SDK Cheat Sheet

This document is a cheat sheet for KBase Software Development Kit (SDK) module usage.

SDK Cheat Sheet

Dynamic services vs. SDK Methods (e.g. apps)

There are two flavors of SDK Module: dynamic services and SDK Methods, known to users of the Narrative interface as Applications or Apps.

SDK Methods are run asynchronously in a queue and are expected to have longer running times. SDK Methods can also call other SDK Methods (see Call one SDK module from another). SDK Methods are typically used for wrapping functionality from 3rd party code, uploaders and downloaders, and other long-running tasks. Almost all narrative applications are implemented as SDK Methods.

Dynamic services (DS) are SDK Modules designed to respond quickly to requests. As such, the module runs as an always-on service. DSs cannot call SDK Methods, but can call other services, dynamic or otherwise, as normal. DSs are typically used as the backend for UI elements (such as Narrative widgets) so those elements can be displayed and updated quickly. The function of a DS is often, for a specific workspace type (such as a KBaseGenome.Genome), to pull the data for an object and process the data into a form that the UI element can understand. The DS will often cache the processed form of the data and return parts of the processed form as the UI needs those parts. Thus the UI doesn't need to keep the entire data, processed or otherwise, in memory.

Creating a dynamic service

By default, all SDK modules are SDK Methods that run asynchronously. To mark a module as a DS, add the following to the kbase.yml file (example):

```
service-config:
    dynamic-service: true
```

You can then register your module as usual and start and stop it using the catalog interface (see Resources above). Otherwise development of a dynamic service module is identical to a Method module.

SDK Cheat Sheet

SDK Modules that are probably useful for your SDK module development

DataFileUtil

- Upload & download files to and from shock
    - Make handles
- Save objects to workspace
    - Automatically handles provenance
- Get objects from workspace
    - Simplifies interface
- Pack and unpack files with gzip, targz, and zip
- Package data for a downloader
- Copy and own shock nodes
    - Make handles

WsLargeDataIO

- Save and get objects to and from the workspace from files
    - E.g. doesn't load the object into memory

KBaseReport

- Aids in creating report objects returned by SDK methods.

SDK Cheat Sheet

Call one SDK module from another

We use python for the examples, but the principles are the same in other languages. Here we assume that the reader is familiar with how to read KIDL specifications and figure out the inputs to a function from the spec.

1. Find the module and function that you wish to call from your SDK module with the catalog function and module browsers
    a. https://narrative-ci.kbase.us/#catalog/modules
    b. https://narrative-ci.kbase.us/#catalog/functions
2. Note the status of the module - released, beta, or dev.
    a. The icons at the bottom of the function box provide a shortcut for this information.
3. Using the SDK, install the module client. From inside your module.
    a. kb-sdk install <module_name>
    b. If the module doesn't install, check the sdk.cfg file created in the root directory of your module. You may need to change the url for the catalog service to the ci, next, or appdev url.
    c. kb-sdk help install for more help.
4. Import the client in your code:
    a. from <module_name>.<module_name>Client import <module_name>
    b. Example
5. Setup the callback url in your constructor:
    a. self.callback_url = os.environ['SDK_CALLBACK_URL']
    b. Example
6. Initialize the client:
    a. cli = <module_name>(self.callback_url)
    b. Example
        i. Note that the example passes a token into the client, which is not actually necessary.
    c. To use an alternate release version, use the keyword argument service_ver:
        i. cli = <module_name>(self.callback_url, service_ver='dev')
        ii. Before releasing your module to production, remove any service_ver arguments.
7. Call the function:
    a. result = cli.<function_name>(input)
    b. Example

Note that only files located in the scratch space (in tests this is available via the test config [e.g. in python cls.cfg['scratch']) will be visible to both modules. In particular, files in the test/data folder will not be visible to the called function and must be moved to the scratch space at the

start of the test.

Error Handling

When an error in a SDK module occurs, it returns an error package to the client that called the SDK module, regardless if the client resides on a programmer's machine, in the Narrative, or is running in another SDK module that called the original module. The client then translates that package into an error class and throws the error. These errors contain a stacktrace for the error in the original SDK module, contained in the 'data' field of the error, that can then be viewed by the client as in the examples below.

Python

Example

Java

```
import java.net.URL;
import java.util.Arrays;

import us.kbase.common.service.ServerException;
import us.kbase.workspace.GetObjects2Params;
import us.kbase.workspace.ObjectSpecification;
import us.kbase.workspace.WorkspaceClient;

public class Temp {

    public static void main(String[] args) throws Exception {
       final WorkspaceClient ws = new WorkspaceClient(new
           URL("https://ci.kbase.us/services/ws"));
       try {
            ws.getObjects2(new GetObjects2Params().withObjects(
                 Arrays.asList(new ObjectSpecification()
                      .withRef("fake/fake/1"))));
       } catch (ServerException se) {
            System.out.println(se.getMessage());
            System.out.println(se.getCode());
            System.out.println(se.getName());
            System.out.println(se.getData());
            throw se;
       }
```

```
    }
}
```

Perl

```
use strict;
use Bio::KBase::Exceptions;
use Bio::KBase::AuthToken;
use Bio::KBase::workspace::Client;

my $ws = new Bio::KBase::workspace::Client(
      "https://ci.kbase.us/services/ws");
eval {
    $ws->get_objects2({"objects" => [{"ref" => "fake/fake/1"}]});
};
if ($@) {
    print "Exception message: " . $@->{"message"} . "\n";
    print "JSONRPC code: " . $@->{"code"} . "\n";
    print "Method: " . $@->{"method_name"} . "\n";
    print "Client-side exception:\n";
    print $@;
    print "Server-side exception:\n";
    print $@->{"data"};
    die $@;
}
```

SDK Cheat Sheet

Start and stop a dynamic service locally

First run kb-sdk test. This will set up a workdir in test_local like this:

```
$ tree test_local/workdir/
test_local/workdir/
├── config.properties
├── tmp
└── token
```

Create a run_container.sh file in test_local:

- Copy run_tests.sh to run_container.sh and make the following edits to the last line:
- Delete ‘test’ at the end of the file
- Delete ‘-e SDK_CALLBACK_URL=$1’
- Add ‘-d -p <external_port>:5000 --dns 8.8.8.8’

When you’re done it should look similar to this (obviously the module name will be different):

```
#!/bin/bash
script_dir="$(cd "$(dirname "$(readlink -f "$0")")" && pwd)"
cd $script_dir/..
$script_dir/run_docker.sh run --user $(id -u) -v
$script_dir/workdir:/kb/module/work -d -p 10001:5000 --dns 8.8.8.8
test/htmlfilesetserv:latest
```

Then run the script:

```
$ ./run_container.sh
c8ea1197f9251323746d9ae42363387381ee79f6c06cd826e6dbfba0a7fd703b
```

You can now interact with the service at the port you specified (in the example above, 10001).

SDK Cheat Sheet

To view logs, get the container ID with docker ps and run docker logs:

```
$ docker ps
CONTAINER ID          IMAGE                          COMMAND
CREATED               STATUS                PORTS
NAMES
c8ea1197f925          test/htmlfilesetserv:latest
"./scripts/entrypoint"   2 minutes ago      Up 2 minutes
0.0.0.0:10001->5000/tcp   gigantic_swirles

$ docker logs c8ea1197f925
2016-10-14 22:55:27.835:INFO::Logging to StdErrLog::DEBUG=false via
org.eclipse.jetty.util.log.StdErrLog
2016-10-14 22:55:27.892:INFO::jetty-7.0.0.v20091005
*snip*
```

When you're done, shut down the docker container:

```
$ docker stop c8ea1197f925
c8ea1197f925
```

SDK Cheat Sheet

Get service URLs from the SDK configuration

Generally speaking, if you're using other SDK modules like DataFileUtil to handle talking to the KBase stores as you should, you never need to worry about getting the correct urls. If you need to talk to the stores directly, you can get the appropriate urls from the configuration maps available in the service.

Configuration map keys

The config map keys are as in the deploy.cfg file.

Python

In python, the configuration is available in the SDK implementation constructor and will automatically be loaded with service endpoint urls, e.g.

self.workspaceURL = config['workspace-url']

Example here.

Java

In Java, the config is available in the super class variable `config` in the compiled server class, e.g.

this.workspaceURL = super.config.get("workspace-url");

Perl

Unfortunately Perl users currently have to parse the config file manually:

```
use Config::IniFiles;
my $config_file = $ENV{ KB_DEPLOYMENT_CONFIG };
my $cfg = Config::IniFiles->new(-file=>$config_file);
my $wsInstance = $cfg->val([module name goes here],'workspace-url');
die "no workspace-url defined" unless $wsInstance;

$self->{'workspace-url'} = $wsInstance;
```

TODO: automate this in the Perl impl file

SDK Cheat Sheet

Handling workspace object provenance

Again, assuming you're using DataFileUtils to save objects to the workspace you don't need to worry about provenance; it's handled for you. The cases where you need to worry about provenance are:

- If you're saving an object to the workspace directly
- If you save an object, and then save another object for which the provenance should point to the former object.
- If you want to manually set parts of the provenance.

The automatic provenance handling in DataFileUtils will include any objects passed into the job runner (e.g. the NJSWrapper) in the provenance of any saved objects, but obviously doesn't know about any objects you create. If those objects need to be linked in the provenance of subsequently created objects, you'll need to pull the provenance from the callback server and perform any necessary modifications manually.

Similarly if you're saving objects to the workspace directly you'll need to provide provenance yourself - again, fetch the provenance from the callback server and make any manual modifications necessary.

The specification of a workspace provenance action is here. The provenance returned by the methods below is a list of provenance actions, which is what the workspace accepts upon saving an object.

Python

In python, fetching provenance from the callback server is easy; just call the provenance() method on the context object provided in every method:

```
prov = ctx.provenance()
```

Example

Java

Java is not as well developed, but still possible.
Example
Example of setting up the callback url

TODO: Make java as easy as python

SDK Cheat Sheet

Perl

Perl is in a similar state as Java - doable, but not as clean as Python.

This example assumes the SDK module is called `perltest`.

Import the module client (we can use any client, but this one must exist):

```
use perltest::perltestClient;
```

Get the callback URL (probably want to do this in the constructor):

```
$self->{callbackURL} = $ENV{SDK_CALLBACK_URL};
```

Get the provenance:

```
my $cli = perltest::perltestClient->new($self->{callbackURL});
my $res = $cli->{client}->call($self->{callbackURL}, [], {
      method => "CallbackServer.get_provenance",
      params => []
});
my $prov = $res->result->[0];
print(Dumper($prov));
```

TODO: Make Perl as easy as python

SDK Cheat Sheet

Reference Data

(authored by Roman, original post)

1. declare version of data
2. add your folder into .dockeringnore
3. customize init entrypoint
4. implement initialization script/code

After that your data will appear in /data folder instead of /kb/module/data

Remove old docker containers left by kb-sdk test

(credit goes to Roman and Shane)

Sometimes an error message might indicate that you're out of space, you can check:
cd test_local
./run_bash.sh
df -h

Remove stopped containers
docker ps -a -f status=exited -q | xargs docker rm

Remove all old docker containers (with caution):
docker ps -a | tail -n+2 | cut -f1 -d " " | xargs docker rm -v

Remove images with 'kbase' or 'test/' or 'none'
docker images | grep -e 'test/' -e '.kbase.us' -e 'none' | awk '{print $3}' | xargs docker rmi

Remove orphan images:
docker rmi $(docker images -q --filter "dangling=true")

Most of containers are left by local sub-jobs. There was recent change in shell-script running such local sub-jobs ("test_local/run_subjob.sh") adding "--rm" after "run_docker.sh run". If you don't see such change in your module (it's created before this change committed for instance) you can add it manually yourself.