Arvados - Story #9045

[SDKs] `arv keep docker` can upload an image already saved as .tar

04/25/2016 03:38 PM - Brett Smith

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assigned To:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:	Arvados Future Sprints		

Description

Use case: You already have a Docker image saved as a .tar (e.g., from arv-mount). You can upload it directly with a simple arv keep docker call. This functionality works without ever calling the docker client locally.

Related issues:

Copied to Arvados - Story #9147: [SDKs] `arv keep docker` can make links for ...	**New**	**05/05/2016**

History

#1 - 04/25/2016 03:38 PM - Brett Smith

- Subject changed from [SDKs] `arv keep docker` exposes individvual operations: upload an image already saved as .tar; make links for an image in a collection to [SDKs] `arv keep docker` exposes individual operations: upload an image already saved as .tar; make links for an image in a collection

#2 - 05/05/2016 02:56 PM - Brett Smith

- Subject changed from [SDKs] `arv keep docker` exposes individual operations: upload an image already saved as .tar; make links for an image in a collection to [SDKs] `arv keep docker` upload an image already saved as .tar

- Description updated

Split the tagging story into #9147.

#3 - 05/05/2016 02:56 PM - Brett Smith

- Target version set to Arvados Future Sprints

#4 - 05/05/2016 02:57 PM - Brett Smith

- Subject changed from [SDKs] `arv keep docker` upload an image already saved as .tar to [SDKs] `arv keep docker` can upload an image already saved as .tar

#5 - 06/17/2016 12:04 AM - Brett Smith

THIS IS OBSOLETE AND HISTORICAL: see #9045-7 instead.

Proposed implementation

Add an optional source argument. (Musings on positional vs. switch below.)

arv-keepdocker already works in two phases: make sure the image is in a collection on the cluster; then copy or create appropriate tags. The idea is to more cleanly divide these, so it's easier to add ways to upload collections, or identify existing ones; and then just have the tagging phase take care of the rest.

The procedure to get up to "collection exists on the cluster":

- If source is specified, read that file to double-check that it's a Docker image, and determine its image hash and creation timestamp. If that's all good, upload it as a new collection. (Yes, this implies --force. I think that's okay since it mirrors arv-put's behavior.) If any of that fails (source is not a file, is not a Docker image, whatever), abort with an error.
- Otherwise, if --force was not specified, look for a matching image on the cluster. If one is found, use that as the source. This is roughly current behavior, but this functionality should be switched to use list_images_in_arv rather than doing its own low-level link matching. Doing so ensures that we get a consistent set of metadata for the image's hash and creation time.
- Otherwise, request the specified image, including the image hash and creation timestamp, from the Docker daemon. If that succeeds, create a collection by streaming docker save to arv put. This is an intentional behavior change from the current cache-and-upload behavior. Cache-and-upload surprises users (using a lot of space unexpectedly, failing when space isn't available), and now that we can upload from a file source, it's no longer necessary: anybody who wants it can manually docker save and arv-keepdocker that. Plus, cache-and-upload was intended to permit resuming uploads, but honestly, for most Docker images it's just not that big a deal if the upload is interrupted.

At this point, we should now have a source collection, and an associated image hash and image creation time. For the collection and two metadata

link objects, search for a matching object in the destination project. If none exists, create an object copy in the destination project.

Positional argument vs. switch: the tags problem

In pre-1.0 Docker, tags were a totally separate thing: you usually referred to them in the CLI with a -t switch. Now the colon-separated syntax prevails, and most users expect to be able to use it. But arv-keepdocker was written in a pre-1.0 world, and right now it expects to get a tag as a separate, optional positional argument.

This makes things awkward for adding positional arguments. It would be natural to add source as a positional argument: arv-keepdocker --project-uuid=UUID repo/imagename source. But right now arv-keepdocker expects that second argument to be a tag, and there's no good way to keep perfect backward compatibility while allowing that kind of use. I thought about bridging the gap with heuristics; i.e., if argument 2 is an existing file, treat it as a source; otherwise, treat it as a tag. But this can lead to hard-to-diagnose errors in corner cases on both ends, like if user happens to have a file that shares a name with a desired tag in the current directory; or the user makes a typo and the argument gets understood as the type they didn't expect.

The only way to provide perfect backward compatibility without surprising users regularly is to eschew new positional arguments. Source would have to be specified using a switch like --source= or --from=.

But maybe we should consider making an API break? arv-keepdocker is not scripted very often, so this is a relatively safe place to do it. (But note ops does script it, so they'd need advance notice about a change for their deployment scripts.) And this problem is only going to get worse as Docker moves on.

No matter what, we should support the repo:tag syntax in the first argument, and use that as the tag when it's given.

Next steps

source could be a collection identifier. This means "assume the collection has a single .tar in it; open that, find the image metadata inside it, and create appropriate metadata links." This should be implemented separately in #9147, but I wanted to put together a UI proposal that could handle both intuitively.

I think arv-keepdocker should just be smart about how it parses the source by default (first look for a file, then a collection, then abort if nothing was found), but there should be a switch to disambiguate. e.g., --from=file means source should only be treated as a file; --from=collection means to search for the source as an existing Arvados collection.

Add support for --project-uuid=source. This means "create link tags for the Docker image in the project where the source collection already lives."

Example uses

Assuming we break API compatibility, and implement this and #9147 as described, here's how you use arv-keepdocker. If we don't break API compatibility, the syntax changes a bit but the basic patterns are all still the same.

As now, upload my/image from the local Docker daemon to Arvados:

```
arv-keepdocker my/image
```

Upload an already-saved image to Arvados as my/image:saved:

```
arv-keepdocker my/image:saved filename.tar
```

As now, efficiently copy my/image from one project to another, assuming it exists on the cluster:

```
arv-keepdocker --project-uuid=UUID my/image
```

Re-tag an existing Docker image collection as my/image:retagged, in the project where COLL_ID already lives:

```
arv-keepdocker --project-uuid=source my/image:retagged COLL_ID
```

Load the existing Docker image collection, efficiently copy it to another project, retagged as my/image:newtag:

```
arv-keepdocker --project-uuid=UUID my/image:newtag COLL_ID
```

#6 - 06/20/2016 08:18 PM - Tom Clegg

I'd like it if we could use a language reminiscent of docker's own:

E.g., fetch an image from dockerhub, store it in keep, tag it:

```
arv docker pull foobar
```

#7 - 06/21/2016 05:57 PM - Brett Smith

Tom Clegg wrote:

> I'd like it if we could use a language reminiscent of docker's own

Agreed. New idea:

Branch 1: Move arv-keepdocker's methods that talk to the API server to the SDK

Add an arvados.docker module with the following contents:

```
# All the search methods return dictionary objects like _new_image_listing
# currently returns in keepdocker.py.
# Specifying project_uuid limits results based on links found immediately
# within that project.

# image_name should be in the format `repo/name:tag`.  `:tag` can be omitted
# to find all results matching `repo/name`.
search_images_by_name(api_client, num_retries, image_name, project_uuid=None)

# image_hash_prefix is a string of hexadecimals.
search_images_by_hash(api_client, num_retries, image_hash_prefix, project_uuid=None)

# image_name_or_hash_prefix can be in either format.  It searches by name first.
# If no results are found, it falls back to searching by hash prefix.
search_images(api_client, num_retries, image_name_or_hash_prefix, project_uuid=None)

# Make the docker_image_repo+tag link for collection_uuid inside project_uuid.
# If `:tag` is omitted from image_name, add `:latest`.
add_image_name(api_client, num_retries, project_uuid, collection_uuid, image_name)

# Make the docker_image_hash link for collection_uuid inside project_uuid.
add_image_hash(api_client, num_retries, project_uuid, collection_uuid, image_hash)

# Convenience method to call add_image_name and add_image_hash.
add_image_metadata(api_client, num_retries, project_uuid, collection_uuid, image_name, image_hash)

# Make sure that project_uuid contains the following, creating whatever objects
# are needed:
# * At least one collection with the same portable data hash as the source
#   collection.
# * At least one docker_image_repo+tag link pointing to one of those collections.
# * At least one docker_image_hash link pointing to the same collection.
# If neither image_name or image_hash is specified, that's an error
# (how we handle that TBD—if it ends up being a noop, that's OK).
# Returns the collection record that matches all these criteria.
find_or_create_image(api_client, num_retries, project_uuid, collection_record_or_id, image_name=None, image_ha
sh=None)
```

These are just the public methods that must exist. The branch can (and is expected to) add private methods as needed to DRY and simplify the implementation. It's expected to add relatively little new code; it should be possible to extract most of the implementation from what's currently in keepdocker.py.

keepdocker.py should be reimplemented with these new methods, but include no functional changes.

Branch 2: Add arv docker load

Usage:

```
arv docker load [arv-put options...] [--project-uuid=UUID] [--input=INPUT|-i INPUT] source
```

input can be a collection PDH or UUID, Docker image name or hash, or path to a .tar file previously saved with docker tar. If it is not specified, the command expects to read a .tar file from stdin.

When input is a collection PDH or UUID, the command looks up the collection's Docker image hash, then copies it to the named project with find_or_create_image.

When input is a Docker image name or hash, the command calls docker save to get a .tar from the local Docker daemon, and streams that directly to arv-put to create a new collection. The lack of a file cache is an intentional behavior change from keepdocker.py, because caching is now superfluous: if you want that, you can just docker save >image.tar && arv docker load -i image.tar

When input is an image .tar, the command gets the image hash out of the metadata inside the .tar file, and creates a new collection from it using arv-put.

This is mostly implemented as arvados.commands.docker in the Python SDK, with a thin wrapper added to the arv command. Use argparse subcommands to register the load subcommand and parse its options.

Branch 3: Add arv docker push

Usage:

```
arv docker push [arv-put options ...] [--project-uuid=UUID] [--source=COLLECTION] image_name
```

arv docker push's main job is to publish Docker images to Arvados project, by making sure all the right metadata links exist. For symmetry with docker push, it can also load images by their name, either from Arvados or the local Docker daemon.

1. Find a source collection.
    1. If source is specified, load that collection and its Docker image hash. If either can't be found, that's a fatal error.
    2. Otherwise, search for an image in Arvados with image_name. If found, use the best matching collection as the source collection.
    3. Otherwise, run (the code equivalent of) arv docker load --project-uuid=UUID image_name.
2. Create appropriate image metadata links.
    1. If source was specified and project_uuid was not, call add_image_name(..., collection['owner_uuid'], collection['uuid'], image_name).
    2. Otherwise, call find_or_create_image with the source image and destination project (or home project, if not specified) to make sure the image and all its metadata are available in that project.

Future work

In the future, we could write a server that implements the Docker Hub API, and does much of the behind-the-scenes work to store image layers in Keep and register images in the Arvados API server. Once that's available, users can use the Docker CLI or other client tools to push images directly to that repository server. At that point, arv docker can become a very thin wrapper around docker that just adds appropriate switches.