TACTIC Developer

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Developer Start-up

1.1 Development Concepts

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 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."

1.2 Architecture Overview

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

SObject - Model(M)	Provides the data model. All interactions with the data model use sObjects and their derived classes.
Widget - View(V)	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).
Command - Command©	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.
Search	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 sType 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.

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 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 be 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.

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 stored 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.

It is just as critical to be able to navigate the filesystem and understand what is located there. Therefore, advanced naming conventions are filtered through naming classes, which use clear procedures to create filenames based on metadata in the database. On the other hand, naming conventions can be driven by some expressions such as {sobject.code}_{snapshot.context}_v{snapshot.version}.{ex

Directories and file naming are handled slightly differently. TACTIC builds file names 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.

1.3 The TACTIC Script 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 file system.

image

The TacticServerStub.log() method writes to the table named *debug_log* in the sthpw database.

The first parameter of the TacticServerStub.log() method is named **level**. The argument for **level** can be one of the following keywords:

level	critical
error	warning
info	debug - arbitrary debug level category

The TacticServerStub.log() method can be used as follows:

```
var server = TacticServerStub.get()
server.log('debug','My log message for the debug group.')
```

The debug level argument provides the convenience of grouping the Debug Log table by debug levels. This table can be found under:

Admin Views → **Server** → **Debug Log**

image

Note

These 5 debug levels are arbitrary.

The only purpose the levels serve are to group the messages when they are sorted in the table.

While writing scripts in the TACTIC Script Editor, messages can be output to the **Web Client Output Log**.

Below are the 5 Javascript methods in use. The most vocal method, log.critical(), is at the top:

image

Below is the Output Log console from above the sample script. It can be found under:

Main Gear menu → **Tools** → **Web Client Output Log**.

The level of the log messages which appear in the Javascript Output Client Log can be controlled. The level can be adjusted under: **My Admin** → **User Preferences**.

Below is a table to illustrate what the setting for each level will display

critical setting	only display messages that are from log.critical()
error setting	only display messages that are from log.critical() or log.error()
warning setting	only display messages that are from log.critical() or log.error() or log.warning()
info setting	only display messages that are from log.critical() or log.error() or log.warning() or log.info()
debug setting	only display messages that are from log.critical() or log.error() or log.warning() or log.info() or log.debug()

For example, if the Web Client Logging Level is set in the preferences to the **warning** level, we will only see messages that are from log.warning(), log.error() and log.critical(). ie. Only messages at the same level or above that level will be displayed in the Web Client Output Log.

image

Example 1: Insert A New sObject

```
// INSERT A NEW SOBJECT
```

```
var server = TacticServerStub.get();

var code = "truck";
var asset_name = "truck";
var description = "A model of a truck.";
var search_type = "toy_factory/lego_set";
var project = "toy_factory";
var data = {
    'code': code,
    'name': asset_name,
    'description': description
};

var search_key = server.build_search_key(search_type, code, project);
var result = server.insert(search_type, data);
log.debug(result);
```

Results after insert:

Example 2: Get An sObject by Its Search Key

```
// GET BY SEARCH_KEY
var server = TacticServerStub.get();

var search_type = "toyrus/lego_set";
var code = "model_crane";
var project = "toyrus";

var search_key = server.build_search_key(search_type, code, project);
var result = server.get_by_search_key(search_key);
alert(result.description);
server.log("debug", result);
```

Results after get_by_search_key():

Example 3: Update An Existing sObject

```
// UPDATE EXISTING SOBJECT

var server = TacticServerStub.get();

var code = "model_crane";
var project = "toyrus";
var asset_name = "model crane";
var description = "Revised description of a crane.";
var search_type = "toyrus/lego_set";
var data = {
    'code': code,
    'name': asset_name,
    'description': description
};
var search_key = server.build_search_key(search_type, code, project);
var result = server.update(search_key, data);
server.log("debug", result);
```

*Results after update:**Example 4: Retire An Existing sObject*

```
// RETIRE AN EXISTING OBJECT

var server = TacticServerStub.get();

var search_type = "toyrus/lego_set";
var code = "model_crane";
var project = "toyrus";

var search_key = server.build_search_key(search_type, code, project);
var results = server.retire_sobject(search_key);
server.log("debug", result);
```

Results after retire:

2 Client API

2.1 Client API Setup

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.

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.

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:

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.

2.2 Client API 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.)

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.

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.

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

Note
this section is still in active development.

2.3 Basic Operations in Python and Javascript

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

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.

```

```

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") )
```

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 specified sObject.

The TACTIC Client API can be accessed in Javascript as well as Python. One can deduce its usage from the Python Client API doc. One main point to notice is that the keyparams in the Client API doc, also known as keyword argumnets, should be expressed as a hash {} in javascript. 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});
```

\5. To get the latest snapshot of the asset chr001 for the current project

```
var server = TacticServerStub.get();
var sk = server.build_search_key('prod/asset','chr001');
var snapshot = server.get_snapshot(sk,  {context:'anim', include_paths_dict: true,  ↩
    versionless: false});
log.critical(snapshot);
```

\6. To run a query of snapshots using filters and limit keyword argumnets

```
var server = TacticServerStub.get();
var filters = [];
// use built-in expression operator EQ, NEQ, EQI, or NEQI to specify the search_type has to ↩
     contain prod/shot
filters.push(['search_type', 'EQ','prod/shot']);
filters.push(['project_code','sample3d']);
var snapshot = server.query_snapshots({filters: filters, limit: 5});
log.critical(snapshot);
```

2.4 Checkin / Checkout Operations

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

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

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 check-in 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 check-in. 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

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 descibed 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 check-ins for both sequence and directory check-ins.

As the name suggests, a directory check-in 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 check-in 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 check-ins (simple_checkin()), because TACTIC treats a directory check-in in a similar manner to a file check-in. 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.

TACTIC allows you to build up a check-in piecewise or stages. This is a powerful feature because you can build a check-in 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 check-in. 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>
```

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 check-in 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 check-in 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 check-in examples is that the mode parameter is set to "move".

Preallocated check-ins are the most efficient check-ins. 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 check-ins enable a client process to be checked directly into the repository. They are recommended for check-ins 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 check-in mode include rendering frames, ingesting plates, simulating data, and so on.

The following steps describe the process for preallocating check-ins:

1. Create an empty snapshot to reserve a check-in 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 check-in modes.

Because the simple_checkin(), group_checkin() and directory_checkin() functions perform the entire check-in process in one step, you cannot use them for preallocated check-ins. Instead, you would use a piecewise check-in to build up the checked in parts. The following is an example of a preallocated check-in 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 check-ins. Files are created directly to the repository, eliminating all of the unnecessary copying of files around the servers. When groups of files reach the muti-gigabyte or even terabyte range, it becomes prohibitively expensive to check in files in the traditional manner. Preallocated check-ins maximize the use of your internal system architecture.

In general, the in-place check-in should be considered as the last resort. In-place check-ins 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 check-in 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 check-ins.

2.5 Snapshot Dependency

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)

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.

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.

3 Changes

3.1 Search ID to Search Code

A change made in TACTIC 4.0 is the use of search code instead of search id when relating sObjects to their snapshots (or checkins). Until 4.0, the search id was being used to maintain this relation. Now, if you look at the code column of a sObject and the search code column of a snapshot checked in to this sObject, you will find that both have the same value. This tells TACTIC that the snapshot is associated with this sObject.

image

image

The reason for this change was merging issues between multiple tables of snapshots. When using search id to merge between tables, there were many discrepancies which could not have been easily solved. Using search code to merge tables is a much easier process. There are also other reasons which are not very important.

4 Custom Widgets

5 Not Organizaed Yet

5.1 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 sobjects, these will be in the form of a dictionary like all other sobjects 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 refere to the CustomLayoutWdg in the Widget Reference documentation for more information on how to use the CustomLayoutWdg.

5.2 Plugin Versions

5.3 Tactic Checkin Process

5.4 Custom Widget Basics

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

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

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 sobjects 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

The most convenient method to test and implement the Javascript examples is in the TACTIC Script Editor. This can be convenient accesses by pressing the "9" hot key to bring it up. Alternatively, the TACTIC Script Editor can be brought up under the gear menu under: **Tools** → **TACTIC Script Editor**.

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.

The simplest way to view this is to open up the TACTIC Script Editor and input the following code:

```
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 going to **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. Input the following into the **config*field and for *view** input *example01*.<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 TACTIC Script Editor, enter the following:

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

The following will appear when you click on "Run" in the TACTIC Script Editor the script above:

image

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.

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 versions may remove this requirement, but currently this is 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 the 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.

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]@COU <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.

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 and expression lanaguage and sometimes, it is necessary to have full programming logic. Mako provides a path to create content that is too complex for the expression alnaguage 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 done with with the "include_make" 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 acccesed here as well:num_snapshots = server.eval("@COUNT(sthpw/snapshot[*contex* % 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.

The CustomLayoutWdg can be used inside of a table element. This makes it easy to create arbiraritly complex table elements within a standard TACTIC table layout widget. The following displays the number of tasks for the row sobject.<config> <my_view> <element name=*num_tasks*> <display class=*tactic.ui.panel.CustomLayoutWdg*> <html> <div class=*top*> <b>[expr]@COU
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 sobject that corresponds to the table row that is passed in and is used as the starting sobject for all expressions. The following expression finds the number of tasks for the sobject 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 sobject for the row available through the bvr object passed into the behvaior 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 sobject.

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_inpu*
<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 sobject.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});

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 pop-up.

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</searc
<view>table</view> </display> </element> </example11> </config>

5.5 Setting Up a Development Environment

Comming Soon

5.6 Packaging a Plugin

image

A TACTIC plugin package is simply a .zip file containing all the files of a plugin. Plugins are installed in the following directory:

<TACTIC_DATA_DIR>/plugins

The .zip files are usually stored in:

<TACTIC_DATA_DIR>/dist

Plugins are defined into categories. Due to the flexibility of the plugin architecture, a single plugin can package tools, columns, and themes in any combination. These categories are only used to organize plugins and can also bootstrap common functionality that would be packaged into a plugin.

All of these will have most of the view definitions in the Custom Layout Editor. Each individual view can have a type. See Custom Layout Editor documentation for more information on this.

- project: this defines the structure of the project. It may or may not include a theme, but it is usually possible to use different themes for a given project provided the theme has been set up correctly.
- theme: a theme defines the look and feel of a project as experienced by end users. A theme should have the following requirements:
  - a means of displaying links as represented by the side bar.
  - a means of logging out
  - overriding the login page (optional)
- column - This represents a plugin that will be added to columns in a table. These will generally consist of one or more columns that can be added to a tabular layout.
- tool - A tool is a widget that provides additional functionality to the users. Generally a tool needs to be launched by a button or a menu item from the sidebar.

To package your created plugin to the tactic data directory, select the plugin and go to the manifest tab. Here, you can make sure that the plugin is named and versioned appropriately. You now need to make sure that the manifest you've wrote is exported, exporting saves the manifest data you have there to the manifest.xml file.You can now select Publish and TACTIC will package all the files and create a .zip file of the plugin folder from the root plugin folder (ie: <TACTIC_DATA_DIR>/plugins). When a version is published, the folder of the current plugin is taken and copied to a new folder with the name <PLUGIN_CODE>-<VERSION>. Note that the PLUGIN_CODE can have "/" to present folders.

5.7 Python Trigger in Tactic Editor Guideline

5.8 Naming Convention Classes

TACTIC has a default file naming convention that has proven to work for a wide variety of productions. A production facility may simply choose to use this default naming convention, or could also override it to match the convention used by its current system. Customizing the directory and file naming conventions has proven to be the most time consuming part of integrating TACTIC into a system. The difficulty required to do this depends largely on being able to access the directories and file names procedurally.

TACTIC allows you to define different project types in the Site Admin → Projects Types View. Here you can define a different type of project and set up the various types of naming conventions for a particular project. When creating a project, you select a project type and it will make use of the information in the project type.

The various naming conventionn are as follows:

1. file_naming_cls: this class determines the file name of every file checked into TACTIC.

2. dir_naming_cls: this class determines the directory of every file checked into TACTIC
3. app_naming_cls: this class determines the node names within an application such as Maya.

The following code snippet is an example of overriding the directory for all files checked into a shot:

```
from pyasm.prod.biz import ProdDirNaming

class CustomDirNaming(ProdDirNaming):
    def prod_shot(my, dirs):

        shot = my.sobject

        dirs = my.get_base_dir()

        # add the sequence code
        sequence_code = shot.get_value("sequence_code")
        dirs.append(sequence_code)

        # add the shot code
        shot_code = shot.get_code()
        dirs.append(shot_code)

        # put all files in the "scenes" directory
        dirs.append("scenes")

        return dirs
```

This will create a directory name that looks something like

`/<base_dir>/<sequence_code>/<shot_code>/scenes`

or

/sample3d/shot/XG/XG002/scenes

Overriding naming conventions is a simple matter of defining your own implementation class and implementing specific functions in this class. Each SObject has its own SObject type. For example a shot in a production may have the type "prod/shot". This naming uniquely identifies this type of SObject.

To customize the naming convention for this class, you replace the slashes "/" in the Search Type with underscores "_" and use this as the name of the function. So in the example above, to customize a Shot (prod/shot), you define a function called prod_shot. Whenever TACTIC is asked to produce a directory for a particular SObject, an implementation function such as this is called. If no such function exists, then the default is used.

get_base_dir() simply gets the base directory of this SObject (default <base>/<project>/<table>)

Overriding the file naming is similar.

```
from pyasm.prod.biz import ProdFileNaming

class CustomFileNaming(ProdFileNaming):
    def prod_shot(my):

        parts = []

        parts.append(my.sobject.get_code())

        parts.append('custom')
        parts.append(my.snapshot.get_context())
        version = my.snapshot.get_value("version")

        version = "v%0.3d" % int(version)
        parts.append(version)
        ext = my.get_ext()
        name = '_'.join(parts)
        name = '%s%s'%(name, ext)
        return name
```

This will create a file name that looks something like

`<shot_code>_<custom>_<context>_<version>.<ext>`

or

`XG002_bedroom_anim_v004.jpg `

Custom in this case is a custom attribute added to a shot. So with these two classes, we would have a full path for this file of:

`/assets/sample3d/shot/XG/XG002/scenes/XG002_bedroom_anim_v004.jpg `

TACTIC comes with a default file and directory naming convention. You may choose to adopt this default naming convention as specified above, or you may create your own naming convention. The choice of which naming conventions should be used is often a hard one. Using TACTIC's default naming convention makes it much simpler and quicker to start using TACTIC in production. This is the recommended route if there is no legacy within the facility. If, however, you have many scripts and processes that rely on a previous naming convention, then you may customize TACTIC to map to your current naming convention.

The rest of this section describes TACTIC's default naming conventions.

To start, there is a base directory under which all asset files are stored. This base directory is specified in the Tactic conf file in <sites_dir>/config/tactic_linux.conf (tactic_win32.conf for windows). The next level is divided by project and then the type of the sobject. All projects of this same type are located under this directory:

`<base_asset_dir>`

The default for any search type checked into a specific context is represented with the following convention:

The next levels represent the subdirectory component and are all associated with metadata for the SObject types in some way. The details are up to the implementation function for each specific SObject type.

```
<base_asset_dir>/<project_code>/<search_type>/<sobject_code>/<sobject_code>_< ↩
    snapshot_context>_<snapshot_version>.<original_file_ext>
```

Default

If an SObject type does not have any overriding function, then there is a default implementation:

Subdir: empty File: <filename>_<file_code>.<ext>

```
    Subdir: empty

    File: <filename>_<file_code>.<ext>

    example: /home/apache/assets/storyboard/castle01_00034355BAR.jpg
```

The file code ensures that the file name is unique. This uniqueness prevents files from overwriting each other, even when files of the same name are checked in. In recent versions TACTIC has moved away from adding the file code to the file name in favor of the clearer v002_BAR ending. (However, the file name format can still exist for numerous asset types where the file name is of little consequence.)

5.9 Create a Plugin

A plugin is a self-contained package of files that TACTIC can make use of to extend the base functionality. Virtually any functionality in TACTIC can be made into a plugin.

A plugin can contain:

- project configuration data

- any database data
- js files
- css files
- documentation
- python files

The manifest file is a description of the entries in the database that are owned by the plugin. This allows the plugin manager to extract the appropriate database entries and commit the .spt files. It contains elements like:

data: a collection of name/value pairs that describe information about the plugin

- code
- description
- version

sobject: describes which sobjects the plugin contains. It's an expression of the form <sobject search_type="[search_type]"> with attributes:

- code: the specfic code of the object
- expression: an expression of which all matched sobject will belong to the plugin
- path: the relative .spt file path that all sobjects will be written to
- ignore_columns: a comma seperate list of columns for the plugin exporter to ignore
- There are some special attributes for specific search types. The config/widget_config search type has the attribute:
  - view

spt files are database files that contain database schema structure and database data. These files enable TACTIC to read and write database data that is both platform and database independent. This abstractions allows TACTIC plugins to be used on any supported TACTIC platform. An important design criteria of .spt files are that they are human readable even when the database entry contains xml or software code. More importantly, they can be easily diff'ed using standard software tools so that the code produced can show proper diffs using any source code management system (such as Perforce, SVN or Git). This is essential for collaborative work building plugins to delivery to a 3rd party.

Once you are in the plugin manager, you can the New button which creates a new plugin outline. Afterwards, you can start filling in the details like name, type, etc. On creation, a plugin type can be specified. Depending on the plugin type a number of bootstrap data will be created to support the structure of the plugin. After selecting Create, the plugin will be created and you will be able to see it in the plugin list.

If you go to the documentation tab, you will find that you are able to create new documentation if the documentation doesn't exist. This will create a new file, doc.html, which you can edit now.

To add files to the plugin, select the plugin and go to the files tab. Here, you will find many options like the ability to upload or simply create a new file. The new files that you are uploading or creating are used properly when their purpose is explained in the manifest.xml file.

After customizing the plugin to your needs, you can now package the plugin to perhaps upload to the community site so others can use it. Documentation on packaging can be found in this section under Packaging a Plugin.

Widget config tables should not include code or id columns or they must be explicitly set to values that are guaranteed to be unique on any installation of TACTIC. Otherwise, the plugin should not depend on the value of the code or id column.

This is also true of "custom_scripts" written in the script editor.

When referring to an sobject, always search by code (not id). When doing this, make sure the code contains a namespace that will not conflict with any other plugin.

5.10 Widget Development

As of 2.5, all widgets are derived from BaseRefreshWdg. This refresh widget is a new style widget which has some added functionality allowing to to be "smart" enough to refresh itself. It also standardizes the interface for passing construction parameters to the widget. All new style widgets take kwargs (keyword arguments) as argumets to the constructor

```
widget = MyWidget(option1=value1, option2=value2)
```

All new style widgets defined a method called "get_args_keys", which return a dictionary of defined and allowable arguments:

```
def get_args_keys(my):
  return {
    "option1": "this is option #1",
    "option2": "this is option #2"
  }
```

TACTIC provides the ability to create your own widgets and integrate them seamlessly into the TACTIC interface.

There are 3 main types of widgets:

*Widget:*A widget derived from a the base Widget class is a free standing widget that requires no parent widget.

*Table Element Widget:*An element widget is a widget that needs expected to be put inside a TableLayoutWdg.

*Input Widget:*An input widget is a widget that requires one or more values to put entered or extracted.

Create your own custom widget

You can create your own custom widgets in Tactic that become completely integrated in the user interface.

All widgets are derived from the base Widget (pyasm.web.Widget) class. This class defines the fundamental functionality required for all widgets that appear in TACTIC. To create your own widget, you can derive off of this class.

Hello World

In order to start showing how custom widgets can be created, we will start with the base "Hello World" widget. Create a folder called "custom" and then create a new file called "hello_world_wdg.py" in this new folder. In the file add the following lines:

```
from pyasm web import Widget

class HelloWorldWdg(Widget):
    def get_display(my):
        return "Hello World"
```

In order for TACTIC to be able to use this class,TACTIC must be able to see this file: this "custom" folder must be either in the PYTHONPATH or in sys.path of the TACTIC process (you can alternatively, use any class that complies with Python’s module handling.

> **Note**
>
> You can also use the python_path variable in the TACTIC config file to add paths to the sys.path dictionary

In order to view this widget quickly, you can open up the javascript editory and type:

```
spt.panel.load("custom.hello_world_wdg.HelloWorldWdg");
```

and press the "Run" button. You should see the following:

image

Note that the title does not change. This is something that the link will do automatically.

Formatting the Widget

We could format the widge a litlle more using some basic HTML widgets.

```
from pyasm.web import Widget, DivWdg

class HelloWorldWdg2(Widget):
    def get_display(my):
        top = DivWdg()
        top.add_style("font-size: 15px")
        top.add_style("margin: 30px")
        top.add_style("padding: 30px")
        top.add_style("width: 150px")
        top.add_style("text-align: center")
        top.add_style("border: solid 1px black")

        top.add("Hello World")

        return top
```

Adding this to a file called hello_world_wdg2.py and then in javascript editor, type:

```
spt.panel.load("custom.hello_world_wdg2.HelloWorldWdg2");
```

Pressing the "Run" button gives:

image

HTML

Here we introduce the basic HTML widget DivWdg. The add_style() allows you to add arbitrary CSS styles to the widget. There are various operations that can be added to HTML widgets that are useful for formatting the layout of the page. These methods include:

- set_attr(name, value)
- add_style(name, value)
- add_class(css_class)
- add_event(event, js_action)

There are few useful predefined widgets that sit on top of HtmlElement:

- DivWdg
- SpanWdg
- Table

These are all based of of HtmlElement which are basic html elements and provide a thin layer above HTML. HtmlElement also defines a number of static constructors to address most HTML elements:

- HtmlElement.br()
- HtmlElement.p()
- HtmlElement.br()

These return variations of HtmlElement that represent the different HTML elements. These are useful for laying out a complex widget. All HTML elements and their properties are accessible from these.

Using other widgets

You can add other predefined widget, for example, the CalendarWdg

```
from pyasm.web import Widget, DivWdg

from tactic.ui.widget import CalendarWdg

class HelloWorldWdg3(Widget):
    def get_display(my):
        top = DivWdg()
        top.add_style("font-size: 15px")
        top.add_style("margin: 30px")
        top.add_style("padding: 30px")
        top.add_style("width: 200px")
        top.add_style("text-align: center")
        top.add_style("border: solid 1px black")

        top.add("Hello World")

        calendar = CalendarWdg()
        top.add(calendar)

        return top
```

Adding this to a file called hello_world_wdg3.py and then in javascript editor, type:

```
spt.panel.load("custom.hello_world_wdg3.HelloWorldWdg3");
```

Pressing the "Run" button gives:

image

This adds one of the predefined widget "CalendarWdg". Widgets are hierarchical and can be added to other widgets. Any widget can embed any other widget within it's display. This provides a very flexible archictecure for building up complex hierarchical widgets.

Create your own table element widget

There is a special class of widgets that are designed to be used in conjuntion with TableLayoutWdg, the primary widget used for laying out tabular data. These widgets should be derived from BaseTableElementWdg, which extends the basic Widget class with a number of specific methods.

The TableLayoutWdg uses it's child widgets slightly differently than most widgets. It creates a single widget for each column and calls the get_display() method repeatedly for each row; each row representing a single sobject. Each element widgets does have knowledge of all of the sobjects, however, for each row, there will be a current sobject set. This means that the widgets get_display() method will be called repeatedly for each row. So, instead of operating on a list of widgets, the table element widget should get the current widget using the "get_current_widget()" method.

The following is a simple example of a table element widget.

```
from pyasm.web import DivWdg
from tactic.ui.common import BaseTableElementWdg

class MyElementWdg(BaseTableElementWdg):
    def get_display(my):
        sobject = my.get_current_sobject()
        first_name = sobject.get_value("first_name")
        last_name = sobject.get_value("last_name")
        div = DivWdg()
        div.add("%s %s" % (first_name, last_name) )
        return div
```

The class is almost identical to a regular class, except that it is derived from BaseTableElementWdg and that it uses get_current_sobject() to get the current sobject being drawn. This widget still has access to all of the sobjects in all of the rows, through get_sobjects(), if this is necessary.

To test this, save the code above in a file called my_element_wdg.py and enter this into the javascript editor:

Note

This only works in 2.6: in 2.5, you have to create the view in the widget config table

```
var config = " \
<config><test>  \
<element name='name'>  \
  <display class='custom.my_element_wdg.MyElementWdg'/> \
</element> \
</test></config>";

var args = {
  'search_type': 'sthpw/login',
  'view': 'test',
  'config_xml': config,
  'do_search': 'true'
};
spt.panel.load("main_body", "tactic.ui.panel.TableLayoutWdg", args);
```

Pressing the "Run" button gives:

image

Your custom table element widget completely integrates within the TACTIC interface. You can add other widgets by expanding the config definition.

```
var config = " \
<config><test>  \
<element name='preview'/>  \
<element name='name'>  \
  <display class='custom.my_element_wdg.MyElementWdg'/> \
</element> \
<element name='email'/>  \
</test></config>";

var args = {
  'search_type': 'sthpw/login',
  'view': 'test',
  'config_xml': config,
  'do_search': 'true'
};
spt.panel.load("main_body", "tactic.ui.panel.TableLayoutWdg", args);
```

This adds a preview and an email column (which are predefined for sthpw/login search type) and appear with your custom widget.

image

BaseTableElementWdg

This example describes how to create your own BaseTableElementWdg to execute a server-side command. The user can type some words in the text field, and then click on the "Action" button. The words will be written as the content of a file in the /tmp folder of the server. In the tactic config file, tactic_linux-conf.xml, let's say the python_path is */home/apache/custom*. You can create a file called custom_wdg.py and *init*.py in it.

image

Here is the content of*init*.py:

```
from custom_wdg import *
```

Here is the content of custom_wdg.py:

```
__all__ = ['CustomToolElementWdg','CustomCmd']
```

```
from tactic.ui.common import BaseTableElementWdg
from tactic.ui.widget import ActionButtonWdg
from pyasm.web import HtmlElement, SpanWdg
from pyasm.widget import TextWdg
from pyasm.command import Command

class CustomToolElementWdg(BaseTableElementWdg):

    def get_display(my):

        top = DivWdg()
        top.add_class('spt_custom_tool_top')
        text = TextWdg('user_input')

        action_button = ActionButtonWdg(title='Action', tip='Write a file in /tmp based on  ↩
            the data in the text field')
        action_button.add_behavior({'type':'click_up',
            'cbjs_action': '''var server = TacticServerStub.get();
            try {
                var top = bvr.src_el.getParent(".spt_custom_tool_top");
                var values = spt.api.get_input_values(top, null, false);

                # this path is assumed importable in your Python environment
                server.execute_cmd('custom_wdg.CustomCmd', values);
            }
            catch(e) {
                alert(spt.exception.handler(e));
            }
            '''})

        top.add(SpanWdg('Input:', css='small'))
        top.add(text)
        top.add(HtmlElement.br())
        top.add(action_button)

        return top

class CustomCmd(Command):

    def execute(my):
        text = my.kwargs.get('user_input')
        f = open('/tmp/my_file.txt','w')
        f.write(text)
        f.close()
```

If you click the first "Action" button, a file with "Hello !!!" will be created. On clicking the second "Action" button, the file content will be replaced with the word "example".

5.11 Custom Layout Editor

image

The Custom Layout Editor allows you to have complete control over the look and feel of TACTIC using many of the standard web technologies (HTML, CSS and Javascript). With this tool, you can build your own TACTIC components (called widgets) that have the ability to interact with one another intelligently, making it easier for you to design your very own TACTIC interface.

Custom Layouts enable the laying out of custom widgets using standard HTML.

TACTIC Custom Layout introduces a new html tag <element> which lets TACTIC widgets to be embedded into HTML.

There are two formats for a TACTIC element: a short form and a long form:

short form:

```
<element view='forms/my_form'/>
```

long form:

```
<element>
  <display class='tactic.ui.panel.CustomLayoutWdg'>
    <view>forms/my_form</view>
  </display>
</element>
```

This ability to reference other views and elements makes it easy to keep a top level view that draws from other views.

For display class names of other widgets, see section on Common Widgets.

You can create styles for each view in the Styles tab. However, most of the time it will be useful to reference a central stylesheet for a number of views.

In order to include a top level stylesheet, you can create an empty view with only styles defined and include these styles into other top level views, just as how you would reference a normal view.

For example, you can create a view called *common/styles* and add this line to the HTML of a view where you want the styles to appear.

```
<element view='common/styles'/>
```

TACTIC's behavior system makes use of standard JavaScript behaviors with the added functionality of some built-in classes.

Here are two ways to add an alert behavior to a button class called *my_button*.

```
<behavior class="my_button" event='click_up'>
alert('Hello World');
</behavior>

<behavior class='my_button'>{
'type': 'click_up',
'cbjs_action': "'

alert('Hello World');

"'
}</behavior>
```

Here are the types of events that the TACTIC behavior system has built-in support for:

```
click_up | click | wheel | double_click | drag | hover | move | change | blur | mouseover | ↩
     mouseout | keyup | keydown | listen
```

You can set the behavior class to activate upon the firing of another event using the *listen* type event.

```
<behavior class='my_button'>{
'type': 'click_up', 'cbjs_action': "'
spt.named_events.fire_event('my_event_trigger'); "'
}</behavior>

<behavior class='my_class'>{
'type': 'listen',
'event_name': 'my_event_trigger',
'cbjs_action': "'

alert('Hello World');

"'
}</behavior>
```

When the behavior is applicable to a specific HTML element (eg. click, click_up, mouseover, etc.), you can get element for which the behavior originated from using the *bvr.src_el* (Behavior Source Element) tag.

```
var table = bvr.src_el.getParent('.my_table');
var cells = table.getElements('.my_cells');
cells.setStyle('background', 'red');
```

TACTIC's powerful framework comes with many API functions that make developing for TACTIC easier. Here are some common ones.

Show loading popup:

```
spt.app_busy.show('Saving data...')
```

Hide loading popup:

```
spt.app_busy.hide()
```

Load an element:

```
spt.panel.load(element_name, class_name, kwargs)
```

Load an element into a popup:

```
spt.panel.load_popup(element_name, class_name, kwargs)
```

Close a popup:

```
spt.popup.close(popup_element)
```

The Custom Layout Editor's Options tab incorporates the administrative ability of modifying column, theme, view and table definitions with the ability of creating these widgets. Various views, themes, columns and tables can be created in the Custom Layout Editor with the desired content and appearance.

Options of these custom features can be defined in the Options tab, similar to how a predefined column or Column Manager custom column is modified through the Edit Column Definition window. These options can be set to change widget appearances, functions and behaviours, often in addition to any options established in the HTML and Python tabs. Many of the same options provided in the Edit Definition window are available and can be used to customize created features.

The custom feature options are set through the use of Python in the Options tab. A variety of different examples written in Python have been shown below. They are structured exactly as they would need to be defined in the Options tab with all possible subsets to refine a particular option.

Option Tab Examples

```
    "mode": {
        'description': "Determines whether to draw with widgets or just use the raw data",
        'type': 'SelectWdg',
        'values': 'widget|raw',
        'order': 00,
        'category': 'Required'
    },
    "search_type": {
        'description': "search type that this panels works with",
        'type': 'TextWdg',
        'order': 01,
        'category': 'Required'
    },
    'expression': {
        'description': 'Use an expression to drive the search.  The expression must return ↩
            sObjects e.g. @SOBJECT(sthpw/task)',
         'category': 'Display',
         'type': 'TextAreaWdg',
        'order': '01'
    },
    "element_names": {
        'description': "Comma delimited list of elemnent to view",
        'type': 'TextWdg',
        'order': 0,
        'category': 'Optional'
    }
```

Customized options are also available to the user which offer additional flexibility when modifying custom features. Some examples have been provided below.

Option Tab Custom Option Examples

```
    {
    'basic_option': 'You can describe your option here',
    'advanced_option': { 'description' : 'You can describe your option here and in type  ↩
        specify what type of edit widget is used to display the option.',
                         'category': 'Display',
                         'type': 'TextWdg'
    }
```

Images can be checked into TACTIC and used in interface design. In the Files tab, you can check in images using the Check-in wizard.

Once the file is checked in, you use the web path as the URL of the image.

TACTIC integrates the efficiency of Python in a Custom Layout Editor tab, functioning together seamlessly with HTML, CSS and JavaScript. As can be seen in the example below, Python works in connection with these other languages to produce a desired output.

Example

HTML code in Custom Layout Editor HTML tab:

```
<div class="hello_world">
    Hello World
</div>
```

CSS code in Custom Layout Editor Styles tab:

```
.hello_world {
    font-size: 2em;
    padding: 1px;
    border: solid 1px black;
    background: ${background};
}
```

Python code in Custom Layout Editor Python tab:

```
background = "#F00"
```

Custom Layout Editor Test Output:

The example demonstrates the use of Python in the creation of a title block. Essentially, the Python coding is only setting the background color of the title block. However, it can be used for more complicated applications, as in the Testing Interface section.

The Python tab provides the user with the opportunity to use the Python language in the editor without the restriction of having to explicitly use embedded Mako, which is another TACTIC feature that is described in the following section. The dedicated Python tab is an implicit integration of Mako. Both Mako and the Python tab essentially use Mako templating, while both providing equivalent Python utility and efficiency.

The custom layout engine embeds the Mako, a powerful templating engine which allows you to embed Python scripts and logic within HTML. In order to embed Python coding within the HTML, the code must be surrounded by the special Mako tag: <% {Python Code} %>. Here is a simple example of its usage:

```
<div>
<%
my_car = 'A ferrari'
%>
</div>
<p>${my_car}</p>
```

Mako makes passing and accessing of data in TACTIC easy, especially combined with the support of XML by TACTIC widgets for passing arguments.

The *kwargs.get* function can be used to get the value of an XML attribute of an element, whether it is an attribute already supported by the element or an arbitrary one. Here is an example of setting a value for an arbitrary attribute.

HTML code in top level view:

```
<element>
  <display class='tactic.ui.panel.CustomLayoutWdg'>
    <view>my_forms.photoshoot_form</view>
    <args>Hello</args>
  </display>
</element>
```

HTML code in a view named *my_forms.photoshoot_form*:

```
<element>
  <display class='tactic.ui.input.TextInputWdg'>
    <default>${kwargs.get("args")}</default>
  </display>
</element>
```

For the example above, the text field will be populated with the string *Hello*.

Most of the time, it will be beneficial to use Mako to pass search keys from one view to another. That's covered in a bit more detail in the Creating Forms section of this document.

You can inject your custom widgets or TACTIC built-in widgets into your view through the user interface. You can do it through the gear menu:

image

All these injection options allow you to inject the widget you want directly where your cursor is in the code. All these injections have the name field in common. The name field allows you to name your widget in case you want to refer to it later in the code.

Inject Widget allows you to inject **any** widget you want. You need to define which widget to inject. You can select your widget through the dropdown or select classpath and write the class path of a built-in TACTIC widget. After selecting, a built-in widget, you may have to fill in additional arguments which are required to successfully run the widget.

image

Similarly, you can inject a thumbnail, video, table, calendar, etc. and customize the options respectively. The more popular widgets have been added for your convenience to the gear menu. They are listed below.

Inject Thumbnail injects a thumbnail widget.

Inject Video injects a video.

Inject Text Input injects the text input field widget. You can specify many options like the width of the input field.

Inject Look Ahead Text Input is similar except there is a look ahead which comes with the input field.

Inject Layout injects a View Panel Widget.

Inject Table injects a Fast Table Layout Widget.

Inject Calendar injects a Calendar Widget.

Inject Search injects a Global Search Widget.

Inject Subscription injects a Subscription Bar Widget into your HTML.

image

You can add the view you have created directly to the sidebar. To do this, click on the gear in the top menu and select “Add to Side Bar”. This will add this view to the sidebar under the Project Views. By default, It will get named according to the view name and “/” will be treated as a space. For example, “app/chart” will be named “App Chart”. You can always rename these views in the sidebar by right clicking on them and selecting “Edit Side Bar”. Now select the view you want to edit and change the Title field.

When you startup tactic and go to the main project URL (. . . /tactic/<project_name>), you are presented with the tactic homepage of the project. That tactic homepage URL can be changed to show one of your created views. To do this, open up your view in the custom layout editor, then from the gear menu select “Set as Project Url”. The current view you have open will be shown when you go the main project URL. You can come back to admin side of tactic by adding “/admin” to the URL (. . . /tactic/<project_name>/admin).

You can also turn your view into a custom URL. This means that your view will open when you go to a specific URL. To do this, open your view in the custom layout editor, then from the gear menu select “Add as Custom Url”. This will open up a dialog box where you can specify what URL should open up the view. The URL specified there is showing the URL which is appended to (. . . /tactic). You can specify which widget to run in the URL in the widget field. By default, it shows the widget code for the view that was open in the custom layout editor. You can check all your custom URLs by going to the gear menu and selecting “Show Custom URLs”. This will show all the existing custom URLs. This is where you can delete existing custom URLs.

Forms provide an interface for updating TACTIC data. The Custom Layout Editor makes the creation of forms easy with built-in widgets and functions.

TACTIC already has some predefined input widgets that can be used as input fields for forms, and they are referenced like any other widget.

TextInputWdg

SelectWdg

TextAreaWdg

CalendarInputWdg

ActionButtonWdg

```
<element name='my_text_input_field'>
  <display class='tactic.ui.input.TextInputWdg'>
    <default>Hello</default>
    <width>100px</width>
  </display>
</element>
```

You can find more details on the exact XML attributes that are supported by each widget in the Common Widgets section.

Here are some useful functions for generating forms.

```
spt.api.get_input_values(div_container)
```

This gets the values of the all the input fields of a div as an array with the attributes being the names of the element names.

```
server.update(search_key, data)
```

This updates an sobject with data that is passed in as an array.

The search key is a key that uniquely identifies an sobject.

Here is an example of usage of both for updating a TACTIC task through a form.

In this example, the search key of an sobject is passed into the view through a list of keyword arguments, and it is kept as a hidden input for ease of access. The clicking of the save button activates the behavior for saving the form.

```
HTML: <div class='spt_form'>
  <input type="hidden" name="spt_search_key" value="${kwargs.get('search_key')}"/>
    <element name='spt_status'>
      <display class='SelectWdg'>
        <values>Assigned|Pending|Approved|Waiting</values>
        <search_key>${kwargs.get("search_key")}</search_key>
      </display>
    </element>

  <input type="button" class="spt_save_button" value="Save >>"/>
</div>

JavaScript:

<behavior class="spt_save_button> {
"type": "click",
"cbjs_action": '''
  //gets the parent of the behavior source element
  var top = bvr.src_el.getParent('.spt_form');

  //gets all the input values
  var values = spt.api.get_input_values(top);

  var data = {
    //gets value of element named 'spt_status'
    //sets it as the value of the 'status' column for the task sobject
    status: values.spt_status;
  }

  search_key = values.spt_search_key;
  server.update(search_key, data) '''
}
```

You can customize your views to behave during a testing phase. To do this, you can add a condition in your code to check whether the code is being run in testing mode. You can use the following condition in the python section of the code:

```
if kwargs.get("is_test") in [True, 'true']:
```

This condition will be true if it is testing mode. You can now use this condition to setup your variables correctly. You can run the view in testing mode by clicking the test button in the top menu.

image

In addition, these custom views can also be defined to take the form of a specific view type: widget, column, chart, report or dashboard.

Widget is a free form view type. It is designed to allow your view to be versatile in its presentation in TACTIC. These widget type views can appear in stand-alone pop-up windows, tab view layouts, forms, tables, text inputs, menus, or even buttons. An extension of this versatility lies in their ability to be injected into other custom views as well, much like what was described in the Injecting Widgets section.

Column allows for the customization of a table column. This column is available to be added to any table through the Column Manager under Plugin Widgets. The ability to modify the aesthetics and data presentation of a column will allow the user to display the data they want in a specific format. For example, if metadata for a particular asset, such as an image, needs to be shown in a table, a column can be formatted to list the metadata information through the use of an embedded table, as shown below. Other formatting techniques, like alternating row colors, can be added as well.

Example

HTML code in HTML Custom Layout tab:

```
<div>
<%
```

```
table = []
table.append("<table cellpadding='4px'>")
i = 0
for name, value in data.items():
    i = i + 1
    if (i % 2 == 0):
        table.append("<tr style='background: #BBB'>")
    else:
        table.append("<tr style='background: #DDD'>")

    table.append("<td style='width: 100px'>%s</td>" % name)
    table.append("<td>%s</td>" % value)
    table.append("</tr>")

table.append("</table>")
table = "".join(table)

context.write(table)
%>
</div>
```

Python code in Custom Layout Python tab:

```
data = {
    'Frame Size:': "768 x 512",
    'Colorspace:': "yuvj420p",
    'Location:': "C:\...\",
    'File Type:': "MJPEG (Motion JPEG)",
}
```

Output Column in Table:

Chart, report and dashboard view types present the ability to create customized charts, reports and dashboards that are easily accessible to users. Custom views can be defined to present data in the form of line charts, budgeting reports or department specific dashboards that display tables or views that are commonly used by that department.

However, only by specifying the view type do these views become easily accessible to users. For example, by setting a custom layout view to Chart, in the Examples section under Project Configuration, this custom view will be added to the list of charts that are already available. By selecting the drop down arrow menu on the created chart icon, there is an option to "Add to Side Bar". This will add this chart view to the side bar for easy accessibility, under a defined "Chart" section.

In addition, if a custom theme is created and utilizes sidebar views and associated links in the menu, this new chart view will be automatically added in the menu of theme.

To begin a custom chart, report or dashboard, going to the Examples section under the Project Configuration is a good place to establish a base to build one of these customized tools. For example, by selecting Dashboards in the Examples section, a selection of different sample dashboards will be displayed. If one of the dashboards is of particular interest, but requires some modification, by going to the arrow drop down menu on the dashboard icon and selecting "Show Definition", the definition can be copied and pasted in the Custom Layout Editor and modified to the desired appearance.

Handling None:

The default value for the empty string in Python is the word "None". This does not help very much when you want to obtain something like the search key of an sobject because if there is no search key, instead of getting an empty string, you get the string "None". And if you try to pass "None" into an element, an error will likely result.

The way to work around that is to add an "or" at the end of your kwargs.get function.

```
ie: kwargs.get("search_key") or ""
```

Embed Elements:

A shortcut for embedding elements into the HTML is by clicking on the gear menu.

Similarly, if you would like to inject another view into your current view, you can do so by right clicking on the view you want to inject.

Element Name as Column of sObject:

If you pass a search key into an element, it automatically takes the element name as the column if you do not specify one. In the example below, the text input will display the id of the sObject with the given search key.

```
<element name="id">
  <display class="tactic.ui.input.TextInputWdg">
    <search_key>${search_key}</search_key>
    <width>100px</width>
  </display>
</element>
```

5.12 Triggers

Triggers are callbacks that are named based on certain events. TACTIC provides three types of triggers which allow you to add to existing functionality.

- **Event-based triggers**: triggers based on specific events that occur within TACTIC. During the execution of a command in TACTIC, various named events may be called. Any one of these events may trigger other actions, if a custom trigger is registered to that event.
- **Server-side pipeline triggers**: triggers defined in a server-side pipeline. They are called as a result of events that occur in the pipeline itself
- **Client-side pipeline triggers**: triggers defined in a client-side pipeline. They are defined in the client API.

As TACTIC runs through its code, it will periodically call named events. These named events provide a mechanism for attaching custom trigger handlers.

There are two styles of named event triggers supported by TACTIC.

The first style of event-based trigger makes use of the client API. The functionality in the client API can be accessed by the server code and is often preferable for third parties to use because it uses a well-defined interface much easier to program in than the complex server code. To create your own custom trigger, create a new class derived from the Handler class and override the execute function:

```
from tactic_client_lib import TacticServerStub
from tactic_client_lib.interpreter import Handler

class CustomTrigger(Handler):
    def execute(my):
        # get a handle to the server stub
        server = TacticServerStub.get()
        server.start("Starting server transaction")
        try:
            # at this point, you have full access to the server using the client API
            ret_val = server.ping()

            # get values from the inputs
            search_key = my.get_input_value("search_key")
            sobject = server.get_by_search_key(search_key)
            if sobject.get('asset_library') != 'character':
                return

            # check to see that the status has changed
            update_data = my.get_input_value("update_data")
            if update_data.get('status'):
                do_something_interesting(sobject)
        except:
            server.abort()
        else:
            server.finish()
```

A reference to the TacticServerStub can be accessed through the static method get(). Once a reference to the server stub is obtained, it is possible to make use of the client API functionality. The main difference is that this code is being run inside the TACTIC server process, so the overhead of XMLRPC is not present. Thus triggers running on the server side will run much faster that those running using the XMLRPC protocol.

It is also possible in the trigger to access another TACTIC server by using the TacticServerStub and explicitly setting the three settings required to connect to another server. For example, here is some code to synchronize the asset list:

```
    server = TacticServerStub()
    server.set_server("tactic2.com")
    server.set_ticket(ticket)
    server.set_project(project)
    server.start("Synchonizing data")
    try:
        search_key = my.get_input_value("search_key")
        update_data = my.get_input_value("update_data")
        server.update(search_key, update_data)
    except:
        server.abort()
    else:
        server.finish()
```

Synchronization of data between two TACTIC servers is possible once authentication is set up. (Note that some priviledged knowledge about the remote server is required in order to authenticate.)

The second style of event-based trigger is driven from the class pyasm.command:

```
from pyasm.command import Trigger
class CustomTrigger(Trigger):
    def execute(my):
         print "executing custom trigger"
```

This trigger style makes use of server-side code and is much more complex to use. It is most often used internally and should generally not be used unless required due to a limitation in the client API.

As TACTIC server code is executed, triggers will be called periodically. TACTIC will call named events, which will then trigger registered handles that are listening to those events.

To better understand the event system, please review the **TACTIC Setup→ Project Automation → TACTIC Event System Introduction** documentation

Each of the handlers for the events listed above get an "input package" delivered to them. This input package contains information that is useful to the handler as determined by the command that called the trigger.

Table 1: Insert / Edit Input Values

param	**description**	**type**
is_insert	specifies whether a particular trigger was an insert or an edit	Boolean
search_key	the search_key of the SObject operated on by the insert/edit	String
prev_data	a dictionary of previous values of attributes that were changed	Dictionary
update_data	a dictionary of updated values of attributes that were changed	Dictionary

In order for a trigger to listen to an event, it must be registered in the trigger search type.

In the TACTIC admin site: http://<server_name>/admin, click on the **triggers** view. This view defines a list of triggers and the events they are registered to.

When you insert a new trigger, you specify the full class path of your new trigger, along with a description and the event that the trigger should listen for.

Time-based triggers allow you to execute custom code on the server at either specific intervals or at a specific time of the day. These are very useful triggers that allow you to handle any number of different actions.

- Backup (although this may be better done with a dedicated backup system)
- Cleanup
- Autobuilding of files
- Statistics gathering
- Data synchronization

In this example, the function get_execute_interval(), used to determine the intervals during which this trigger will be run, is overridden to 3600. This trigger will be run every hour (60*60) seconds. (The shortest hard coded interval is every 60 seconds. If you set a smaller number it will still execute once every 60 seconds.)

```
class SampleTimedTrigger(TimedTrigger):
    def get_execute_interval(my):
        '''return number of seconds between execution'''
        return 3600

    def execute(my):
        print "doing a bunch of stuff"
        print "sleeping"
        time.sleep(15)
        print ".... done"
```

In order for TACTIC to recognize this trigger, it has to be registered in the list of triggers in the Admin site. All timed triggers listen to the "timed" event.

5.13 Plugin Manager interface

image

The plugin Manager View is where you will be managing all your plugins. From this view, you can create a plugin, fully install a plugin, and modify existing plugins. You can find all your installed plugins in the plugin list at the left hand side of the view. This shows all the plugins you have installed along with all the built-in plugins which have come with your TACTIC installation. After selecting a plugin, you have access to:

Plugin Info:

- Name
- Code: This is an important entity
- Version: Imporant when you are planning to use or create multiple versions of a plugin
- Description

Documentation:

- shows all the documentation which has been provided for the plugin from the developer

Manifest file:

- ability to export the manifest.xml
- ability to publish the plugin (more information on this in the documentation on creating a plugin)
- contains technical information about the plugin
- can find more information about this in the documentation about creating a plugin

Files: The files tab shows the raw folder structure and files of the plugin. From here, a number of file operations can be performed.

- Adding files (Uploading)
- Removing files
- Creating folders
- Renaming files

5.14 Custom Checkin Pipeline

Partial override

There are four points in the current Application check-in process that the developer can insert handlers to perform custom actions. These events are called checkin/pre_export, checkin/create, checkin/process, checkin/dependency.

Here is a plain pipeline:

```
<pipeline>
    <process name="model/>
    <process name="texture"/>
    <process name="shader"/>
    <process name="rig"/>
    <connect to="texture" from="model" context="model"/>
    <connect to="shader" from="texture" context="texture"/>
   <connect to="shot/layout" from="rig" context="rig"/>
    <connect to="rig" from="texture" context="texture"/>
    <connect to="shot/lighting" from="shader"/>
</pipeline>
```

If we want to intercept the model process checkin with before exporting occurs and the texture process before and after the export of the node occurs, we will have a pipeline like this:

```
<pipeline>
    <process name="model">
         <action event="checkin/pre_export" scope="client"
 class="pyasm.application.common.interpreter.MayaModelCheckinPreexport"/>
    </process>
     <process name="texture">
           <action event="checkin/pre_export" scope="client"
class="pyasm.application.common.interpreter.MayaTextureCheckinPreexport"/>
           <action event="checkin/process" scope="client"
class="pyasm.application.common.interpreter.MayaTextureCheckinProcess"/>
      </process>
    <process name="shader"/>
    <process name="rig"/>
    <connect to="texture" from="model" context="model"/>
    <connect to="shader" from="texture" context="texture"/>
    <connect to="shot/layout" from="rig" context="rig"/>
    <connect to="rig" from="texture" context="texture"/>
    <connect to="shot/lighting" from="shader"/>
</pipeline>
```

The class attribute can point to a custom python path, usually accessible on the network where the client computer is on. This python class can do something as simple as adding a cube and parent it to the to-be-exported node, the scene file is free from user-created junk nodes, or making sure a certain special node exists in the scene file. Please to the full override section for some python class examples. The main method required is just execute(). And presumably you will import the application's python module to do the manipulation desired. For Maya, you would run this to create a cube:

```
import maya as cmaya
cmaya.cmds.polyCube()
```

Full override

Warning
This method requires more set-up on the developer's end as it does not leverage the exisiting application checkin functionalities. This section describes how to customize the checkin pipeline, which is a series of processes, each with an action handler defined.

Normally TACTIC handles many of the details for checking in files. However, this process can be completely taken over and customized.

An example checkin pipeline might look like the following:

```
<pipeline>
  <process name="validation">
    <action class="pyasm.application.common.interpreter.MayaModelValidate"/>
  </process>
  <process name="extractor">
    <action class="pyasm.application.common.interpreter.MayaModelCheckin"/>
  </process>
  <connect from="validation" to="extractor"/>
</pipeline>
```

This structure is the same for all pipelines defined in TACTIC. It describes a series of processes with actions. The actions have an attribute "class" that handles a particular part of the checkin process. TACTIC delivers a defined pipeline to a pipeline interpreter, which then executes the handlers in order. Handlers make use of the Client API to interact with TACTIC.

Note
For information on the Client API, refer to the **Client API Documentation**

A process handler is a function or subroutine that contains commands that are executed in response to an event. In TACTIC, all handlers are derived from the Handler class. This class defines a simple interface which has some basic functions which can be overridden:

execute()	The commands to be performed by the handler.
undo()	The method called when an exception occurs. TACTIC calls the *undo()* method for each handler in the pipeline in the reverse order that they were executed.

There are several helper methods you can use to set and retrieve information using handlers. Any particular handler has two sources of information:

1. **Package:** this data is global to all of the nodes. It is the dictionary data structure that TACTIC delivers to the client machine and includes such settings as status information and user interface selections. This data should be considered read-only.

You can retrieve package information using the method:

`get_package_value(my, key)`

where *key* is the name of the dictionary key for the data. The exact list of the keys delivered will depend on the user interface settings. 2. **Input:** this data is received from the previous process handler. The handler itself determines which input it receives.

You can retrieve input information using the method:

`get_input_value(my, key)`

Handlers can deliver these values to future nodes with output values, which become the input values for the next node. You can set output information using the method:

`set_output_value(my, key)`

The following sample is simple validation handler code that checks a Maya session for the existence of a particular node through its search key.

```
import maya.cmds as cmds

from pyasm.application.common.interpreter import Handler

class MayaModelValidate(Handler):
    def execute(my):
        # get the search key from the delivered package
        search_key = my.get_package_value("search_key")

        # get the sobject from the server
        sobject = my.server.get_by_search_key(search_key)
        if not sobject:
            raise Exception("SObject with search key [%s] does not exist" % \
                search_key)

        # code and verify in maya that the node is in session
        code = sobject.get('code')
        if not cmds.ls(code):
            raise Exception("Cannot checkin: [%s] does not exist" % code)

        my.set_output_value('sobject', sobject)
```

This code example, although simple, illustrates a number of handler interaction requirements.

`import maya.cmds as cmds`

This first line imports the standard Maya command libraries to allow the handler to interact with Maya.

`search_key =my.get_package_value("search_key")`

This line requires user input from a field in the interface on the search key (unique identifier) for a particular SObject.

`sobject =my.server.get_by_search_key(search_key)`

Using the search key obtained from the interface, this line uses the client API to retrieve data about the specific SObject. Handlers can access the server stub code by using the `my.server` prefix. All methods defined in the Client API are accessible through this type of reference. (See the Client API documentation for more information.)

The data structure returned is a dictionary of values that can be accessed as follows:

```
code = sobject.get('code')
if not cmds.ls(code):
    raise Exception("Cannot checkin: [%s] does not exist" % code)
```

The code then checks the Maya session to verify that a node exists with the same name as defined in the SObject. If not, an exception is created that halts the checkin process and informs the user with the appropriate error message that the checkin failed.

5.15 Performance

The TACTIC Client API interacts with teh server through an XMLRPC connection. This has a number of advantages and disadvantages that the developer should be aware of when programming the Client API. XMLRPC is a standard web sevice protocol built on top of HTTP. This means that the protocol is stateless. It also means that it requires an HTTP request for every interaction.

HTTP requests are very slow when compared to running code directly on the server, so care must be taken to minimize the number of interactions that occur between the client code and the server code. However, if a client side application is written with a few basic best practice guidelines, performance issues should not be a problem.

The TACTIC server should be treated as a special resource. The more client side processing you do, the lower the load on the server and the more scalable your client side application.

If possible, it is always preferable to pool queries into a single request with the use of proper filters: Unfortunately, this sometime sacrifices pure Object Oriented elegance, but it is a tradeoff that is well worth it in practice. For example, an object oriented approach to aquiring data would be:

```
shots = server.query("prod/shot", filters=[['sequence_code': 'XG']])
for shot in shots:
    tasks = server.get_all_children( shot.get('__search_key__'), 'sthpw/task')
```

When using this approach, a call to the server will be made for every shot. While, in principle, this will work, it could potentially be quire slow. A faster way to do this would be to get all of the tasks for all of the shots in a single statement:

```
shots = server.query("prod/shot", filters=[['sequence_code': 'XG']])
shot_keys = [ shot.get('__search_key__') for shot in shots)
tasks = server.get_all_children( shot_keys, 'sthpw/task')
```

This will get all of the tasks for all of the shots in one call to the server. Of course, some extra processing is required to relate the retrived tasks to the shot, however, this is all done on the client side and is executed very quickly.

```
tasks_dict = {}
for task in tasks:
    parent_key = task.get('__parent_key__')
    task_list = tasks_dict.get(parent_key)
    if not task_list:
        task_list = []
        task_dict['parent_key') = task_list
    tasks_list.append(task)
```

Creating this dictionary will enable rapid look up of the tasks for each shot.

Of course, this is done for you by providing the "return_mode" flag.

```
tasks = server.get_all_children( shot_keys, 'sthpw/task', return_mode='dict' )
```

By default, the return mode is "list", which just returns a flat list allow you to restructure as desired.

This applies to the more general "query" method:

```
tasks = server.query("sthpw/task")
```

5.16 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 javacript 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.

5.17 Navigating Search Type Hierarchy

Each project in TACTIC contains a collection of search types. The schema defines how these search types are related to each other. There is a wide variety of possible ways that two search types can be related to each other. The schema abstracts these relationships so that it is easy to navigate through these hierarchies.

The following relationship types are used:

- **parent_code**: The column named "parent_code" is used to define the parent code. You would need to look at the schema definition to know the exact search_type of each parent. This relationship type has the advantage that it standardizes the name of the parent column.
- **sobject_code**: A naming convention of <parent_table>_code is used to define the parent code. SObjects reference each other through the "code" column, which is guaranteed to be unique. (The code column is used instead of "id" because it is easier to read.) This is a more intuitive relationship type than "parent_code".
- **search_type**: The parent code is defined by an arbitrary relationship using two columns: search_type and search_id. Together, they uniquely identify parent SObjects.

- **search_key**: The parent code is defined by a single column called "search_key," which contains a unique identifier for the parent.

Of the above types, sobject_code and search_type are used most often. Any of these types can be used at any time and be related to each other. Having an intimate knowledge of these relationships can be confusing, so to keep things organized a project schema is used to define which search_types can be related to other search_types and in which ways. In other words, TACTIC uses the schema definition for the project to abstract relationships and make them easier to understand.

get_parent()

There are a number of methods to help navigate through the search type hierarchy.

Every search type can have a single parent type. You can query this type with get_parent_type(). For example, to find the parent type of a "prod/asset":

```
search_type = "prod/asset"
parent_type = server.get_parent_type(search_type)
print parent_type
```

When executed, the above code snippet would return the string "prod/asset_library".

get_child_types()

When the parent/child relationship is search_type or search_key, each SObject will have its own parent. In this case, the parent would return "*", which indicates that all search types are a possible parent.

To find child types, use the get_child_types() function. This function returns a list because a search_type can and will have a number of search types as children. This method will return all of the possible search types.

get_parent()

Most search_types will only have one parent type (except those that defer the parentage to the SObject itself). The get_parent() method allows you to obtain the individual parent SObject of an SObject.

```
search_type = "prod/asset"
code = "vehicle011"
search_key = server.build_search_key(search_type, code)
parent = server.get_parent(search_key)
print parent.get('code')
```

Executing the above code snippet would result in the output:

```
vehicles
```

because the parent type of "prod/asset" is "prod/asset_library" and the parent of "vehicle011" is the asset library "vehicles"

get_all_children()

Search types can and will have a number of child types. Some types defer the parentage to the SObject itself to determine the parent type. So when searching for children of parents, it is necessary to pass in a child type to narrow down the search. The options for child types can be found by the method get_child_types().

```
search_type = "prod/asset_library"
code = "vehicles"
search_key = server.build_search_key(search_type, code)
child_type = "prod/asset"
children = server.get_all_children(search_key, child_type)
for child in children:
    print child.get('code')
```

This code snippet will print out all of the codes of the children of this particular asset library, namely all of the assets in the asset library "vehicles."

get_all_children() can also be used to get snapshots (sthpw/snapshot) or tasks (sthpw/task) as well. These are special child types that defer the parent type to the individual SObjects.

```
search_type = 'prod/asset'
code = 'vehicle011'
search_key = seaver.build_search_key(search_type, code)
child_type = 'sthpw/task'
tasks = server.get_all_children(search_key, child_type)
```

This code snippet will obtain all of the tasks associated with vehicle011.

5.18 Widget Architecture

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 sobjects. Generally a search is performed to retrieve sobjects which are then used to draw the widget. The widget itself can perform the search or it can recieve sobjects 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 encompases 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 warrent 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 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 sobjects

\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.