt.table.get_selected_search_keys();
var exp = "@SOBJECT(sthpw/note)";
if (search_keys.length > 0){
    var result = server.eval(exp, {search_keys: search_keys});
    log.critical(result);
}
```

3. Display only the task code in anim or lgt process with description containing the word fire, not specific to any particular asset.

```
var server = TacticServerStub.get();
var exp = "@GET(sthpw/task['process', 'in', 'anim|lgt']['description','EQ','fire'].code)";
var result = server.eval(exp);
log.critical(result);
```

4. To insert a note for an asset chr001 under the model process and context.

```
var server = TacticServerStub.get();
var sk = server.build_search_key('prod/asset','chr001');
server.insert('sthpw/note', {'note': 'A test note', process: 'model', context: 'model', login:
    'admin'},
    {parent_key: sk});
```

# Checkin / Checkout Operations

## Checking files in

The Client API has access to the full range of TACTIC's asset management system.

Any SObject can become a "container" for checkins. This has the advantage that you can use this one SObject (container) to check in files using the deep set of checkin tools provided by TACTIC. The rest of this section describes the different types of checkins available.

## Simple Checkin

The `simple_checkin()` function allows you to check in a single file.

```
file_path = "../test/miso_ramen.jpg"

# now check in the file
search_type = "unittest/person"
code = "joe"
context = "test_checkin"
search_key = my.server.build_search_key(search_type, code)

# simple checkin of a file. No dependencies
desc = 'A Simple Checkin'
snapshot = my.server.simple_checkin(search_key, context, file_path, description=desc,
mode="upload")
print snapshot.get('snapshot')
```

The simple checkin is the most basic type of checkin. It creates a snapshot and then checks a file into that snapshot. The newly created snapshot is returned.

```
<snapshot>
<file name="miso_ramen_v001.jpg" type='main' code='123BAR' />
</snapshot>
```

The exact file name that is checked in will vary depending on the specific implemented naming conventions

## Group (or Sequence) Checkin

The `group_checkin()` function allows you to check in a sequence of files, defined by a frame range:

```
<start>-<end>/<by>
```

For example, a frame range of 1 to 10 is described as "1-10". Or every second frame from frame 20 to frame 50 can be described as "20-50/2".

TACTIC provides two notations to describe the file names of a range of frames. This special notation, in conjunction with the frame range, can generate a sequence of files. The two notations are as follows:

- `<base>####.<ext>`
- `<base>.%0.4d.<ext>`

Here is a code example of checking in a sequence of files:

```
pattern = "../test/miso_ramen.%0.4d.tif"
file_range = '1-24'
context = 'beauty '

# build the search key
search_type = "unittest/person"
```

```
code = "joe"
search_key = my.server.build_search_key(search_type, code)

# simple checkin of a file
desc = 'A Checkin of a group of files'
context = "test_checkin"
snapshot = server.group_checkin(search_key, context, file_pattern, file_range)
print snapshot.get('snapshot')
```

When executed, this example will check in a sequence of 24 files starting from 1 to 24. It should be noted that this method will by default expect that the files have been uploaded to the server. For this reason, it is often recommended to use preallocated checkins for both sequence and directory checkins.

## Directory Checkin

As the name suggests, a directory checkin enables an entire directory and all of its subdirectories to be checked in. TACTIC does not keep track of the contents of the checked-in directory. This allows you to check in complex directory structures without having to inform TACTIC of all of the details of the contents. This might be the best approach when all the details of the directory are already handled by some other system so it is not necessary for TACTIC to track things.

Here is a code example of checking in a directory:

```
file_path = "./test/XG002/beauty"

# build the search key
search_type = "unittest/person"
code = "joe"
search_key = my.server.build_search_key(search_type, code)
context = "test_checkin"

# simple checkin of a file.
desc = 'A Simple Checkin'
snapshot = my.server.directory_checkin(search_key, context, file_path, description=desc)
print snapshot.get('snapshot')
```

Note that this code is very similar to single file checkins ( `simple_checkin()` ), because TACTIC treats a directory checkin in a similar manner to a file checkin. It uses the leaf directory as the file name. It is important to consider naming conventions, because this leaf directory will be handled using file naming conventions even though it is a directory.

As with `group_checkin()`, this method already expects the files to have been uploaded to the server in the appropriate place. There are various modes that can be used to alter the manner in which the files get to the server repository. For details, see the "modes" section below.

## Piecewise checkins

TACTIC allows you to build up a checkin piecewise, in stages. This is a powerful feature because you can build a checkin over the course of many operations (and many transactions if desired) and the whole set of operations will be treated as a single versioned entity. The TACTIC snapshot definition allows for the entry of multiple files into a single checkin. Typically, the process begins by creating a new "empty" snapshot. This snapshot is a placeholder which reserves a version and context for a particular set of future operations. Once this empty snapshot is created, you can start adding files and dependencies to it.

The following example checks in a Maya file and a corresponding OBJ file.

```
maya_path = "./test/chr001/chr001_model.ma"
obj_path = "./test/chr001/chr001_mode.obj"

# build the search key
```

```

search_type = "unittest/person"
code = "joe"
context = "test_checkin"
search_key = my.server.build_search_key(search_type, code)

# create an empty snapshot
desc = 'A Piecewise Checkin'
snapshot = my.server.create_snapshot(search_key, context, description=desc)
print "empty"
print snapshot.get('snapshot')

snapshot_code = snapshot.get('code')
snapshot = my.server.add_file(snapshot_code, maya_path, file_type='maya')
snapshot = my.server.add_file(snapshot_code, obj_path, file_type='obj')
print
print "two files"
print snapshot.get('snapshot')

```

Executing this code will result in the following:

```

empty
<snapshot/>

two files
<snapshot>
  <file name='chr001_model_v001.ma' file_code='1044BAR' type='maya' />
  <file name='chr001_model_v001.obj' file_code='1045BAR' type='obj' />
</snapshot>

```

First, an empty snapshot is created using `create_snapshot()`, then files are added to this snapshot one by one. Note that the type here is explicitly specified. This type differentiates one file in a snapshot from another.

It is also possible to add a sequence of files or even a directory to a snapshot:

```

pattern = "./test/miso_ramen.%0.4d.tif"
file_range = '1-24'
snapshot = server.add_group(snapshot_code, file_pattern, file_range, file_type='sequence')
print snapshot.get('snapshot')

directory = "./test/test_directory"
snapshot = server.add_directory(snapshot_code, directory, file_type='directory')
print snapshot.get('code')

```

Executing the last code snippet will give the following results:

```

<snapshot>
  <file name="mise_ramen.%0.4d.tif" file_code='1047BAR' type='sequence' />
</snapshot>

<snapshot>
  <file name="mise_ramen.%0.4d.tif" file_code='1047BAR' type='sequence' />
  <file name="test_directory" file_code='1047BAR' type='directory' />
</snapshot>

```

## Checkin Modes

There are various modes that you can use to check in files. These modes determine how a file will be transferred to the repository.

- **upload:** Uploads the files to a temporary directory
- **copy:** Copies the files to the handoff directory
- **move:** Moves the files to the handoff directory.

The previous `simple_checkin()` example uses the "upload" mode. This means that the client will connect to the server and use an HTTP connection to upload the file to the server where it will be subsequently checked in. HTTP does not require any additional setup and it may be the only choice available for facilities having only WAN access to the TACTIC server. However, HTTP is a very slow transport protocol so, if possible, it is better and faster to use other available modes.

The copy and move modes use a "handoff" directory, which is an intermediate directory that is visible on the network to both the client machine and the TACTIC server. When the checkin is executed, the files are first copied or moved to this handoff directory. The TACTIC server is then notified and grabs the files and puts them into the repository, renaming as the naming conventions stipulate. The files are always "moved" from the handoff directory to the repository. The advantage of using these modes over the "upload" mode is that they go through NFS or CIFS. These modes make use of the fast networks and huge file servers that are available in typical media and production facilities.

The copy and modes require a bit of setup because the server and the client must be able to see the handoff directory. You need to configure the TACTIC server configuration file, located in `<site_dir>/config/tactic_<os>-conf.xml`. This file contains the following relevant settings:

- `win32_client_handoff_dir`: the handoff directory as seen from a Windows client
- `linux_client_handoff_dir`: the handoff directory as seen from a Linux client
- `win32_server_handoff_dir`: the handoff directory as seen from a Windows TACTIC server
- `linux_server_handoff_dir`: the handoff directory as seen from a Linux server

Note that the win32 settings apply to all flavors of Windows, including Windows 64-bit machines. The Linux settings apply to all POSIX machines including Debian base operating systems and Mac OS X.

After you set the configuration, you can then use the copy or move modes to take advantage of the handoff directory:

```
# simple checkin of a file using move mode
desc = 'A Simple Checkin'
snapshot = my.server.simple_checkin(search_key, context, file_path, description=desc,
mode="move")
print snapshot.get('snapshot')
```

Note that the only difference in this example from earlier checkin examples is that the mode parameter is set to "move".

## Preallocated checkin (mode="preallocate")

Preallocated checkins are the most efficient checkins. Bandwidth and storage space are expensive commodities in a typical media or production facility, so there is a definite cost and time benefit to reducing their use as much as possible.

Preallocated checkins enable a client process to be checked directly into the repository. They are recommended for checkins that are very heavy in either bandwidth or disk usage and are designed to minimize both. Some production processes that would benefit from using this checkin mode include rendering frames, ingesting plates, simulating data, and so on.

The following steps describe the process for preallocating checkins:

1. Create an empty snapshot to reserve a checkin version and context.
2. Ask for a path in the repository from the TACTIC server.
3. Create the files directly in the path given by the TACTIC server.

#### 4. Inform TACTIC that the files have been placed in the appropriate location.

The path supplied by TACTIC in the preallocation is located directly in the repository. The process generating the files can thus save the files directly to the correct location in the repository (following all the predefined naming conventions). Files are created directly in the repository with the correct directory and file name as TACTIC would have checked them in using the other methods. This eliminates later having to copy or move files around the network unnecessarily, as is typically required by other checkin modes.

Because the `simple_checkin()`, `group_checkin()` and `directory_checkin()` functions perform the entire checkin process in one step, you cannot use them for preallocated checkins. Instead, you would use a piecewise checkin to build up the checked in parts. The following is an example of a preallocated checkin using a piecewise approach:

```
search_type = "prod/render"
code = "XG002_beauty"
search_key = my.server.build_search_key(search_type, code)

# create an empty snapshot
desc = 'A Preallocated Checkin'
context = "render"
snapshot = my.server.create_snapshot(search_key, context, description=desc)

# get the preallocated path
snapshot_code = snapshot.get('code')
file_pattern = snapshot.get_preallocated_path(snapshot_code, file_type="main")
print "file_pattern: ", file_path

# generate the files
for i in range(1, 20):
    file_path = file_pattern % i
    render_file(file_path)

# add the files to the snapshot
snapshot = server.add_group(snapshot_code, file_type="main", file_range="1-20",
mode="preallocate")
print snapshot.get("snapshot")
```

Executing the above code would result in output something like:

```
file_pattern: XG002_beauty_v012.%0.4d.tif
<snapshot>
  <file name="XG002_beauty_v012.%0.4d.tif" file_code="123BAR" type="main"/>
</snapshot>
```

The file pattern returned is completely dependent on naming conventions. In this case, the `search_type` would have had to define a naming convention whereby the context of "render" produces the above file pattern. For example, the file naming convention code could include:

```
def prod_render(my):
    render = my.sobject
    ext = my.get_file_ext()

    parts = []
    parts.append( render.get_value('code') )
    parts.append( "v%0.3d" % my.snapshot.get_value("version") )

    file_name = "_".join(parts) + ".%0.4d" + ext
    return file_name
```

(See the naming convention documentation for more information on how to set up naming conventions.)

It should be noted that the function `get_preallocated_path()` returns a full path, including the filename as specified by the naming conventions. Ideally, TACTIC must be able to generate the correct path that can be used to save the files (as in the example above).

There is enormous advantage to using preallocated checkins. Files are created directly to the repository, eliminating all of the unnecessary copying of files around the servers. When groups of files reach the multi-gigabyte or even terabyte range, it becomes prohibitively expensive to check in files in the traditional manner. Preallocated checkins maximize the use of your internal system architecture.

## **In-Place Checkins**

In general, the in-place checkin should be considered as the last resort. In-place checkins do not make use of the TACTIC naming conventions, and may be the only option when you are confronted by a legacy directory structure. Using this checkin method makes the assumption that you will be able to later define logic that will map to a desired naming convention. As a guideline, naming conventions should be procedural and as simple as possible, so you must plan carefully before considering in-place checkins.

# Snapshot Dependency

## Types of dependencies

Snapshots control versioning in TACTIC. When processing a checkin, TACTIC creates a snapshot that contains an XML description of what was checked in. Snapshots can also be dependent on any number of other snapshots (through a "ref" tag). Taking advantage of this dependency relationship, you can create complex dependency trees for complex scenes, with the option of undoing them if required.

There are two types of dependencies:

- hierarchical: The given snapshot contains the referenced snapshot
- input: The given snapshot used or was created from a referenced snapshot (but does not contain the contents of that snapshot)

## Connecting snapshots

Dependencies are connected using the `add_dependency_by_code()` method, which takes an existing snapshot and adds the appropriate reference tag to it.

The following example shows how to connect two snapshots:

```
search_type = "prod/asset"
code = "chr001"
search_key = server.build_search_key(search_type, code)

# checkin a model
model_snapshot = server.simple_checkin(search_key, model_path, context="model")
model_snapshot_code = model_snapshot.get('code')

# checkin a rig
rig_snapshot = server.simple_checkin(search_key, rig_path, context="rig")
rig_snapshot_code = rig_snapshot.get('code')

# add the model dependency to the rig
snapshot = server.add_dependency_by_code(rig_snapshot_code, model_snapshot_code)
print snapshot.get('snapshot')
```

Executing the above example would output:

```
<snapshot>
  <file name="chr001_rig_v001.ma" file_code="123BAR" type='main' />
  <ref context='model' version='3' search_type='prod/asset?project=sample3d' search_id='4' />
</snapshot>
```

The ref tag is the reference to another checkin. In this case, the reference can be interpreted as being contained in the snapshot (that is, this is a hierarchical dependency).

Sometimes, it is not possible to store or retrieve version information for an SObject within a session if a particular application provides only the filename. It is generally assumed that a filename is unique for each search\_type in each project (this is not strictly enforced, but should be as best practice), so it is possible to reverse-map a filename to a snapshot. In this case, you can try to add a dependency using the `add_dependency()` method:

```
file_path = extract_dependent_path()
snapshot = server.add_dependency(snapshot_code, file_path)
```

This method will attempt to link the filename with the appropriate snapshot.

## Input references

As opposed to the previous example of hierarchical references, there is a second type of dependency called an input reference. Input references are dependencies where a particular snapshot was used to produce another snapshot, but the resulting snapshot does not contain the contents of the originating snapshot. As an example, a Photoshop file may be used to generate a texture map, but the texture map does not need to contain the Photoshop file.

Adding an input reference is simply a matter of setting the "type" argument to "input\_ref":

```
source_path = "../test/texture.psd"
image_path = "../test/texture.tif"

# check in the photoshop file
source_snapshot = server.simple_checkin( search_key, context="source", file_path=source_path )
source_snapshot_code = source_snapshot.get('code')
source_repo_path = server.get_path_from_snapshot( source_snapshot_code )

# checkin the image
image_snapshot = server.simple_checkin( search_key, context="image", file_path=image_path )

# add an input dependency
image_snapshot_code = image_snapshot.get('code')
image_snapshot = server.add_dependency( image_snapshot_code, source_repo_path, type="input_ref" )
print snapshot.get('snapshot')
```

The above code would produce output like the following:

```
<snapshot>
  <file name="texture_image_v001.tif" file_code="123BAR" type='main' />
  <ref context='source' version='3' search_type='prod/asset?project=sample3d' search_id='4'
type="input_ref" />
</snapshot>
```

By managing dependencies at the time of each checkin, it is possible to build up a dependency tree. Thus each version of every checkin has its own independent dependency tree.

# Widgets

## TACTIC Widgets

### What are Widgets?

Widgets are drawable entities. They have the ability to draw themselves and also have the ability to contain other widgets and call on their drawing.

### Widget Architecture?

The TACTIC interface is entirely built on top of widget architecture. A widget has a drawing mechanism which displays the widget. Widgets can contain any number of other widgets and pass information to them.

Certain widgets also make use of configuration xml documents in order to configure how they should be drawn. These configs are useful because they allow very quick and readable configuration of complex widgets. This document can also be stored in the database as a way of remembering the state of how to redraw a particular widget. This is widely used in TACTIC to store various parts of the interface in the database.

Every widget has a display method which completely controls how a widget is displayed. This display is recursive as each widget will call all of it's children's display method. In this manner, the entire interface is build up.

Widgets derive data to draw from subjects. Generally a search is performed to retrieve subjects which are then used to draw the widget. The widget itself can perform the search or it can recieve subjects from some external source.

### Widget Config

Numerous widgets use configuration xml documents to help them draw their display. These widgets are considered to be "layout" widgets in that they generally use the configurations to determine what the child widgets are and how and where they are drawn within the parent layout widget. The widget config is an xml document which describes the child elements and how they should be display. The format is defined as follows.

```
<config>
  <VIEW>
    <element name='NAME' OPTION='VALUE'>
      <display class='CLASS_PATH'>
        <KWARG>VALUE</KWARG>
        <KWARG>VALUE</KWARG>
      </dispaly>
    </element>
    <element name='NAME' OPTION='VALUE'>
      <display class='CLASS_PATH'>
        <KWARG>VALUE</KWARG>
        <KWARG>VALUE</KWARG>
      </dispaly>
    </element>
  </VIEW>
</config>
```

Where capitalized words represent variable entries.

VIEW	The name of a view which encompasses a particular configuration. There can be any number of views in a configuration documentation
OPTION	An option defining a state or setting of this element. This information does not get passed to the element widget

VALUE	A value or a particular argument or options
CLASS_PATH	The fully qualified python path of the widget class
KWARG	A kwarg that is passed to the class on construction

A simple example of a configuration is as follows:

```
<config>
<simple>
  <element name='email'>
    <display class='custom.MyCustomWdg'>
      <title>My Widget</title>
    </display>
  </element>
</simple>
</config>
```

In this case, the "simple" view defines a single element called "email". This element

The configuration document can contain any number of "views". Each "view" can contain any number of elements. Inside each element, there are xml snippets which represents an xml serialization of a widget. In the example above:

```
<display class='custom.MyCustomWdg'>
  <title>My Widget</title>
</display>
```

translates into python server code as follows:

```
from custom import MyCustomWdg
widget = MyCustomWdg(title='My Widget')
```

TACTIC uses this format extensively to serialize widgets to the database. Although any source can be used, the config is most often defined in the widget config table of a particular project.

There are a couple of layout classes that make heavy use of the widget config.

### SideBarWdg:

**TableLayoutWdg:** this class is the used to display most tabular data in TACTIC. It contains many features to make the display of tabular data dynamic and flexible. Views can be customized and saved. It is probably the most used layout class in TACTIC. It makes heavy use of the widget config for its display. It's importance is sufficient to warrant a section on its own below.

**CustomLayoutWdg:** this class makes use of a special version of the config. It defines elements, but they are defined within an html tag, allowing for precise layout of elements using HTML. This allows for very flexible layouts while still being able make use of TACTIC widgets.

## SideBarWdg

The SideBarWdg defines the look of the side bar on the left of the TACTIC interface. The SideBarWdg makes heavy use of the widget config to determine the contents of the side bar. There are 3 main types of widgets that would be defined as elements in the SideBarWdg:

- LinkWdg
- FolderWdg (Currently SectionWdg)
- SeparatorWdg

The top level view for the project views can be found in the widget config table with the criteria:

- search\_type = 'SideBarWdg'
- view = 'project\_view'

This will defined a list of elements that appear in the top level of the "Project View". An example would look like the following:

```
<config>
  <project_view>
    <element name='summary' />
    <element name='modeling' />
  </project_view>
</config>
```

Although, you could defined the display section here, there are hierarchical definitions to the elements. If a definition is not found inline, TACTIC will look at the the database for the specially named "definition" view.

- search\_type = 'SideBarWdg'
- view = 'definition'

```
<config>
  <definition>
    <element name='summary' title='Asset Summary'>
      <display class='LinkWdg'>
        <class_name>tactic.ui.panel.ViewPanelWdg</class_name>
        <search_type>prod/asset</search_type>
        <view>summary</view>
      </display>
    </element>
    <element name='modeling' title='Modelling'>
      <display class='FolderWdg'>
        <view>modeling</view>
      </display>
    </element>
  </definition>
</config>
```

Both the summary and modeling elements are defined in this special "definition" view"

Since all of the folders at all levels cascade to look at the "definition" view, it is useful to always define defintions of elements in the "definiton" view. This will allow a consistent definition for all of the "views" in the project view.

The "summary" view is defined as a LinkWdg. This widget takes the information defined in the options and then displays that class in the main body of the TACTIC interface.

```
widget = ViewPanelWdg( search_type='prod/asset', view='summary' )
```

As stated ealier, the ViewPanelWdg, combines a SearchWdg with a TableLayoutWdg.

The second element defines a "modeling" folder. Whe a folder is click, it will open up and display another list that is derived from the "modeling" view.

## TableLayoutWdg

This widget is the primary class used in TACTIC to lay out tabular data. It makes heavy use of widget config to define what to display.

To display the rows and columns of the tabular layout, this widget makes use of the following:

- a) rows which are subjects

b) columns which are widgets derived from BaseTableElementWdg.

The table layout widget is able to perform a search base on input criteria. It is also able to receive sobjects through its set\_objects() method.

This widget iterates through each of the sobjects per row.

For each column, the table draws the list of widgets provided by the config. This config is typically defined in in the database in the widget config table.

Two parameters are typcially used to find a particular widget config.

a) Search Type

b) View

## **BaseTableElementWdg**

BaseTableElementWdg are extensively used in the UI. Each column in a table you see in TACTIC derives from it. For examples of how to create your own, please refer to the Widget Development section.

# Custom Interfaces

## Introduction

Although any execution environment can interact with TACTIC by interfacing through the Client API, most often, users will be interacting with TACTIC through the browser. TACTIC's main interface is the browser. All browsers come with the Javascript language interpreter built-in and thus any rich interface that integrates with TACTIC will need to interact with the various components using Javascript.

Three core frameworks in TACTIC work together to create a rich web interface.

- CustomLayoutWdg: provides the ability to create the visual interface by laying out widgets using html templating
- Behaviors: provides a framework to create complex behaviors that is much easier to use than the browsers default event system.
- Applet: provides the interaction to the client machine to do operations that the browser would otherwise not be permitted to do

## Accessing the server from Javascript

The TACTIC Client API can access server functionality through the TacticServerStub in the same manner as its Python equivalent. Note the similarities

Python code:

```
server = TacticServerStub.get()
snapshot = server.checkin(search_key, context, path, mode="upload")
print snapshot.get("code")
```

Javascript code:

```
var server = TacticServerStub.get();
var snapshot = server.checkin(search_key, context, path, {mode: "upload"} );
alert(snapshot.code)
```

There are a few differences due to the syntax of the two different languages. Keyword arguments are not natively supported by Javascript. Since some of the functions in the server stub have numerous arguments, it is desirable to only use those that are needed without having to "fill in" all of the preceding arguments with nulls.

For example, the previous javascript code would have to read like the above:

```
server.checkin(search_key, context, path, null, null, null, null, "upload")
```

In general, a given function will have a few necessary arguments and all "optional" arguments are given in a kwargs dictionary. Another difference is that the subjects returned are javascript "objects" whose members are values from the database. Attributes can be accessed in two ways:

1. code = snapshot['code']
2. code = snapshot.code

## Testing Javascript

The most convenient method to test and implement the javascript examples is in the Javascript Editor. This can be convenient accesses by pressing the "9" hot key to bring it up. Alternatively, the Javascript Editor can be brought up under the gear menu under Tools->JavaScript Editor.

## The CustomLayoutWdg:

This is a simple "Hello World" example.

```
<html>
  <h1>Hello World</h1>
</html>
```

The xml document embeds an html tag that will be used to layout elements in the application.

## Example 01: Hello World

The simplest way to view this is to open up the TACTIC javascript editor and type the following code in:

```
var html = "<html><h1>Hello World</h1></html.>";
var kwargs = {
  'html': html
};
spt.panel.load_popup('Hello', 'tactic.ui.panel.CustomLayoutWdg', kwargs);

// NOTE: this should be:
// spt.api.load_popup('Hello', 'tactic.ui.panel.CustomLayoutWdg', kwargs);
```

This previous code is completely in javascript, however, layout pages using strings in javascript rapidly becomes unwieldy. It is thus preferential to create these layouts using the widget config. This is done by going to the side bar and clicking on "Project Admin->Widget Config". This will open up the "widget\_config" table. This table is used to store all custom interface configurations for widgets.

Create a new entry by pressing the [+] button on the right side.

```
<config>
<example01>
<html>
  <h1>Hello World</h1>
</html>
</example01>
</config>
```

This is the full xml document describing the widget config. Note that the html is now embedded within that xml document. This will be important to know later when behaviors and elements are added to the widget.

Finally, in the JS Editor, enter the following:

```
kwargs = {
  view: 'example01'
};
spt.panel.load_popup('Example01', 'tactic.ui.panel.CustomLayoutWdg', kwargs);
```

You should see the following when you click on "Run"

<IMAGE>

## Example 02: Adding to button with a behavior

Add a new entry to the widget\_config table with view = 'example02' and with the following config definition.

```
<config>
<example02>
<html>
  <span>This is a button:</span>
  <input type='button' class='button1' value='Press Me' />
</html>
<behavior class='button1'>{
  "type": "click_up",
  "cbjs_action": ''
    alert('Hello World');
  ''
}</behavior>
</example02>
</config>
```

In this example, an html button is added to the html layout. By default, a button doesn't do anything when it is clicked. A behavior has to be added for something to happen. TACTIC behaviors are added to DOM elements by their class attributes.

When the button is clicked (corresponding to the "click\_up" event type), the javascript in the "cbjs\_action" attribute is executed. This example will alert a "Hello World" message on clicking.

## Example 03 – Using form value

The following example will add a text area to the interface as well as extract information from that text area once the button has been clicked.

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- This examples displays some html UI and then reacts to it using the TACTIC
      behavior system -->
<config>
<example03>
<html>
<div class='spt_top'>
  <textarea name='description' class='spt_input'></textarea>
  <input type='button' class='spt_button1' value='Press Me' />
</div>
</html>

<behavior class='spt_button1'>{
  "type": "click_up",
  "cbjs_action": ''
    var top = bvr.src_el.getParent('.spt_top');
    var values = spt.api.Utility.get_input_values(top);
    var description = values.description;
    alert('You entered: ' + description);
  ''
}</behavior>

</example03>
</config>
```

Note that currently, `get_input_values()` requires that every input element have `class='spt_input'` attribute. Future version may remove this requirement, but this is currently necessary.

Please note that when an API for 2.6/2.7, the following lines will be changed:

```
The following line:
var values = spt.api.Utility.get_input_values(top);
will be replace by:
var values = spt.api.get_input_values(top);

The following line:
var top = bvr.src_el.getParent('.spt_top');
will be replaced by:
var top = spt.api.get_parent(bvr.src_el, ".spt_top");
```

The behavior definition warrants a closer examination:

```
<behavior class='spt_button1'>{
  "type": "click_up",
  "cbjs_action": ''
  var top = bvr.src_el.getParent('.spt_top');
  var values = spt.api.Utility.get_input_values(top);
  var description = values.description;
  alert('You entered: ' + description);
  ''
}</behavior>
```

First, there is an implied `bvr` object that exists in the namespace of the behavior. This `bvr` objects contains useful data for the purposes of executing behaviors. The most important attribute is `"bvr.src_el"`. This element is source element that called the event. This element can be used as a starting point to navigate the DOM to search for elements.

```
var top = bvr.src_el.getParent('.spt_top');
```

It is common practice to find a top level element of a widget from the source element. This top element is a starting point from which searches under a DOM hierarchy can be made. By starting from a top element, it is ensured that the returned values are isolated to that single widget.

The next line gets all of the values of all of the input elements under the top element.

```
var values = spt.api.Utility.get_input_values(top);
```

This returns a dictionary of name/value pairs of all of the input elements underneath the top element.

## Example 04 - Adding Expressions

By adding expressions to a report, it becomes very easy to create reports that extract important information and combine it into a single view.

```

<?xml version='1.0' encoding='UTF-8'?>
<config>
<example04>
<html>
  <h1>My login is [expr]$LOGIN[expr]</h1>
  <table>
    <tr><td>Number of tasks</td><td>[expr]@COUNT(sthpw/task)[/expr]</td></tr>
    <tr><td>Number of checkins</td><td>[expr]@COUNT(sthpw/snapshot)[/expr]</td></tr>
    <tr><td>Number of model checkins</td>
      <td>[expr]@COUNT(sthpw/snapshot['context','model'])[/expr]</td>
    </tr>
  </table>
</html>
</example04>
</config>

```

Expression can be added into the html code by inserting it between [expr][/]expr tags. The expression will be evaluated and the result will be replaced into the html. This provides an ability to layout an arbitrary layout in javascript and then fill in the missing data with expressions. The full power of the TACTIC expression language is available. Please refer to the expression language reference for more information on the expression language.

## Example 05: Mako integration (2.6.0+)

The CustomLayoutWdg can make use of the Mako templating engine to create dynamic content. Mako is a powerful templating system similar in concept to PHP, but instead uses the Python programming language. The expression language on its own is quite powerful, but it is still an expression language and sometimes, it is necessary to have full programming logic. Mako provides a path to create content that is too complex for the expression language to handle alone.

The following example shows a report generated with the help of Mako:

```

<?xml version='1.0' encoding='UTF-8'?>
<!-- Simple test using mako templating -->
<config>
<example06 include_mako='true'>
<html>
<div>
<![CDATA[
<%
  # get some data
  total = 0
  for ctx in ['model', 'texture', 'rig']:
    num_snapshots = server.eval("@COUNT(sthpw/snapshot['context','%s'])" % ctx)
    context.write("Number of %s checkins: %s<br/>" % (ctx, num_snapshots) )
    total += num_snapshots
  %>
  Total number of tasks: ${total}<br/>

]]>
</div>
</html>
</example06>
</config>

```

Mako is not enabled by default. This must be done with the "include\_mako" attribute:

```

<example06 include_mako='true'>

```

All code between `<%` and `%>` tags are parsed as python code and executed on the server. In order to write out to the html, Mako uses the `context.write()` method. This is important to note because the "context" is a reserved word in Mako. This can cause a confusing error because context is a common variable name when programming in TACTIC.

```
context.write("Number of %s checkins: %s<br/>" % (ctx, num_snapshots) )
```

The python code with the python block can still make use of the entire TACTIC Client API through the use of a builtin variable "server". This also means that expressions can be accessed here as well:

```
num_snapshots = server.eval( "@COUNT(sthgw/snapshot['context','%s'])" % ctx)
```

Also note that the entire Mako code is wrapped around an XML CDATA block ( `<![CDATA[ ... ]]>` ). This is because python code very easily breaks XML integrity rules. The CDATA block allows for any special characters to be entered in the XML document. It is good practice to add the CDATA tags in order to avoid errors later on.

Any variables that are declared in python blocks can be accessed outside of the python blocks using the `${var}` syntax. The following will replace `${total}` with the corresponding variable defined in the python block.

```
Total number of tasks: ${total}<br/>
```

Combining the expression language with Mako Templating provides unlimited flexibility in creating complex reports.

## Example 07 - Using a CustomLayoutWdg inside of a TableLayoutWdg

The CustomLayoutWdg can be used inside of a table element. This makes it easy to create arbitrarily complex table elements within a standard TACTIC table layout widget. The following displays the number of tasks for the row subject.

```
<config>
<my_view>
<element name='num_tasks'>
  <display class='tactic.ui.panel.CustomLayoutWdg'>
    <html>
      <div class='top'>
        <b>[expr]@COUNT(sthgw/task)[/expr] tasks</b>
      </div>
    </html>
    <behavior>{
      'type': 'load',
      'cbjs_action': ''
      var search_key = bvr.kwargs.search_key;
      alert(search_key)
    }
  </behavior>
</display>
</element>
</my_view>
</config>
```

This element behaves just like the previous CustomLayoutWdg, however there are a few additions. There is a starting subject that corresponds to the table row that is passed in and is used as the starting subject for all expressions. The following expression finds the number of tasks for the subject in question and not all of the tasks in the system.

```
<b>[expr]@COUNT(sthpw/task)[/expr] tasks</b>
```

Another addition is that callbacks have the search key of the subject for the row available through the bvr object passed into the behavior callback.

```
var search_key = bvr.kwargs.search_key;
```

With the search key, it becomes possible to use the client API to change data or checkin files for that specific subject.

## Example - Connecting to the server from Javascript

It is often necessary to be able to interact with the server using Javascript in a behavior callback. This is done using the Javascript implementation of the TACTIC Client API

The following example illustrates how to interact with the server using the TacticServerStub object. This object is used to issue commands that will be run on the server such as updating data in the database or checking in files.

First, add any image in "C:/Temp/test.jpg"

```
<?xml version='1.0' encoding='UTF-8'?>
<config>
<example04>
<html>
<div class='spt_top'>
  <textarea name='description' class='spt_input'></textarea>
  <input type='button' class='spt_button1' value='Press Me' />
</div>
</html>

<behavior class='spt_button1'>{
  "type": "click_up",
  "cbjs_action": ''
  var top = bvr.src_el.getParent('.spt_top');
  var values = spt.api.Utility.get_input_values(top);
  var description = values.description;

  var applet = spt.Applet.get();
  var paths = applet.open_file_browser("C:/Temp");
  var path = paths[0];

  var search_key = bvr.kwargs.search_key();

  var server = TacticServerStub.get();
  server.checkin(search_key, "icon", path, {description: description});
  ''
}</behavior>

</example04>
</config>
```

The applet is used to interact with the client machine. It defines a number of useful methods such as listing directories, moving and copying files, uploading and downloading files. For a complete list of the functionality present in the

applet, please refer to the Applet Reference manual. In this case, the example is using the applet to open up a file browser so the user can select a file.

```
var applet = spt.Applet.get();
var paths = applet.open_file_browser("C:/Temp");
var path = paths[0];
```

The search key can be obtained from the behavior. This will be required to check into the correct object.

```
var search_key = bvr.kwargs.search_key();
```

Once a file path has been selected, the server stub is used to check in the file to the server.

```
var server = TacticServerStub.get();
server.checkin(search_key, "icon", path, {description: description});
```

## Example 10 - Converting to a button

Generally, it is not desirable to show a full interface for the checking directly in the table cell. It is much cleaner to have a simple publish button that will open up the interface in a popup.

## Example 11 - Integrating Server Side widgets

Many widgets are defined on the server side. These can be integrated in a custom interface by using the TACTIC specific <element> tag in the html definition of a CustomLayoutWdg.

```
<config>
<example11>
<html>
  <h1>This is a list of users</h1>
  <element name='users' />
</html>
<element name='users'>
  <display class='tactic.ui.panel.TableLayoutWdg'>
    <search_type>sthpw/login</search_type>
    <view>table</view>
  </display>
</element>
</example11>
</config>
```

# Expression Language

## Using Expressions in Scripting

### Using Expressions in Python - Server code

Expressions can be accessed directly through Python code. The expression language is often very convenient to quickly perform relatively complex searches quickly and easily.

To access the expressions in Python, you would use the following code:

```
from pyasm.biz import ExpressionParser
parser = ExpressionParser()
expr = "@GET(prod/shot['code','chr001'].prod/shot_instance.prod/asset.code)"
result = parser.eval(expr)
```

It is often more convenient just to access it through the Search module:

```
from pyasm.search import Search
expr = "@GET(prod/shot['code','chr001'].prod/shot_instance.prod/asset.code)"
result = Search.eval(expr)
```

## Using Expressions in Python - Client API code

To access the expressions in the Python Client API, you would use the following code:

```
server = TacticServerStub.get()
expr = "@GET(prod/shot['code','chr001'].prod/shot_instance.prod/asset.code)"
result = server.eval(expr)
```

When the expression language returns objects, these will be in the form of a dictionary like all other subjects in the client API.

## Using Expressions in Javascript - Client API code

To access the expressions in the Javascript Client API, you would use the following code:

```
var server = TacticServerStub.get()
expr = "@GET(prod/shot['code','chr001'].prod/shot_instance.prod/asset.code)"
var result = server.eval(expr)
```

## Using Expressions in Widget Config

The main widget to use expressions is "tactic.ui.table.ExpressionElementWdg".

When using the ExpressionElementWdg, the starting point of the expression is automatically the SObject associated with the row. This allows you to use the shorthand form without having to filter.

```
<element name='code'>
  <display class='tactic.ui.table.ExpressionElementWdg'>
    <expression>@GET(.code)</expression>
  </display>
</element>
```

## Using Expressions inline in HTML

When using the CustomLayoutWdg, inline expressions are supported using a [expr]/[expr] tag formatting.

```
<div>
  <h2>There are [expr]@COUNT(prod/asset['asset_library', 'chr'])[/expr] Characters</h2>
</div>
```

## Using Expressions in CustomLayoutWdg

The custom layout widget has a special html tag which can have html embedded within it. CustomLayoutWdg provides the ability to embed expressions within its html definition.

The following demonstrates a widget config using expressions:

```
<?xml version='1.0' encoding='UTF-8'?>
<config>
<example>
<html>
  <table>
    <tr><td>[expr]$LOGIN[/expr]</td></tr>
    <tr><td>[expr]{@GET(.code)} : {@GET(.description)}[/expr]</td></tr>
  </table>
</html>
</example>
</config>
```

Please refer to the CustomLayoutWdg in the Widget Reference documentation for more information on how to use the CustomLayoutWdg.

# Validation

## Validation Set-up

To limit what a user can enter in a field, you can set up validation for the column. It is particularly useful when the user is required to type in a text field instead of a selection list. This works on the client side so it activates before you click on the save button.

Example 1: Ensure the field description of prod/shot starts with the word "Client"

In the edit view of prod/shot, make sure there is an element for description defined with these display options:

```
<element name='description'>
  <display class='TextWdg'>
    <validation_js>return value.test(/^Client/)</validation_js>
    <validation_warning>It needs to start with Client</validation_warning>
  </display>
</element>
```

If the person types in something, press Enter and it fails the validation, the text field will turn red. You can view the warning message when the mouse pointer is over the text field. The variable 'value' is assumed to be value the user types in.

Example 2: Ensure the field description of prod/shot contains the code in the same row. The assumption is that the user would pick a show code in the previous column before typing in a description.

In the edit view prod/shot, make sure there is an element for description defined with these display options:

```
<element name='description'>
  <display class='TextWdg'>
    <validation_script>validate_desc</validation_script>
    <validation_warning>It needs to contain the shot code</validation_warning>
  </display>
</element>
```

The script it refers to is a javascript saved in the Script Editor. It has a code equal to 'validate\_desc'.

```
// value, display_target_el, and bvr are assumed variables
var row = display_target_el.getParent('.spt_table_tr');
var td = row.getElement('td[spt_element_name=shot_code]');
var shot_code = td.getAttribute('spt_input_value');
var exp = new RegExp(shot_code);
if (!shot_code) {
  return false;
}
if (value.test(exp)) {
  return true;
}
else {
  return false;
}
```

Like 'value', 'display\_target\_el' and 'bvr' are assumed variables.. The former represents the html element holding the value whereas the latter is the behavior object.

# TACTIC Python Client API Reference

## **\_\_init\_\_**

**\_\_init\_\_(login=None, setup=True, protocol=None, server=None, project=None, ticket=None, user=None, password='')**

Initialize the TacticServerStub

### **keyparam:**

**login** - login\_code

**setup** - if set to True, it runs the protocol set-up

**protocol** - xmlrpc or local. it defaults to xmlrpc

**server** - tactic server

**project** - targeted project

**ticket** - login ticket key

**user** - tactic login\_code that overrides the login

**password** - password for login

## abort

**abort(ignore\_files=False)**

Abort the transaction. This undos all commands that occurred from the beginning of the transactions

### keyparam:

**ignore\_files: (boolean)** - determines if any files moved into the repository are left as is. This is useful for very long processes where it is desirable to keep the files in the repository even on abort.

### example:

A full transaction inserting 10 shots. If an error occurs, all 10 inserts will be aborted.

```
server.start('Start adding shots')
try:
    for i in range(0,10):
        server.insert("prod/shot", { 'code': 'XG%0.3d'%i } )
except:
    server.abort()
else:
    server.finish("10 shots added")
```

## add\_config\_element

```
add_config_element(search_type, view, name, class_name=None, display_options={},
action_class_name=None, action_options={}, element_attrs={}, login=None, unique=True,
auto_unique_name=False, auto_unique_view=False)
```

This method adds an element into a config. It is used by various UI components to add new widget element to a particular view.

### param:

**search\_type** - the search type that this config belongs to

**view** - the specific view of the search type

**name** - the name of the element

### keyparam:

**class\_name** - the fully qualified class of the display

**action\_class\_name** - the fully qualified class of the action

**display\_options** - keyword options in a dictionary to construct the specific display

**action\_options** - keyword options in a dictionary to construct the specific action

**element\_attrs** - element attributes in a dictionary

**login** - login name if it is for a specific user

**unique** - add an unique element if True. update the element if False.

**auto\_unique\_name** - auto generate a unique element and display view name

**auto\_unique\_view** - auto generate a unique display view name

### return:

**boolean** - True

### example:

This will add a new element to the "character" view for a 3D asset

```
search_type = 'prod/asset'
view = 'characters'
class_name = 'tactic.ui.common.SimpleElementWdg'
server.add_config_element(search_type, view, class_name)
```

This will add a new element named "user" to the "definition" view. It contains detailed display and action nodes

```
data_dict = {} # some data here
search_type = 'prod/asset'
server.add_config_element(search_type, 'definition', 'user', class_name
= data_dict['class_name'], display_options=data_dict['display_options'],
element_attrs=data_dict['element_attrs'], unique=True,
action_class_name=data_dict['action_class_name'], action_options=data_dict['action_options'])
```

## add\_dependency

**add\_dependency(snapshot\_code, file\_path, type='ref')**

This method will append a dependency referent to an existing checkin.

All files are uniquely contained by a particular snapshot. Presently, this method does a reverse lookup by file name. This assumes that the filename is unique within the system, so it is not recommended unless it is known that naming conventions will produce unique file names for every this particular file. If this is not the case, it is recommended that `add_dependency_by_code()` is used.

### param:

**snapshot\_code** - the unique code identifier of a snapshot

**file\_path** - the path of the dependent file. This function is able to reverse map the file\_path to the appropriate snapshot

### keyparam:

**type** - type of dependency. Values include 'ref' and 'input\_ref'

ref = hierarchical reference: ie A contains B

input\_ref = input reference: ie: A was used to create B

**tag** - a tagged keyword can be added to a dependency to categorize the different dependencies that exist in a snapshot

### return:

**dictionary** - the resulting snapshot

## add\_dependency\_by\_code

**add\_dependency\_by\_code(to\_snapshot\_code, from\_snapshot\_code, type='ref')**

Append a dependency reference to an existing checkin. This dependency is used to connect various checkins together creating a separate dependency tree for each checkin.

### param:

**to\_snapshot\_code**: the snapshot code which the dependency will be connected to

**from\_snapshot\_code**: the snapshot code which the dependency will be connected from

**type** - type of dependency. Values include 'ref' and 'input\_ref'

ref = hierarchical reference: ie A contains B

**input\_ref** - input reference: ie: A was used to create B

**tag** - a tagged keyword can be added to a dependency to categorize the different dependencies that exist in a snapshot

### return:

**dictionary** - the resulting snapshot

## add\_directory

**add\_directory(snapshot\_code, dir, file\_type='main', mode="copy", dir\_naming="", file\_naming=")**

Add a full directory to an already existing checkin.

This informs TACTIC to treat the entire directory as single entity without regard to the structure of the contents. TACTIC will not know about the individual files and the directory hierarchy within the base directory and it is left up to the external program to interpret and understand this.

This is often used when logic on the exact file structure exists in some external source outside of TACTIC and it is deemed to be complicated to map this into TACTIC's snapshot definition.

### param:

**snapshot\_code** - a unique identifier key representing an object

**dir** - the directory that needs to be checked in

### keyparam:

**file\_type** - file type is used more as snapshot type here

**mode** - copy, move, preallocate, manual, inplace

**dir\_naming** - explicitly set a dir\_naming expression to use

**file\_naming** - explicitly set a file\_naming expression to use

### return:

**dictionary** - snapshot

### example:

This will create a new snapshot for a search\_key and add a directory using manual mode

```
dir = 'C:/images'
handoff_dir = my.server.get_handoff_dir()
shutil.copytree('%s/subfolder' %dir, '%s/images/subfolder' %handoff_dir)

snapshot_dict = my.server.create_snapshot(search_key, context='render')
snapshot_code = snapshot_dict.get('code')
my.server.add_directory(snapshot_code, dir, file_type='dir', mode='manual')
```



## add\_file

**add\_file(snapshot\_code, file\_path, file\_type='main', use\_handoff\_dir=False, mode=None, create\_icon=False)**

Add a file to an already existing snapshot. This method is used in piecewise checkins. A blank snapshot can be created using `create_snapshot()`. This method can then be used to successively add files to the snapshot.

In order to checkin the file, the server will need to have access to these files. There are a number of ways of getting the files to the server. When using copy or move mode, the files are either copied or moved to the "handoff\_dir". This directory is an agreed upon directory in which to handoff the files to the server. This mode is generally used for checking in user files. For heavy bandwidth checkins, it is recommended to use preallocated checkins.

### param:

**snapshot\_code** - the unique code identifier of a snapshot

**file\_path** - path of the file to add to the snapshot.

Optional: this can also be an array to add multiple files at once.

This has much faster performance than adding one file at a time.

Also, note that in this case, `file_types` must be an array of equal size.

### keyparam:

**file\_type** - type of the file to be added.

Optional: this can also be an array. See `file_path` argument for more information.

**use\_handoff\_dir** - DEPRECATED: (use mode arg) use handoff dir to checkin file. The handoff dir is an agreed upon directory between the client and server to transfer files.

**mode** - upload|copy|move>manual|inplace

the file to the server.

**create\_icon** - (True|False) determine whether to create an icon for this appended file. Only 1 icon should be created for each snapshot.

**dir\_naming** - explicitly set a dir\_naming expression to use

**file\_naming** - explicitly set a file\_naming expression to use

## **return:**

**dictionary** - the resulting snapshot

## **example:**

This will create a blank model snapshot for character chr001 and add a file

```
search_type = 'prod/asset'
code = 'chr001'
search_key = server.build_search_type(search_type, code)
context = 'model'
path = "./my_model.ma"

snapshot = server.create_snapshot(search_key, context)
server.add_file( snapshot.get('code'), path )
```

Different files should be separated by file type. For example,

to check in both a maya and houdini file in the same snapshot:

```
maya_path = "./my_model.ma"
houdini_path = "./my_model.hip"

server.add_file( snapshot_code, maya_path, file_type='maya' )
server.add_file( snapshot_code, houdini_path, file_type='houdini' )
```

To transfer files by uploading (using http protocol):

```
server.add_file( snapshot_code, maya_path, mode='upload' )
```

To create an icon for this file

---

```
path = 'image.jpg'
server.add_file( snapshot_code, path, mode='upload', create_icon=True )
```

To add multiple files at once

```
file_paths = [maya_path, houdini_path]
file_types = ['maya', 'houdini']
server.add_file( snapshot_code, file_paths, file_types=file_types, mode='upload' )
```

## add\_group

**add\_group(snapshot\_code, file\_path, file\_type, file\_range, use\_handoff\_dir=False, mode=None)**

Add a file range to an already existing snapshot

### param:

**snapshot\_code** - the unique code identifier of a snapshot

**file\_path** - path of the file to add to the snapshot

**file\_type** - type of the file to be added.

**file\_range** - range with format s-e/b

### keyparam:

**use\_handoff\_dir** - use handoff dir to checkin file

**mode** - one of 'copy','move','preallocate'

### return:

**dictionary** - the resulting snapshot

## **add\_initial\_tasks**

**add\_initial\_tasks(search\_key, pipeline\_code=None, processes=[])**

Add initial tasks to an object

### **param:**

**search\_key** - the key identifying a type of object as registered in the search\_type table.

### **keyparam:**

**pipeline\_code** - override the object's pipeline and use this one instead

**processes** - create tasks for the given list of processes

### **return:**

**list** - tasks created

## build\_search\_key

**build\_search\_key(search\_type, code, project\_code=None, column='code')**

Convenience method to build a search key from its components. A

search\_key uniquely identifies a specific subject. This string

that is returned is heavily used as an argument in the API to

identify an subject to operate one

A search key has the form: "prod/shot?project=bar&code=XG001"

where search\_type = "prod/shot", project\_code = "bar" and code = "XG001"

### param:

**search\_type** - the unique identifier of a search type: ie prod/asset

**code** - the unique code of the subject

### keyparam:

**project\_code** - an optional project code. If this is not

included, the project from get\_ticket() is added.

### return:

**string** - search key

### example:

```
search_type = "prod/asset"
code = "chr001"
search_key = server.build_search_key(search_type, code)
e.g. search_key = prod/asset?project=code=chr001
```

```
search_type = "sthpw/login"
code = "admin"
search_key = server.build_search_key(search_type, code, column='code')
e.g. search_key = sthpw/login?code=admin
```

## build\_search\_type

**build\_search\_type(search\_type, project\_code=None)**

Convenience method to build a search type from its components. It is a simple method that build the proper format for project scoped search types. A full search type has the form:

prod/asset?project=bar.

It uniquely defines a type of sobject in a project.

### param:

**search\_type** - the unique identifier of a search type: ie prod/asset

**project\_code (optional)** - an optional project code. If this is not included, the project from get\_ticket() is added.

### return:

**string** - search type string

### example:

```
search_type = "prod/asset"  
full_search_type = server.build_search_type(search_type)
```

## checkout

**checkout**(search\_key, context, version=-1, file\_type='main', dir='', level\_key=None, to\_sandbox\_dir=False, mode='copy')

Check out files defined in a snapshot from the repository. This will copy files to a particular directory so that a user can work on them.

### param:

**search\_key** - a unique identifier key representing an object

**context** - context of the snapshot

### keyparam:

**version** - version of the snapshot

**file\_type** - file type defaults to 'main'. If set to '\*', all paths are checked out

**level\_key** - the unique identifier of the level in the form of a search key

**to\_dir** - destination directory defaults to '.'

**to\_sandbox\_dir** - (True|False) destination directory defaults to

sandbox\_dir (overrides "to\_dir" arg)

**mode** - (copy|download)

to copy the files to the destination location

### return:

**list** - a list of paths that were checked out

## **clear\_upload\_dir**

**clear\_upload\_dir()**

Clears the upload directory to ensure clean checkins

**param:**

None

**keyparam:**

None

**return:**

None

## **create\_search\_type**

**create\_search\_type**(search\_type, title, description="", has\_pipeline=False)

Create a new search type

### **param:**

**search\_type** - Newly defined search\_type

**title** - readable title to display this search type as

### **keyparam:**

**description** - a brief description of this search type

**has\_pipeline** - determines whether this search type goes through a pipeline. Simply puts a pipeline\_code column in the table.

### **return**

**string** - the newly created search type

## create\_snapshot

**create\_snapshot(search\_key, context, snapshot\_type="file", description="No description", is\_current=True, level\_key=None, is\_revision=False )**

Create an empty snapshot

### param:

**search\_key** - a unique identifier key representing an subject

**context** - the context of the checkin

### keyparam:

**snapshot\_type** - [optional] describes what kind of a snapshot this is.

More information about a snapshot type can be found in the

prod/snapshot\_type subject

**description** - [optional] optional description for this checkin

**is\_current** - flag to determine if this checkin is to be set as current

**is\_revision** - flag to set this as a revision instead of a version

**level\_key** - the unique identifier of the level that this

is to be checked into

### return:

**dictionary** - representation of the snapshot created for this checkin

## create\_task

**create\_task**(search\_key, process="publish", subcontext=None, description=None, bid\_start\_date=None, bid\_end\_date=None, bid\_duration=None, assigned=None)

Create a task for a particular subject

### param:

**search\_key** - the key identifying a type of subject as registered in the search\_type table.

### keyparam:

**process** - process that this task belongs to

**subcontext** - the subcontext of the process (context = process/subcontext)

**description** - detailed description of the task

**bid\_start\_date** - the expected start date for this task

**bid\_end\_date** - the expected end date for this task

**bid\_duration** - the expected duration for this task

**assigned** - the user assigned to this task

### return:

**dictionary** - task that was created

## **delete\_subject**

**delete\_subject(search\_key)**

Invoke the delete method. Note: this function may fail due to dependencies. Tactic will not cascade delete. This function should be used with extreme caution because, if successful, it will permanently remove the existence of an object

### **param:**

**search\_key** - a unique identifier key representing an object.

Note: this can also be an array.

### **return:**

**dictionary** - a object that represents values of the object in the form name:value pairs

## directory\_checkin

**directory\_checkin**(search\_key, context, dir, snapshot\_type="directory", description="No description", file\_type='main', is\_current=True, level\_key=None, metadata={}, mode="copy", is\_revision=False)

Check in a directory of files. This informs TACTIC to treat the entire directory as single entity without regard to the structure of the contents. TACTIC will not know about the individual files and the directory hierarchy within the base directory and it is left up to the external program to interpret and understand this. This is often used when logic on the exact file structure exists in some external source outside of TACTIC and it is deemed too complicated to map this into TACTIC's snapshot definition.

### param:

**search\_key** - a unique identifier key representing an object

**dir** - the directory that needs to be checked in

### keyparam:

**snapshot\_type** - type of snapshot this checkin will have

**description** - description related to this checkin

**file\_type** - the type of file that will be associated with this group

**is\_current** - makes this snapshot current

**level\_key** - the search key of the level if used

**metadata** - add metadata to snapshot

**mode** - determines whether the files passed in should be copied, moved or uploaded. By default, this is 'copy'

**is\_revision** - flag to set this as a revision instead of a version

### return:

**dictionary** - snapshot

## download

```
download(my, url, to_dir=".", filename="", md5_checksum="")
```

Download a file from a given url

### param:

**url** - the url source location of the file

### keyparam:

**to\_dir** - the directory to download to

**filename** - the filename to download to, defaults to original filename

**md5\_checksum** - an md5 checksum to match the file against

### return:

**string** - path of the file downloaded

## eval

**eval(expression, search\_keys=[], mode=None, single=False, vars={}, show\_retired=False)**

Evaluate the expression. This expression uses the TACTIC expression language to retrieve results. For more information, refer to the expression language documentation.

### param:

**expression** - string expression

### keyparam:

**search\_keys** - the starting point for the expression.

**mode** - string|expression

**single** - True|False

**vars** - user defined variable

**show\_retired** - defaults to False to not return retired items

### return:

results of the expression. The results depend on the exact nature of the expression.

### example:

#1. Search for snapshots with context beginning with 'model' for the asset with the search key 'prod/asset?project=sample3d&id=96'

```
server = TacticServerStub.get()
exp = "@SOBJECT(sthpw/snapshot['context','EQ','^model'])"
result = server.eval(exp, search_keys=['prod/asset?project=sample3d&id=96'])
```

Please refer to the expression language documentation for numerous examples on how to use the expression language.

## **execute\_cmd**

**execute\_cmd**(class\_name, args={}, values={})

Execute a command

### **param:**

**class\_name** - the fully qualified class name of the widget

### **keyparam:**

**args** - keyword arguments required to create a specific widget

**values** - form values that are passed in from the interface

### **return:**

**string** - description of command

## **execute\_pipeline**

**execute\_pipeline(pipeline\_xml, package)**

Spawn an execution of a pipeline as delivered from

'get\_pipeline\_xml()'. The pipeline is a xml document that describes  
a set of processes and their handlers

### **param:**

**pipeline\_xml** - an xml document describing a standard Tactic pipeline.

**package** - a dictionary of data delivered to the handlers

### **return:**

**instance** - a reference to the interpreter

## **execute\_python\_script**

**execute\_python\_script**(class\_name, args={}, values={})

Execute a command

### **param:**

**script\_path** - script path in Script Editor, e.g. test/eval\_sobj

### **return:**

**dictionary** - returned data structure

## finish

### **finish()**

End the current transaction and cleans it up

### **params:**

description: this will be recorded in the transaction log as the  
description of the transaction

### **example:**

A full transaction inserting 10 shots. If an error occurs, all 10  
inserts will be aborted.

```
server.start('Start adding shots')
try:
    for i in range(0,10):
        server.insert("prod/shot", { 'code': 'XG%0.3d'%i } )
except:
    server.abort()
else:
    server.finish("10 shots added")
```

## get\_all\_children

**get\_all\_children(search\_key, child\_type, columns=[])**

Get all children of a particular child type of an subject

### **param:**

**ticket** - authentication ticket

**search\_key** - a unique identifier key representing an subject

**child\_type** - the search\_type of the children to search for

### **keyparam:**

**filters** - extra filters on the query : see query method for examples

**columns** - list of column names to be included in the returned dictionary

### **return:**

**list of dictionary** - a list of subjects dictionaries

## get\_all\_dependencies

```
get_all_dependencies(snapshot_code, mode='explicit', type='ref', include_paths=False,  
include_paths_dict=False, include_files=False, repo_mode='client_repo', show_retired=False)
```

Retrieve the latest dependent snapshots of the given snapshot

### param:

**search\_key** - unique identifier of subject whose snapshot we are  
looking for

### keyparam:

**mode** - explicit (get version as defined in snapshot)

- latest

- current

**type** - one of ref or input\_ref

**include\_paths** - flag to specify whether to include a `__paths__` property  
containing all of the paths in the dependent snapshots

**include\_paths\_dict** - flag to specify whether to include a  
`__paths_dict__` property containing a dict of all paths in the  
dependent snapshots

**include\_files** - includes all of the file objects referenced in the  
snapshots

**repo\_mode** - client\_repo, web, lib, relative

**show\_retired** - defaults to False so that it doesn't show retired dependencies

### return:

**list** - snapshots

## get\_all\_paths\_from\_snapshot

```
get_all_paths_from_snapshot(snapshot_code, mode='client_repo', expand_paths=False, filename_mode='')
```

Get all paths from snapshot

### param:

**snapshot\_code** - the unique code of the snapshot

### keyparam:

**mode** - forces the type of folder path returned to use the value from the appropriate tactic\_<SERVER\_OS>-conf.xml configuration file.

Values include 'lib', 'web', 'local\_repo', 'sandbox', 'client\_repo', 'relative'

lib = the NFS asset directory from the server point of view

web = the http asset directory from the client point of view

local\_repo = the local sync of the TACTIC repository

sandbox = the local sandbox (work area) designated by TACTIC

client\_repo (default) = the asset directory from the client point of view

If there is no value for win32\_client\_repo\_dir or linux\_client\_repo\_dir in the config, then the value for asset\_base\_dir will be used instead.

relative = the relative directory without any base

**expand\_paths** - expand the paths of a sequence check-in or for a directory check-in, it will list the contents of the directory as well

**filename\_mode** - source or "", where source reveals the source\_path of the check-in

### return:

**list** - paths

## **get\_by\_search\_key**

**get\_by\_search\_key**(search\_key)

Get the info on an object based on search key

### **param:**

**ticket** - authentication ticket

**search\_type** - the key identifying a type of object as registered in the search\_type table.

### **return:**

**list of dictionary** - objects that represent values of the object in the form of name:value pairs

## **get\_child\_types**

**get\_child\_types(search\_key)**

Get all the child search types

### **param:**

**search\_key** - a unique identifier key representing an subject

### **return:**

**list** - the child search types

## **get\_client\_api\_version**

**get\_client\_api\_version()**

**return:**

**string** - client api version

## get\_client\_dir

**get\_client\_dir(snapshot\_code, file\_type='main', mode='client\_repo')**

Get a dir segment from a snapshot

### param:

**snapshot\_code** - the unique code of the snapshot

### keyparam:

**file\_type** - each file in a snapshot is identified by a file type.

This parameter specifies which type. Defaults to 'main'

**mode** - Forces the type of folder path returned to use the value from the appropriate tactic\_<SERVER\_OS>-conf.xml configuration file.

Values include 'lib', 'web', 'local\_repo', 'sandbox', 'client\_repo', 'relative'

lib = the NFS asset directory from the server point of view

web = the http asset directory from the client point of view

local\_repo = the local sync of the TACTIC repository

sandbox = the local sandbox (work area) designated by TACTIC

client\_repo (default) = the asset directory from the client point of view

If there is no value for win32\_client\_repo\_dir or linux\_client\_repo\_dir in the config, then the value for asset\_base\_dir will be used instead.

relative = the relative directory without any base

### return:

**string** - directory segment for a snapshot and file type

### example:

If the tactic\_<SERVER\_OS>-conf.xml configuration file contains the following:

```
<win32_client_repo_dir>T:/assets</win32_client_repo_dir>
```

and if the call to the method is as follows:

```
snapshot = server.create_snapshot(search_key, context)
```

```
code = snapshot.get('code')
server.get_path_from_snapshot(snapshot.get('code'))
```

Then, on a Windows client, `get_client_dir()` will return:

```
T:/assets/sample3d/asset/chr/chr003/scenes
```

## **get\_client\_version**

**get\_client\_version()**

**return:**

**string** - Version of TACTIC that this client came from

## **get\_column\_info**

**get\_column\_info(search\_type)**

Get column information of the table given a search type

### **param:**

**search\_type** - the key identifying a type of subject as registered in the search\_type table.

**return - a dictionary of info for each column**

## **get\_column\_names**

**get\_column\_names(search\_type)**

This method will get all of the column names associated with a search type

### **param:**

**search\_type** - the search type used to query the columns for

### **return**

list of columns names

## get\_config\_definition

**get\_config\_definition(search\_type, view, element\_name)**

Get the widget configuration definition for an element

### **param:**

**search\_type** - search type that this config relates to

**view** - view to look for the element

**element\_name** - name of the element

### **keyparam:**

**personal** - True if it is a personal definition

### **return:**

**string** - xml of the configuration

## get\_dependencies

```
get_dependencies(snapshot_code, mode='explicit', tag='main', include_paths=False,
include_paths_dict=False, include_files=False, repo_mode='client_repo', show_retired=False)
```

Return the dependent snapshots of a certain tag

### params:

**snapshot\_code** - unique code of a snapshot

### keyparams:

**mode** - explicit (get version as defined in snapshot)

- latest

- current

**tag** - retrieve only dependencies that have this named tag

**include\_paths** - flag to specify whether to include a `__paths__` property

containing all of the paths in the dependent snapshots

**include\_paths\_dict** - flag to specify whether to include a

`__paths_dict__` property containing a dict of all paths in the

dependent snapshots

**include\_files** - includes all of the file objects referenced in the

snapshots

**repo\_mode** - client\_repo, web, lib, relative

**show\_retired** - defaults to False so that it doesn't show retired dependencies

### return:

a list of snapshots

## **get\_expanded\_paths\_from\_snapshot**

**get\_expanded\_paths\_from\_snapshot(snapshot\_code, file\_type='main')**

Return the expanded path of a snapshot (used for ranges of files)

### **param:**

**snapshot\_code** - the unique code of the snapshot

### **keyparam:**

**file\_type** - each file in a snapshot is identified by a file type.

This parameter specifies which type. Defaults to 'main'

### **return:**

**string** - path

## **get\_full\_snapshot\_xml**

**get\_full\_snapshot\_xml(snapshot\_code)**

Retrieve a full snapshot xml. This snapshot definition contains all the information about a snapshot in xml

### **param:**

**snapshot\_code** - unique code of snapshot

### **return:**

**string** - the resulting snapshot xml

## **get\_handoff\_dir**

**get\_handoff\_dir()**

Return a temporary path that files can be copied to

### **return:**

**string** - the directory to copy a file to handoff to Tactic without having to go through http protocol

## **get\_home\_dir**

**get\_home\_dir()**

OS independent method to Get the home directory of the current user.

### **return:**

**string** - home directory

## get\_info\_from\_user

**get\_info\_from\_user(force=False)**

Get input from the user about the users environment. Questions asked pertain to the location of the tactic server, the project worked on and the user's login and password. This information is stored in an .<login>.tacticrc file.

### **keyparam:**

**force** - if set to True, it will always ask for new infomation from the command prompt again

## **get\_login**

**get\_login()**

get the login user for this instantiated TacticServerStub

**return:**

**string** - the login name

## get\_md5\_info

**get\_md5\_info(md5\_list, texture\_codes, new\_paths, parent\_code, texture\_cls, file\_group\_dict, project\_code)**

Get md5 info for a given list of texture paths, mainly returning if this md5 is a match or not

### param:

**md5\_list** - md5\_list

**new\_paths** - list of file\_paths

**parent\_code** - parent code

**texture\_cls** - Texture or ShotTexture

**file\_group\_dict** - file group dictionary storing all the file groups

**project\_code** - project\_code

**mode** - texture matching mode (md5, file\_name)

### return:

**dictionary** - a dictionary of path and a subdictionary of is\_match, repo\_file\_code, repo\_path, repo\_file\_range

## **get\_parent**

**get\_parent(search\_key, columns=[])**

Get the parent of an object.

### **param:**

**search\_key** - a unique identifier key representing an object

### **keyparam:**

**columns** - the columns that will be returned in the object

### **return:**

**dictionary** - the parent object

## **get\_parent\_type**

**get\_parent\_type(search\_key)**

Get of the parent search type

### **param:**

**search\_key** - a unique identifier key representing an subject

### **return:**

**list** - a list of child search\_types

## get\_path\_from\_snapshot

**get\_path\_from\_snapshot(snapshot\_code, file\_type='main')**

Get a full path from a snapshot

### param:

**snapshot\_code** - the unique code / search\_key of the snapshot

### keyparam:

**file\_type** - each file in a snapshot is identified by a file type.

This parameter specifies which type. Defaults to 'main'

**mode** - Forces the type of folder path returned to use the value from the appropriate tactic\_<SERVER\_OS>-conf.xml configuration file.

Values include 'lib', 'web', 'local\_repo', 'sandbox', 'client\_repo', 'relative'

lib = the NFS asset directory from the server point of view

web = the http asset directory from the client point of view

local\_repo = the local sync of the TACTIC repository

sandbox = the local sandbox (work area) designated by TACTIC

client\_repo (default) = the asset directory from the client point of view

If there is no value for win32\_client\_repo\_dir or linux\_client\_repo\_dir in the config, then the value for asset\_base\_dir will be used instead.

relative = the relative directory without any base

### return:

**string** - the directory to copy a file to handoff to Tactic without having to go through http protocol

### example:

If the tactic\_<SERVER\_OS>-conf.xml configuration file contains the following:

```
<win32_client_repo_dir>T:/assets</win32_client_repo_dir>
```

and if the call to the method is as follows:

```
snapshot = server.create_snapshot(search_key, context)
code = snapshot.get('code')
server.get_path_from_snapshot(snapshot.get('code'))

# in a trigger
snapshot_key = my.get_input_value("search_key")
server.get_path_from_snapshot(snapshot_key)
```

Then, on a Windows client, `get_path_from_snapshot()` will return:

```
T:/assets/sample3d/asset/chr/chr003/scenes/chr003_rig_v003.txt
```

## get\_paths

**get\_paths( search\_key, context="publish", version=-1, file\_type='main', level\_key=None, single=False, versionless=False)**

Get paths from an object

### params:

**search\_key** - a unique identifier key representing an object

### keyparams:

**context** - context of the snapshot

**version** - version of the snapshot

**file\_type** - file type defined for the file node in the snapshot

**level\_key** - the unique identifier of the level that this

was checked into

**single** - If set to True, the first of each path set is returned

**versionless** - boolean to return the versionless snapshot, which takes a version of -1 (latest) or 0 (current)

### return

A dictionary of lists representing various paths. The paths returned

are as follows:

- **client\_lib\_paths**: all the paths to the repository relative to the client
- **lib\_paths**: all the paths to the repository relative to the server
- **sandbox\_paths**: all of the paths mapped to the sandbox
- **web**: all of the paths relative to the http server

## **get\_pipeline\_processes**

**get\_pipeline\_processes(search\_key, recurse=False)**

DEPRECATED: use get\_pipeline\_processes\_info()

Retrieve the pipeline processes information of a specific subject.

### **param:**

**search\_key** - a unique identifier key representing an subject

### **keyparams:**

**recurse** - boolean to control whether to display sub pipeline processes

### **return:**

**list** - process names of the pipeline

## get\_pipeline\_processes\_info

**get\_pipeline\_processes\_info(search\_key, recurse=False, related\_process=None)**

Retrieve the pipeline processes information of a specific subject. It provides information from the perspective of a particular process if related\_process is specified.

### param:

**search\_key** - a unique identifier key representing an subject

### keyparams:

**recurse** - boolean to control whether to display sub pipeline processes

**related\_process** - given a process, it shows the input and output processes and contexts

### return:

**dictionary** - process names of the pipeline or a dictionary if related\_process is specified

## get\_pipeline\_xml

**get\_pipeline\_xml(search\_key)**

DEPRECATED: use get\_pipeline\_xml\_info()

Retrieve the pipeline of a specific subject. The pipeline return is an xml document and an optional dictionary of information.

### **param:**

**search\_key** - a unique identifier key representing an subject

### **return:**

**dictionary** - xml and the optional hierarachy info

## get\_pipeline\_xml\_info

**get\_pipeline\_xml\_info(search\_key, include\_hierarchy=False)**

Retrieve the pipeline of a specific subject. The pipeline

returned is an xml document and an optional dictionary of information.

### **param:**

**search\_key** - a unique identifier key representing an subject

### **keyparam:**

**include\_hierarchy** - include a list of dictionary with key info on each process of the pipeline

### **return:**

**dictionary** - xml and the optional hierarchy info

## get\_preallocated\_path

**get\_preallocated\_path(snapshot\_code, file\_type='main', file\_name='', mkdir=True, protocol='client\_repo', ext='')**

Get the preallocated path for this snapshot. It assumes that this checkin actually exists in the repository and will create virtual entities to simulate a checkin. This method can be used to determine where a checkin will go. However, the snapshot must exist using `create_snapshot()` or some other method. For a pure virtual naming simulator, use `get_virtual_snapshot_path()`.

### param:

**snapshot\_code** - the code of a preallocated snapshot. This can be create by `get_snapshot()`

### keyparam:

**file\_type** - the type of file that will be checked in. Some naming conventions make use of this information to separate directories for different file types

**file\_name** - the desired file name of the preallocation. This information may be ignored by the naming convention or it may use this as a base for the final file name

**mkdir** - an option which determines whether the directory of the preallocation should be created

**protocol** - It's either `client_repo`, `sandbox`, or `None`. It determines whether the path is from a client or server perspective

**ext** - force the extension of the file name returned

### return:

**string** - the path where `add_file()` expects the file to be checked into

### example:

it saves time if you get the path and copy it to the final destination first.

---

```
snapshot = my.server.create_snapshot(search_key, context)
snapshot_code = snapshot.get('code')
file_name = 'input_file_name.txt'
orig_path = 'C:/input_file_name.txt'
path = my.server.get_preallocated_path(snapshot_code, file_type, file_name)

# the path where it is supposed to go is generated
new_dir = os.path.dirname(path)
if not os.path.exists(new_dir):
    os.makedirs(new_dir)
shutil.copy(orig_path, path)
my.server.add_file(snapshot_code, path, file_type, mode='preallocate')
```

## **get\_project**

**get\_project()**

get the project code for this instantiated TacticServerStub

**return:**

**string** - project code

## **get\_protocol**

**get\_protocol()**

**return:**

**string** - local or xmlrpc

## **get\_related\_types**

**get\_related\_types(search\_type)**

Get related search types given a search type

### **param:**

**search\_type** - the key identifying a type of subject as registered in the search\_type table.

**return - list of search\_types**

## get\_resource\_path

**get\_resource\_path(login=None)**

Get the resource path of the current user. It differs from

create\_resource\_paths() which actually create dir. The resource path

identifies the location of the file which is used to cache connection information.

An exmple of the contents is shown below:

```
login=admin
server=localhost
ticket=30818057bf561429f97af59243e6ef21
project=unittest
```

The contents in the resource file represent the defaults to use

when connection to the TACTIC server, but may be overridden by the

API methods: set\_ticket(), set\_server(), set\_project() or the

environment variables: TACTIC\_TICKET, TACTIC\_SERVER, and TACTIC\_PROJECT

Typically this method is not explicitly called by API developers and

is used automatically by the API server stub. It attempts to get from

home dir first and then from temp\_dir is it fails.

### param:

**login (optional)** - login code. If not provided, it gets the current system user

### return:

**string** - resource file path

## **get\_server\_api\_version**

**get\_server\_api\_version()**

**return:**

**string** - server API version

## **get\_server\_name**

**get\_server\_name()**

get the server name for this instantiated TacticServerStub

**return:**

**string** - server name

## **get\_server\_version**

**get\_server\_version()**

**return:**

**string** - server version

## get\_snapshot

**get\_snapshot(search\_key, context="publish", version='-1', level\_key=None, include\_paths=False, include\_full\_xml=False, include\_paths\_dict=False, include\_files=False, include\_web\_paths\_dict=False, versionless=False)**

Method to retrieve an object's snapshot

Retrieve the latest snapshot

### param:

**search\_key** - unique identifier of object whose snapshot we are looking for

### keyparam:

**context** - the context of the snapshot

**version** - snapshot version

**revision** - snapshot revision

**level\_key** - the unique identifier of the level in the form of a search key

**include\_paths** - flag to include a list of paths to the files in this snapshot.

**include\_full\_xml** - whether to include full xml in the return

**include\_paths\_dict** - flag to specify whether to include a `__paths_dict__` property containing a dict of all paths in the dependent snapshots

**include\_web\_paths\_dict** - flag to specify whether to include a `__web_paths_dict__` property containing a dict of all web paths in the returned snapshots

**include\_files** - includes all of the file objects referenced in the snapshots

**versionless** - boolean to return the versionless snapshot, which takes a version of -1 (latest) or 0 (current)

### return:

**dictionary** - the resulting snapshot

### example:

```
search_key = 'prod/asset?project=sample3d&code=chr001'  
snapshot = server.get_snapshot(search_key, context='icon', include_files=True)
```

```
# get the versionless snapshot  
search_key = 'prod/asset?project=sample3d&code=chr001'  
snapshot = server.get_snapshot(search_key, context='anim', include_paths_dict=True,  
versionless=True)
```

## get\_ticket

**get\_ticket(login, password)**

Get an authentication ticket based on a login and password.

This function first authenticates the user and then issues a ticket.

The returned ticket is used on subsequent calls to the client api

### **param:**

**login** - the user that is used for authentications

**password** - the password of that user

### **return:**

**string** - ticket key

## **get\_types\_from\_instance**

gets the connector types from an instance type

### **param:**

**instance\_type** - the search type of the instance

### **return:**

**tuple** - (from\_type, parent\_type)

a tuple with the from\_type and the parent\_type. The from\_type is the connector type and the parent type is the search type of the parent of the instance

## get\_unique\_subject

**get\_unique\_subject(search\_type, data={})**

This is a special convenience function which will query for an object and if it doesn't exist, create it. It assumes that this object should exist and spares the developer the logic of having to query for the subject, test if it doesn't exist and then create it.

### param:

**search\_type** - the type of the subject

**data** - a dictionary of name/value pairs that uniquely identify this subject

### return:

**subject** - unique subject matching the criteria in data

## get\_virtual\_snapshot\_path

**get\_virtual\_snapshot\_path**(search\_key, context, snapshot\_type="file", level\_key=None, file\_type='main', file\_name='', mkdirs=False, protocol='client\_repo', ext='')

Create a virtual snapshot and returns a path that this snapshot would generate through the naming conventions. This is most useful testing naming conventions.

### param:

snapshot creation:

-----

**search\_key** - a unique identifier key representing an subject

**context** - the context of the checkin

### keyparam:

**snapshot\_type** - [optional] describes what kind of a snapshot this is.

More information about a snapshot type can be found in the prod/snapshot\_type subject

**description** - [optional] optional description for this checkin

**level\_key** - the unique identifier of the level that this is to be checked into

### keyparam:

path creation:

-----

**file\_type** - the type of file that will be checked in. Some naming conventions make use of this information to separate directories for different file types

**file\_name** - the desired file name of the preallocation. This information may be ignored by the naming convention or it may use this as a base for the final file name

**mkdir** - an option which determines whether the directory of the preallocation should be created

**protocol** - It's either client\_repo, sandbox, or None. It determines whether the path is from a client or server perspective

**ext** - force the extension of the file name returned

**return:**

**string** - path as determined by the naming conventions

## get\_widget

**get\_widget(class\_name, args={}, values={})**

Get a defined widget

### params:

**class\_name** - the fully qualified class name of the widget

### keyparams:

**args** - keyword arguments required to create a specific widget

**values** - form values that are passed in from the interface

### return:

**string** - html form of the widget

### example:

```
class_name = 'TableLayoutWdg'
```

```
args = {
```

```
  'view': 'manage',
```

```
  'search_type': 'prod/asset',
```

```
}
```

```
widget = server.get_widget(class_name, args))
```

## group\_checkin

**group\_checkin**(search\_key, context, file\_path, file\_range, snapshot\_type="sequence", description="", file\_type='main', metadata={}, mode=None, is\_revision=False, info={})

Check in a range of files. A range of file is defined as any group of files that have some sequence of numbers grouping them together.

An example of this includes a range frames that are rendered.

Although it is possible to add each frame in a range using add\_file, adding them as as sequence is lightweight, often significantly reducing the number of database entries required. Also, it is understood that test files form a range of related files, so that other optimizations and manipulations can be operated on these files accordingly.

### param:

**search\_key** - a unique identifier key representing an sobject

**file\_path** - expression for file range: ./blah.####.jpg

**file\_type** - the typ of file this is checked in as. Default = 'main'

**file\_range** - string describing range of frames in the form '1-5/1'

### keyparam:

**snapshot\_type** - type of snapshot this checkin will have

**description** - description related to this checkin

**file\_type** - the type of file that will be associated with this group

**metadata** - add metadata to snapshot

**mode** - determines whether the files passed in should be copied, moved or uploaded. By default, this is a manual process (for backwards compatibility)

**is\_revision** - flag to set this as a revision instead of a version

**info** - dict of info to pass to the ApiClientCmd

### return:

**dictionary** - snapshot

## insert

**insert(search\_type, data, metadata={}, parent\_key=None, info={}, use\_id=False, triggers=True)**

General insert for creating a new subject

### param:

**search\_key** - a unique identifier key representing an subject

**data** - a dictionary of name/value pairs which will be used to update the subject defined by the search\_key.

**parent\_key** - set the parent key for this subject

### keyparam:

**metadata** - a dictionary of values that will be stored in the metadata attribute if available

**info** - a dictionary of info to pass to the ApiClientCmd

**use\_id** - use id in the returned search key

**triggers** - boolean to fire trigger on insert

### return:

**dictionary** - represent the subject with it's current data

### example:

insert a new asset

```
search_type = "prod/asset"

{
    'code': chr001,
    'description': 'Main Character'
}

insert( search_type, data )
```

insert a new note with a shot parent

```
# get shot key
shot_key = server.build_search_key(search_type='prod/shot',code='XG001')

data = {
    'context': 'model',
```

```
    'note': 'This is a modelling note',  
    'login': server.get_login()  
}  
  
server.insert( search_type, data, parent_key=shot_key)
```

insert a note without firing triggers

```
search_type = "sthpw/note"  
  
data = {  
    'process': 'roto',  
    'context': 'roto',  
    'note': 'The keys look good.',  
    'project_code': 'art'  
}  
  
server.insert( search_type, data, triggers=False )
```

## insert\_update

**insert\_update(search\_key, data, metadata={}, parent\_key=None, info={}, use\_id=False, triggers=True)**

Insert if the entry does not exist, update otherwise

### param:

**search\_key** - a unique identifier key representing an subject.

**data** - a dictionary of name/value pairs which will be used to update

the subject defined by the search\_key

### keyparam:

**metadata** - a dictionary of values that will be stored in the metadata attribute if available

**parent\_key** - set the parent key for this subject

**info** - a dictionary of info to pass to the ApiClientCmd

**use\_id** - use id in the returned search key

**triggers** - boolean to fire trigger on insert

### return:

**dictionary** - represent the subject with its current data.

## log

**log(level, message, category="default")**

Log a message in the logging queue. It is often difficult to see output of a trigger unless you are running the server in debug mode. In production mode, the server sends the output to log files which are general buffered, so it cannot be predicted exactly when this output will be dumped to a file.

This method will make a request to the server and store the message in the database in the debug log table.

### **param:**

**level** - debug level

**message** - freeform string describing the entry

### **keyparam:**

**category** - a label for the type of message being logged.

It defaults to "default"

## query

**query**(search\_type, filters=[], columns=[], order\_bys=[], show\_retired=False, limit=None, offset=None, single=False, distinct=None, return\_objects=False)

General query for subject information

### param:

**search\_type** - the key identifying a type of subject as registered in the search\_type table.

### keyparam:

**filters** - an array of filters to alter the search

**columns** - an array of columns whose values should be retrieved

**order\_bys** - an array of order\_by to alter the search

**show\_retired** - sets whether retired subjects are also returned

**limit** - sets the maximum number of results returned

**single** - returns only a single object

**distinct** - specify a distinct column

**return\_objects** - return objects instead of dictionary. This works only when using the API on the server.

### return:

**list of dictionary/objects** - Each array item represents an object and is a dictionary of name/value pairs

### example:

```
filters = []
filters.append( ("code", "XG002") )
order_bys = ['timestamp desc']
columns = ['code']
server.query(ticket, "prod/shot", filters, columns, order_bys)
```

The arguments "filters", "columns", and "order\_bys" are optional

The "filters" argument is a list. Each list item represents an individual filter. The forms are as follows:

```
(column, value)           -> where column = value
(column, (value1,value2)) -> where column in (value1, value2)
(column, op, value)       -> where column op value
      where op is ('like', '<=', '>=', '>', '<', 'is', '~', '!~', '~*', '!~*')
(value)                   -> where value
```

## query\_snapshots

**query\_snapshots(filters=None, columns=None, order\_bys=[], show\_retired=False, limit=None, offset=None, single=False, include\_paths=False, include\_full\_xml=False, include\_paths\_dict=False, include\_parent=False, include\_files=False)**

thin wrapper around query, but is specific to querying snapshots

with some useful included flags that are specific to snapshots

### params:

**ticket** - authentication ticket

**filters** - (optional) an array of filters to alter the search

**columns** - (optional) an array of columns whose values should be retrieved

**order\_bys** - (optional) an array of order\_by to alter the search

**show\_retired** - (optional)

returned

**limit** - sets the maximum number of results returned

**single** - returns a single object that is not wrapped up in an array

**include\_paths** - flag to specify whether to include a `__paths__` property containing a list of all paths in the dependent snapshots

**include\_paths\_dict** - flag to specify whether to include a `__paths_dict__` property containing a dict of all paths in the dependent snapshots

**include\_full\_xml** - flag to return the full xml definition of a snapshot

**include\_parent** - includes all of the parent attributes in a `__parent__` dictionary

**include\_files** - includes all of the file objects referenced in the snapshots

### return:

list of snapshots

## **reactivate\_subject**

**reactivate\_subject**(search\_key)

Invoke the reactivate method.

### **param:**

**search\_key** - the unique key identifying the subject.

### **return:**

**dictionary** - subject that represents values of the subject in the form name:value pairs

## redo

**redo(transaction\_ticket=None, transaction\_id=None)**

Redo an operation. If no transaction id is given, then the last undone operation of this user on this project is redone

### **keyparam:**

**transaction\_ticket** - explicitly redo a specific transaction

**transaction\_id** - explicitly redo a specific transaction by id

## **retire\_subject**

**retire\_subject(search\_key)**

Invoke the retire method. This is preferred over delete\_subject if you are not sure whether other subjects has dependency on this.

### **param:**

**search\_key** - the unigue key identifying the sobject.

### **return:**

**dictionary** - sobject that represents values of the sobject in the form name:value pairs

## **set\_current\_snapshot**

**set\_current\_snapshot(snapshot\_code)**

Set this snapshot as a "current" snapshot

### **param:**

**snapshot\_code** - unique code of snapshot

### **return:**

**string** - the resulting snapshot xml

## **set\_login\_ticket**

**set\_login\_ticket(ticket)**

Set the login ticket with the ticket key

## **set\_project**

**set\_project(project\_code)**

Set the project code

### **param:**

**project\_code** - designated project the subsequent API calls will operate in

## **set\_protocol**

**get\_protocol()**

**param:**

**string** - local or xmlrpc

## **set\_server**

**set\_server(server\_name)**

Set the server name for this XML-RPC server

### **param:**

**server\_name** - server name or IP address of the TACTIC server

## simple\_checkin

```
simple_checkin( search_key, context, file_path, snapshot_type="file", description="No description",
use_handoff_dir=False, file_type="main", is_current=True, level_key=None, breadcrumb=False,
metadata={}, mode=None, is_revision=False, info={}, keep_file_name=False, create_icon=True,
checkin_cls='pyasm.checkin.FileCheckin', context_index_padding=None, checkin_type="strict",
source_path=None )
```

Simple method that checks in a file.

### param:

**search\_key** - a unique identifier key representing an object

**context** - the context of the checkin

**file\_path** - path of the file that was previously uploaded

### keyparam:

**snapshot\_type** - [optional] describes what kind of a snapshot this is.

More information about a snapshot type can be found in the

prod/snapshot\_type object

**description** - [optional] optional description for this checkin

**file\_type** - [optional] optional description for this file\_type

**is\_current** - flag to determine if this checkin is to be set as current

**level\_key** - the unique identifier of the level that this

is to be checked into

**breadcrumb** - flag to leave a .snapshot breadcrumb file containing

information about what happened to a checked in file

**metadata** - a dictionary of values that will be stored as metadata

on the snapshot

**mode** - inplace, upload, copy, move, inplace

**is\_revision** - flag to set this as a revision instead of a version

**create\_icon** - flag to create an icon on checkin

**info** - dict of info to pass to the ApiClientCmd

**keep\_file\_name** - keep the original file name

**create\_icon** - Create an icon by default

**checkin\_cls** - checkin class

**context\_index\_padding** - determines the padding used for context

indexing: ie: design/0001

**checkin\_type** - auto or strict which controls whether to auto create versionless

**source\_path** - explicitly give the source path

**return:**

**dictionary** - representation of the snapshot created for this checkin

## **split\_search\_key**

**split\_search\_key(search\_key)**

Convenience method to split a search\_key in into its search\_type and search\_code/id components. Note: only accepts the new form prod/asset?project=sample3d&code=chr001

### **param:**

**search\_key** - the unique identifier of a subject

### **return:**

**tuple** - search type, search code/id

## start

**start(title, description="")**

Start a transaction. All commands using the client API are bound in a transaction. The combination of start(), finish() and abort() makes it possible to group a series of API commands in a single transaction. The start/finish commands are not necessary for query operations (like query(...), get\_snapshot(...), etc).

### param:

**title** - the title of the command to be executed. This will show up on transaction log

### keyparam:

**description** - the description of the command. This is more detailed.

### example:

A full transaction inserting 10 shots. If an error occurs, all 10 inserts will be aborted.

```
server.start('Start adding shots')
try:
    for i in range(0,10):
        server.insert("prod/shot", { 'code': 'XG%0.3d'%i } )
except:
    server.abort()
else:
    server.finish("10 shots added")
```

## undo

**undo(transaction\_ticket=None, transaction\_id=None, ignore\_files=False)**

undo an operation. If no transaction id is given, then the last operation of this user on this project is undone

### keyparam:

**transaction\_ticket** - explicitly undo a specific transaction

**transaction\_id** - explicitly undo a specific transaction by id

**ignore\_files** - flag which determines whether the files should also be undone. Useful for large preallocated checkins.

## update

**update(search\_key, data={}, metadata={}, parent\_key=None, info={}, use\_id=False, triggers=True)**

General update for updating subject

### param:

**search\_key** - a unique identifier key representing an subject.

Note: this can also be an array, in which case, the data will

be updated to each subject represented by this search key

### keyparam:

**data** - a dictionary of name/value pairs which will be used to update

the subject defined by the search\_key

Note: this can also be an array. Each data dictionary element in

the array will be applied to the corresponding search key

**parent\_key** - set the parent key for this subject

**info** - a dictionary of info to pass to the ApiClientCmd

**metadata** - a dictionary of values that will be stored in the metadata attribute if available

**use\_id** - use id in the returned search key

**triggers** - boolean to fire trigger on update

### return:

**dictionary** - represent the subject with its current data.

If search\_key is an array, This will be an array of dictionaries

## update\_config

**update\_config(search\_type, view, element\_names)**

Update the widget configuration like ordering for a view

### **param:**

**search\_type** - search type that this config relates to

**view** - view to look for the element

**element\_names** - element names in a list

### **return:**

**string** - updated config xml snippet

## update\_multiple

**update\_multiple(data, triggers=True)**

Update for several subjects with different data in one function call. The data structure contains all the information needed to update and is formatted as follows:

```
data = {  
    search_key1: { column1: value1, column2: value2 }  
    search_key2: { column1: value1, column2: value2 }  
}
```

### params:

**data** - data structure containing update information for all subjects

### keyparam:

**data** - a dictionary of name/value pairs which will be used to update the subject defined by the search\_key

Note: this can also be an array. Each data dictionary element in the array will be applied to the corresponding search key

**triggers** - boolean to fire trigger on insert

### return:

None

## **upload\_file**

**upload\_file(path)**

Use http protocol to upload a file through http

### **param:**

**path** - the name of the file that will be uploaded