Adama Documentation

Release 0.4

Walter Moreira

June 21, 2016

Contents

Warning: This document is currently a collection of mostly independent pieces of documentation for Adama. In the near future we plan to organize it properly, to function as the one-stop-shop for all Adama documentation.

CHAPTER 1

Contents:

1.1 Adama Tutorial

Warning: This API is preliminary, and it **may** change.

Follow this tutorial with any HTTP client. Here we'll use curl.

The languages supported by Adama are currently: Python, Javascript.

The base url of the Adama is https://api.araport.org/community/v0.3. For brevity, in what follows we use the environment variable `API` to refer to this base. Declare the variable in your shell to be able to copy and paste the examples:

```
$ export API=https://api.araport.org/community/v0.3
```

You need to obtain a token through your method of choice. In what follows, the environment variable `TOKEN` is assumed to be set to such token. It is convenient to set it as:

```
$ export TOKEN=my-token
```

1.1.1 Checking access to Adama

A `GET` request to `$API/status` should return:

```
$ curl -L -X GET $API/status -H "Authorization: Bearer $TOKEN"
{
    "api": "Adama v0.3",
    "hash": "6869fde8e2617ab8f8a58c5c09b1512a80185500",
    "status": "success"
}
```

The `hash` field points to the git commit of Adama that is currently serving the response.

1.1.2 Registering a Namespace

Namespaces allow Adama to group adapters. Create a new namespace with:

```
$ curl -X POST $API/namespaces -Fname=tacc -Fdescription="TACC namespace" \
   -H "Authorization: Bearer $TOKEN"
{
    "result": "https://api.araport.org/community/v0.3/tacc",
```

```
    "status": "success"
}
```

Retrieve information about a known namespace from the url `$API/<namespace>` (for example `$API/tacc`). Obtain the list of all registered namespaces with:

```
$ curl -X GET $API/namespaces -H "Authorization: Bearer $TOKEN"
{
    "result": [
        {
            "description": "TACC namespace",
            "name": "tacc",
            "url": null
        }
    ],
    "status": "success"
}
```

Delete a namespace with the verb `DELETE` to the url `$API/<namespace>`.

1.1.3 Registering an Adapter

Adama currently supports two types of adapters: `query` and `map_filter`.

A `query` adapter receives a request through Adama, performs a query to an external service and returns the results as JSON objects.

A `map_filter` adapter transforms and/or filters JSON objects returned from an external service.

An adapter can be registered using two methods (or a combination of them):

- `POST` the code and the metadata. The code can be a single file, tarball, or zip archive.
- `POST` an URL to a git repository containing the code and the metadata.

It is strongly recommended to use the second method, since it makes it easier to share, to modify, and to keep track of changes in the adapters.

We show an example of a `query` adapter registered via the first method, and an example of a `map_filter` adapter registered via the second method.

Writing a query adapter

Write a Python module `main.py`, with a function `search` that takes a JSON object as argument in the form of a dictionary. Print JSON objects to standard output, separated by the characters `"---"`.

For example:

```
# file: main.py

import json

def search(args):
    print json.dumps({'obj': 1, 'args': args})
    print "---"
    print json.dumps({'obj': 2, 'args': args})
```

This function can be tested in the Python interpreter:

```
>>> import main
>>> main.search({'x': 5})
{"args": {"x": 5}, "obj": 1}
---
{"args": {"x": 5}, "obj": 2}
```

Registering

To register this adapter with the name `example` in the namespace `tacc`, we `POST` to `$API/tacc/services` with the metadata and the code. In this example we show only some of the optional fields, refer to the API docs for the full documentation.

- **name** (mandatory): the name of the adapter (`example` in this case),
- **type** (mandatory): the type of adapter: `query`, or `map_filter`,
- **version** (optional): version (default `0.1`),
- **url** (mandatory): URL of the external service (`http://example.com` in this case),
- **notify** (optional): URL to notify with a POST request when the adapter is ready to use,
- **code** (mandatory): module `main.py`.

Using curl:

```
$ curl -L -X POST $API/tacc/services \
    -F "name=example" -F "type=query" -F "url=http://example.com" \
    -F code=@main.py -F "notify=https://my.url" \
    -H "Authorization: Bearer $TOKEN"
{
    "message": "registration started",
    "result": {
        "notification": "https://my.url",
        "search": "https://api.araport.org/community/v0.3/search",
        "list": "https://api.araport.org/community/v0.3/list",
        "state": "https://api.araport.org/community/v0.3/example_v0.1"
    },
    "status": "success"
}
```

At this point the registration procedure is started in the server. It may take some time, and in the meantime the state of the adapter can be checked with:

```
$ curl -L -X GET $API/tacc/example_v0.1 \
   -H "Authorization: Bearer $TOKEN"
{
    "result": {
        "msg": "Workers started",
        "service": null,
        "slot": "busy",
        "stage": 4,
        "total_stages": 5
    },
    "status": "success"
}
```

When ready, Adama will post to the url specified in the `notify` parameter (if any), and the adapter can be seen in the directory of services. To see a list of all the available services:

```
$ curl -L -X GET $API/tacc/services \
   -H "Authorization: Bearer $TOKEN"
{
    "result": [
        {
            "code_dir": "/tmp/tmpolAjqz/user_code",
            "description": "",
            "json_path": "",
            "language": "python",
            "main_module": "main",
            "metadata": "",
            "name": "example",
            "namespace": "tacc",
            "notify": "https://my.url",
            "requirements": [],
            "type": "query",
            "url": "http://example.com",
            "version": "0.1",
            "whitelist": [
                "localhost",
                "example.com"
            ],
            "workers": [
                "57a4e10cb84aba5473d81c58011fcb78ce1b2684d67f0c2cc7540be191d4b589"
            ]
        }
    ],
    "status": "success"
}
```

Delete the service `example_v0.1` by using the `DELETE` verb to `$API/tacc/example_v0.1`.

Writing a map_filter adapter

Start a git repository as:

```
$ mkdir map_filter_example
$ cd map_filter_example
$ git init
```

Add the file `main.py` with content:

```
def map_filter(obj):
    obj['processed_by'] = 'Adama'
    return obj
```

This module can be tested in the Python interpreter:

```
>>> import main
>>> main.map_filter({'key': 1})
{'key': 1, 'processed_by': 'Adama'}
```

Add also the file `metadata.yml` with the metadata information:

```
---
name: map_example
version: 0.1
type: map_filter
```

```
main_module: main.py
url: https://api.araport.org/community/v0.3/json
whitelist: ['127.0.0.1']
description: ''
requirements: []
notify: ''
json_path: result
```

The url `https://api.araport.org/community/v0.3/json` returns a sample JSON response:

```
$ curl https://api.araport.org/community/v0.3/json
{
    "result": [
        {
            "key": 1
        },
        {
            "key": 2
        },
        {
            "key": 3
        }
    ],
    "status": "success"
}
```

The array of objects we want to process is in the field `result`, so we declare it in the `json_path` field of the metadata file.

Commit both files into the git repository:

```
$ git add main.py metadata.yml
$ git commit -m "Add main and metadata"
```

The git repository has to be made available somewhere. For example, if using Github with the username `waltermoreira` and repository name `map_adapter`, we can register the adapter with:

```
$ curl -L -X POST $API/tacc/services \
    -F "git_repository=https://github.com/waltermoreira/map_adapter.git"
```

1.1.4 Performing a query

Use the adapter `example_v0.1` registered in the `tacc` namespace by doing a `GET` from `$API/tacc/example_v0.1/search`.

For example:

```
$ curl -L "$API/tacc/example_v0.1/search?word1=hello&word2=world" \
   -H "Authorization: Bearer $TOKEN"
{"result": [
{"args": {"worker": "887e5cf7c82f", "word1": "hello", "word2": "world"}, "obj": 1}
, {"args": {"worker": "887e5cf7c82f", "word1": "hello"], "word2": "world"}, "obj": 2}
],
"metadata": {"time_in_main": 0.0001881122589111328},
"status": "success"}
```

Notice that the result consists of the two objects generated by `main.py`, including the query argument (in this case containing some extra metadata added by Adama).

Use the adapter `map_example_v0.1` in a similar way:

```
$ curl -L $API/map_example_v5/search \
   -H "Authorization: Bearer $TOKEN"
{"result": [
{"processed_by": "Adama", "key": 1}
, {"processed_by": "Adama", "key": 2}
, {"processed_by": "Adama", "key": 3}
],
"metadata": {},
"status": "success"}
```

1.1.5 Summary

Current endpoints for Adama:

- `$API/status`
  - `GET`: get information about Adama server
- `$API/namespaces`
  - `GET`: list namespaces
  - `POST`: create namespace
- `$API/<namespace>`
  - `GET`: get information about a namespace
  - `DELETE`: remove a namespace
- `$API/<namespace>/services`
  - `GET`: list all services
  - `POST`: create a service
- `$API/<namespace>/<service>`
  - `GET`: get information about a service
  - `DELETE`: remove a service
- `$API/<namespace>/<service>/search`
  - `GET`: perform a query
- `$API/<namespace>/<service>/list`
  - `GET`: perform a listing

1.2 Documenting Parameters

Starting in version 0.3, Adama allows to document the parameters for all the types of adapters.

The user provided documentation is a backward compatible extension of the metadata file. The syntax for declaring parameters is based in Swagger 2.0. The full Swagger spec can be used in the `endpoints` field in the file `metadata.yml`, but a simpler subset is also supported, allowing easier documentation of the most common cases of adapters.

1.2.1 Example

The following is a basic example of an Adama adapter with autogenerated documentation. Refer to the base tutorial for instructions about registering an adapter, and change the base URL from `api.araport.org` to `adama-dev.cloudapp.net` (this is a sandbox, don't worry to break things here).

This adapter is hosted at github.com/waltermoreira/sample-parameter-docs.

The git repository contains two files: `main.py` and `metadata.yml`. The file `metadata.yml` describes an adapter with four parameters:

```
1  ---
2
3  name: my_adapter
4  type: query
5
6  validate_request: yes
7
8  endpoints:
9    /search:
10     parameters:
11       - name: x
12         description: Parameter X
13         type: string
14         required: true
15       - name: y
16         description: Parameter Y
17         type: integer
18         format: int64
19         required: false
20         default: 5
21       - name: z
22         description: Parameter Z
23         type: array
24         required: true
25         collectionFormat: multi
26         items:
27           type: number
28           format: double
29       - name: w
30         description: Parameter W
31         type: string
32         required: true
33         enum:
34           - Spam
35           - Eggs
```

Let's explain the meaning of the several sections.

Line 6 declares that the service wants Adama to validate the input before passing the control to the module `main.py`. The default is `no`, which allows for backward compatibility and gradual documentation of existing adapters. When Adama validates the input, the developer of the adapter can be confident that the arguments for the function `main.search` have the types specified in `metadata.yml`.

For example, the function `main.search` in this adapter contains the lines:

```
def search(args):
    # ...
    w = args['w']
```

```
    assert w in ('Spam', 'Eggs')
    # ...
```

When `validate_request: yes`, the developer knows that these two lines will not raise an error, since the parameter `w` is mandatory and it is one of the values described in the enumeration.

Lines 8-9 declare that we are documenting the endpoint `/search` of the service. Current Adama adapters have a defined list of supported endpoints (see live API docs), but the list can be expanded in the future.

Lines 11-14 describe the mandatory parameter `x` of type string.

Lines 15-20 describe the optional parameter `y` of type integer. In case it is not passed in the request, the adapter will get the default value 5.

Lines 21-28 describe a parameter `z` of type array of numbers. An array can be passed in a request in different ways. Line 25 declares that this adapter accepts a `multi` format. This means that a request of the form:

```
https://araport-dev.cloudapp.net/.../my_adapter_v0.1/search?z=2&z=3.1&z=10&...
```

will pass the array `[2.0, 3.1, 10.0]` to the function `main.search`. See the Swagger spec for the available ways to pass arrays in requests.

Lines 29-35 describe a parameter `w` with a defined set of possible values.

Registering the adapter

Register the adapter following the instructions in the base tutorial. Here is an example using the tool httpie [1] for performing the requests:

```
export ADAMA=https://adama-dev.cloudapp.net/community/v0.3
export TOKEN=...my token...
http POST https://$ADAMA/my_namespace/services \
    Authorization:"Bearer $TOKEN" \
    git_repository=https://github.com/waltermoreira/sample-parameter-docs \
    validate_request=yes
```

The adapter health can be checked with the request:

```
http https://$ADAMA/my_namespace/my_adapter_v0.1 Authorization:"Bearer $TOKEN"
```

which should return the a successful response with a lot of "nerd stats".

Accessing the documentation

Once the adapter is successfully registered, the full Swagger documentation can be accessed in the `/docs` endpoint of the adapter. For example:

```
# return Swagger documentation in JSON format
http https://$ADAMA/my_namespace/my_service_v0.1/docs \
    Authorization:"Bearer $TOKEN"

# return Swagger documentation in YAML format
http https://$ADAMA/my_namespace/my_service_v0.1/docs?format=yaml \
    Authorization:"Bearer $TOKEN"
```

[1] `httpie` is a strongly recommended replacement for `curl`: http://httpie.org/

This output is usually not meant for human consumption (although the YAML output is very readable). The main goal of this endpoint is to be fed to any Swagger 2.0 complaint browser. Araport near future plans include to provide a developer console that will include all the adapters documentation, the Agave API, and the Adama base API. In the meantime, Adama provides the endpoint `/docs/swagger` which is a basic instance of a Swagger browser.

To interact with the documentation, access with the browser the URL:

```
https://adama-dev.cloudapp.net/community/v0.3/my_namespace/my_adapter_v0.1/docs/swagger
```

1.3 Metadata format

An example of `metadata.yml` file:

```
---
description: "Given a valid AGI locus, fetch coexpressed genes from the ATTED-II database"
url: http://atted.jp/
main_module: main.py
name: atted_coexpressed_by_locus
type: query
version: 0.1
whitelist:
  - atted.jp
```

The file `metadata.yml` accepts the following fields:

`name` The name of the adapter

`version` The version of the adapter. Please, use semantic versioning.

`type` Type of the adapter. One of:

- `query`
- `generic`
- `map_filter`
- `passthrough`

`description` Free form text to describe the purpose of the adapter.

`url` Url for the third party data source the adapter is accessing, if any. Depending on the type of the adapter, this may be for documentation purposes (`query` and `generic`), or it may be used directly by Adama to access the service on behalf of the user (`map_filter` and `passthrough`).

`whitelist` An additional list of ip's or domains that the adapter may need to access.

`main_module` The name (including the path relative to the root of the git repository) of the main module for this adapter. If omitted, Adama will search for a module named `main.*`.

`notify` An url that will receive a POST with the data of the new registered adapter once it is ready to receive requests.

`requirements` A list of extra modules to add to the adapter at installation time. These modules should be installable via the standard package manager of the language used by the adapter (for example: `pip` for Python, `gem` for Ruby, etc.)

`validate_request` Whether to validate the parameters according to the provided documentation. By default this option is `no`. If enabled, the parameters of a request are validated before passing control to the user's code in the adapter.

`endpoints` Documentation about the parameters accepted by this adapter (see documenting parameters).

json_path This field is meaningful only for `map_filter` adapters. If the third party service returns an array of JSON objects to be processed by the adapter, then this field can be empty. Otherwise, if the array is nested inside a JSON object, this field is used to specify how to reach it.

For example, if the response of the third party service is the following JSON object:

```
{
  "status": "success",
  "result":
     {
       "data": [1, 2, 3, 4, 5],
       "name": "integers"
     }
}
```

and the adapter is interested in the array of integers, then we set:

```
json_path: result.data
```

1.4 Types of Adapters

At the fundamental level, Adama is a builder of webservices. Its task is to abstract the infrastructure necessary to publish a webservice, such as security, scalability, fault tolerance, monitoring, caching, etc. A developer of an adapter can concentrate just in the source and transformations of the data.

To make it easy to develop webservices, Adama provides several types of adapters, trying to minimize the amount of code a developer has to write.

Note: Webservices

A *webservice* is a function or process (usually located at some URL) that accepts a HTTP request (usually a GET or POST, but in general any HTTP verb) and returns a response. The type of response varies wildy: JSON, text, images, HTML, etc. In Adama there are extra features for dealing with the JSON type, since it's the preferred format for webservices.

We denote a webservice that returns a response of type T by

$$\text{WS}_T \equiv \text{Request} \to T$$

Note: Simple webservices

We call *simple webservice* to a webservice that restricts the request types to just a `GET` with query parameters. Such requests have the form:

```
GET http://example.com?key1=value1&key2=value2&...
```

In other words, a simple webservice is a service that accepts a set of key/values and returns a response of type T. We denote it with the symbol:

$$\text{SimpleWS}_T \equiv \{\text{key} : \text{value}\} \to T$$

Adama prefers simple webservices that return data as an array of JSON objects: $\text{SimpleWS}_{[\text{JSON}]}$.

There are two types of adapters that return a simple webservice of type $[JSON]$:

- **query**: a *query* adapter has the type:

$$\big(\{k : v\} \to \text{Stream(JSON)}\big) \to \text{SimpleWS}_{\text{[JSON]}}$$

  This means that the developer provides a function that accepts a set of key/values and that returns a stream of JSON objects. Given this function, Adama constructs a simple webservice that accepts a `GET` request with query parameters and returns an array of JSON objects.

  To return a stream of JSON objects, the developer just has to print to standard output, separating each object with the line $---$.

- **map_filter**: a *map_filter* adapter has the type:

$$\big(\text{JSON} \to \text{JSON}, \text{SimpleWS}_{\text{[JSON]}}\big) \to \text{SimpleWS}_{\text{[JSON]}}$$

  This adapter takes two arguments: a function that transforms JSON objects, and an existing simple webservice returning an array of JSON objects. Given those parameters, Adama constructs a simple webservice that consists in transforming the output of the original webservice via the provided function.

There are two additional adapters that provide extra functionality, for the cases when returning JSON objects is not feasible:

- **generic**: a *generic* adapter is similar to a *query* adapter, but the return type is arbitrary:

$$\big(\{k : v\} \to T\big) \to \text{SimpleWS}_T$$

  Also, rather than returning an stream via printing to standard output, a generic adapter simply returns the object of type T.

- **passthrough**: a *passthrough* adapter makes Adama a proxy for an arbitrary existing webservice. The type is:

$$\big(\text{WS}_T\big) \to \text{WS}_T$$

  That is, it takes an existing webservice and it constructs the same webservice, except by changing the URL and by providing extra features such as caching, authentication, etc.

1.5 History

Warning: This section contains some legacy documents. Most of the information here may be out-of-date or incorrect.

1.5.1 Introduction

The Arabidopsis Information Portal project wants to present a unified interface to a wide set of services that provide data for Arabidopsis research.

Problem

There is a huge amount of data related to Arabidopsis being produced and being served to the internet. However, the way the data is presented varies wildly across each service. The main aspects in which services vary are:

- The format the data is presented: anything from CSV to binary formats;
- The API used to access the service: anything from a RESTful service to SOAP based service;

- The vocabulary used to denote the objects.

Trying to collect and to unify these aspects is difficult given their dynamic nature. Data formats and interfaces change quickly. A central service would need to be continuously updating its software. *Adama* tries to solve this problem by engaging the developers of each service in a distributed and federated architecture.

Strategy

Adama is a RESTful service that provides two main endpoints:

- **Registration interface**: a provider of a data source (see *Roles*) can register a service to be accessible through Adama. The provider's work (see *Developer Role Quickstart*) consists in defining a conversion procedure between the Araport language (see *Araport Language*) and the native format of the data source.
- **Query interface**: an user of Araport (see *User Role Quickstart*) can perform queries in the Araport language and direct them to one or more registered services.

In addition to this interface or API, the architecture provides the following features that aim to solve the points in *Problem*:

- *Unified language and API*. Araport provides access through a RESTful service, accepting and returning JSON objects. The schema for these objects is defined in the *Araport Language*, a rationalized and extensible format designed to express queries for all the registered services. This allows a query to be spread to multiple data sources simultaneously and the results to be collected through a single endpoint.
- *Distributed responsability*. The task of maintaining an up-to-date conversion between the Araport Language and the third party services is distributed among the developers of the latter. Adama provides an extremely easy way to develop the adapters and to test them in complete isolation of the Araport site (see *Developer Role Quickstart*).
- *Security and isolation*. The adapters from each developer and for each service are run fully isolated and with a restricted access to the network. This allows to control the security in execution, and to control the network usage (including throttling, caching, etc.).
- *Scalability*. Adama takes care of the horizontal scaling of the adapters. The scalability can be performed in a dynamic way, depending on the load to particular services. In addition, load balancing to the full Adama API is performed automatically.
- *Extra Services*. Adama provides extra services for each data source: pagination, count, and caching. These services do not require support from the data source and they do not require extra work from the developer of the adapter.

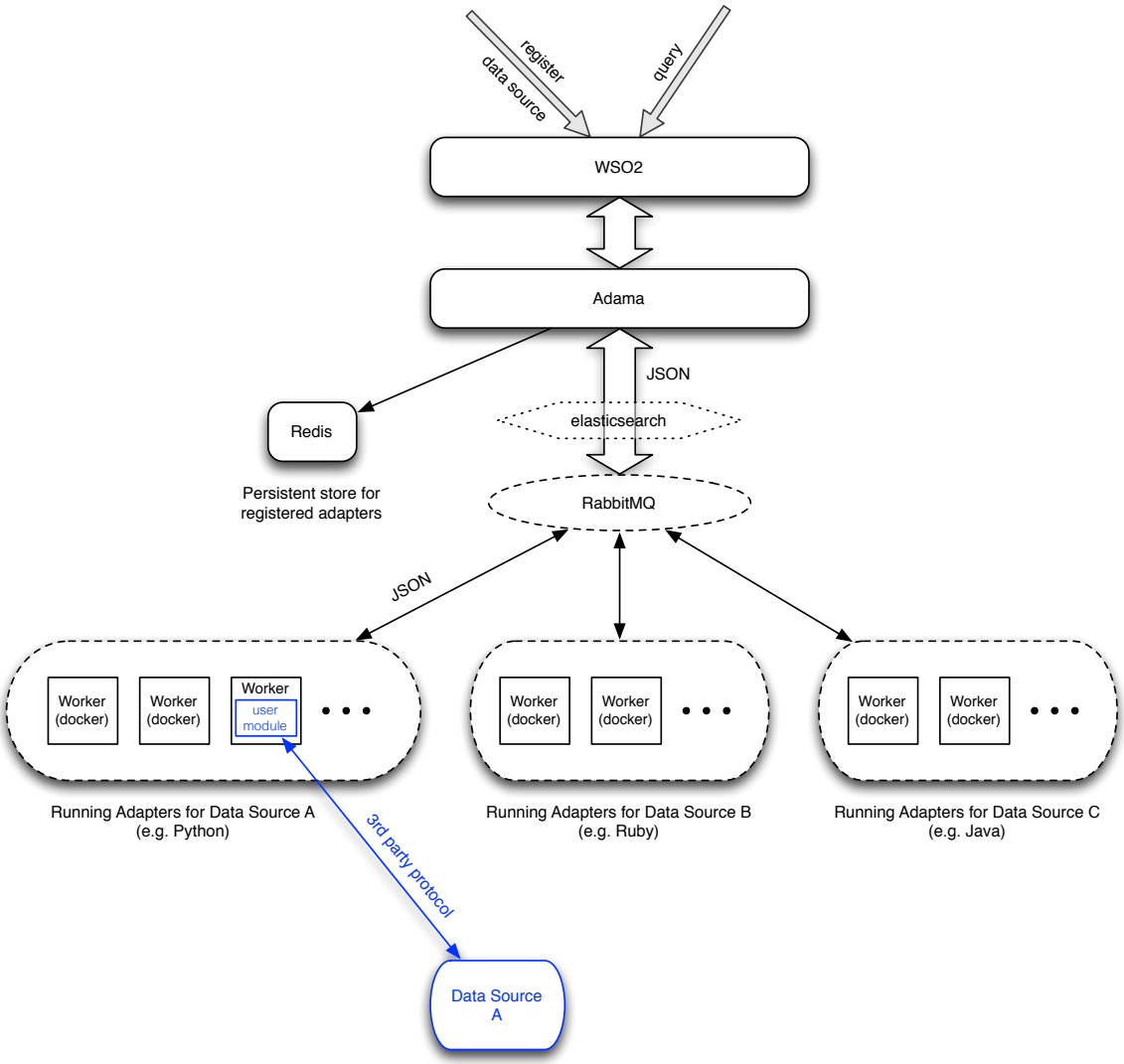
1.5.2 Architecture

Overview

The following picture illustrates the architecture of *Adama*:

The main components are:

- *Adama* itself is a Python application running over an Apache web server, providing the REST endpoints and business logic. (see *API*).
- A data store (Redis) collects a registry of adapters.
- A task queue (RabbitMQ) distributes work to the adapters for each request to the `query` endpoint.


register
data source
query
WSO2
Adama
JSON
Redis
elasticsearch
Persistent store for
registered adapters
RabbitMQ
JSON
Worker
(docker)
Worker
(docker)
Worker
user
module
Worker
(docker)
Worker
(docker)
Worker
(docker)
Worker
(docker)
Running Adapters for Data Source A
(e.g. Python)
Running Adapters for Data Source B
(e.g. Ruby)
Running Adapters for Data Source C
(e.g. Java)
3rd party protocol
Data Source
A

- The workers are Docker containers that get created through the `register` endpoint, and get started and shut-down under Adama command, depending on the load of the `query` endpoint. The workers are continuously consuming work from the task queue.
- The *user module* in the picture is the adapter provided by the developer at registration time (see *Adapter API*).
- The user module, data sources, and 3rd party protocol (colored blue in the picture) specify the parts under control of the developer role (see *Developer Role Quickstart*).
- Elasticsearch intercepts objects generated by the queries and creates an aggregated database that allows extra capabalities for searching and analyzing the data sources. Data sources can opt-in or opt-out of this functionality at any time.

API

Warning: This section is in flux. The API will change before a stable release.

In what follows, assume that `$ADAMA` is the base url of the Adama services.

The resource `$ADAMA/register` accepts a `GET` and `POST` request. The `GET` verb returns a list of registered adapters in the form:

```
{
  "identifier": "foo_v0.1",
  "language": "python",
  "name": "foo",
  "url": "http://example.com",
  "version": "0.1",
  "workers": [
    "ad89eefd01ca4648dc388dd12b95816cc24fe938ca801bd938ef982fa057a489",
    "7ec01fefe8c8a54d92a773569e0dc0b38be8b3a9bbdea7a16da05c5a800117ad",
    "ca88c73a13c17704e348eca468c101c784654dbe90b1156846e34507d0cccd6a",
    "7f3fbb1faadd7493c349dd316ad3d1dcf8775b8851dbf0e9111b00bf64c03612"
  ]
}
```

The identifier is unique and can be used in the queries to refer to the service provided by this adapter. The `workers` field contains internal information about the workers currently running to attend this adapter (it may be removed from the public API in the future). The `POST` verb allows to register a new adapter. It accepts the parameters described below. The type of the parameter and whether they are mandatory or optional is described besides the paramter name.

`name` [form, mandatory] Name of the service provided by this adapter (together with the version they must form a unique identifier)

`version` [form, mandatory] Version of the service.

`url` [form, mandatory] URL of the data source. The network access for the adapter may be restricted to access only this URL.

`description` [form, mandatory] Human readable description of the service.

`requirements` [form, optional] Comma separated list of third party modules to be installed inside the workers. They should be accessible in the standard package repository for the language bein used (i.e., `pypi` for Python, `rubygems` for Ruby, etc.).

`code` [file, mandatory] The user's code for the adapter. See *Adapter API* for its API and requirements. The code can be provided in a single file, or in a tarball or zip compressed archive. The type is detected automatically.

Workers are started immediately after registration. The response is the standard (`status`, `message`, `result`) triple (see Agave).

The verbs `PUT` and `DELETE` will be implemented in the future to allow administration of already registered adapters.

The resource `$ADAMA/query` accepts `POST` requests to perform queries to a selected list of services. The parameter is a JSON encoded in the body with the schema:

```
{
  "serviceName": "foo_v0.1",
  "query": "...Araport Language query..."
}
```

The `serviceName` field can also be a list of multiple services. The query will be delivered to all of them, and responses will be collected together. See *Araport Language* for the schema of the queries.

Adapter API

Warning: This section is in flux. The API will change before a stable release.

An *adapter* can be written in any of the programming languages supported by *Adama*. The list initially includes:

> Python, Javascript (node.js), Ruby, Java, Lua, Perl.

Other languages can be added in the future by request.

The description that follows is generic, and details for each language will be provided in newer revisions of this document.

An adapter is a module called `main`. It contains a function named `process` which accepts a string and returns nothing. The string argument is a JSON encoded object that will be passed by Adama, and it will contain the query from the user (in the Araport Language).

The task of the function `process` is:

- Convert the Araport Language query to the proper format for the 3rd party service.
- Send the query to the 3rd party service and retrieve the results.
- For each result, convert it to Araport Language and **print** it to screen (as a JSON encoded value).

  The output may use several lines with no restriction. **Print** the line `---` to separate results. Many results can be generated from every result from the data source.

The function `process` in the module `main` can be tested by the developer by simply running it in his or her own system, with no access to Adama or Araport. As long as the adapter follows the protocol to print to standard output as JSON, and to separate the objects with `---`, Adama will be able to capture the results.

Note: By printing to standard output, the adapter is effectively using an *asynchronous* output model, allowing Adama to start delivering results to the clients as soon as possible. It does not require any effort from the developer. And it is actually easier to code than collecting the results in a temporary container and returning them.

The `main` module can have dependencies of two types:

- It can depend on other modules provided by the developer. In such case, the developer can choose to register the adapter as tarball or zip compressed archive. The only requirement is that the `main` module has to be at the root level of the compressed archive. Other files or assets needed by the module can be at the same level or in subdirectories.

- Or it can depend on modules or packages from the standard package repository for the corresponding language. In this case, the extra modules will be installed at registration time in the workers (see *Registration API*).

The language of the adapter is detected automatically by looking at the module `main` uploaded during registration.

See *Developer Role Quickstart* for an example of this API.

Araport Language

Todo

To be defined and to be written.

1.5.3 Using and Developing for Adama

This section shows very basic and quick examples of using the `register` and `query` endpoints of Adama.

Note: These examples are meant to be executed against an Adama instance and they should work. Currently, we do not have an official and public accessible instance, but a dedicated user can build his or her own from `github.com/waltermoreira/adama`, if desired. We expect to have an alpha release on **TBD**.

Roles

We called *developer* an individual who has access or knowledge of an existing data source, and he or she wants to register it into Adama. The developer does not need special access to the third party data source, other than the standard API access it already provides. He or she needs an account on Araport with permissions to register adapters. A developer is expected to have a minimum of skills on some programming language (from the list supported by Adama, see *programming languages*).

We called *user* an individual who is using Adama through the `query` interface. No special permissions are required. Knowledge to access a RESTful API is recommended, although command line and web tools are also planned in the future.

Developer Role Quickstart

In this setion we register an adapter for the service *Expressolog by locus* at `bar.utoronto.ca`.

The service accepts queries as `GET` requests to the URL:

```
http://bar.utoronto.ca/webservices/get_expressologs.php
```

with parameter `request=[{"gene":  "..."}]`.

The **native output** of the service for the gene At2g26230 looks like :

```
{"At2g26230": [
  {"probeset_A": "267374_at",
   "gene_B": "Glyma20g17440",
   "probeset_B": "Glyma20g17440",
   "correlation_coefficient": "0.5264",
   "seq_similarity":"67",
   "efp_link": "http://bar.utoronto.ca/efp_soybean/cgi-bin
```

```
                /efpWeb.cgi?dataSource=soybean&primaryGene
                =Glyma20g17440&modeInput=Absolute"
  },
  {"probeset_A": "267374_at",
   "gene_B":"Solyc11g006550",
   "probeset_B": "Solyc11g006550",
   "correlation_coefficient": "0.1768",
   "seq_similarity": "68",
   "efp_link":"http://bar.utoronto.ca/efp_tomato/cgi-bin
              /efpWeb.cgi?dataSource=tomato&primaryGene
              =Solyc11g006550&modeInput=Absolute"
  }
 ]
}
```

The **Araport Language query** would look like (see *Araport Language*):

```
{
   "query": {"locus": "At2g26230"},
   "countOnly": false,
   "pageSize": 100,
   "page": 1
}
```

The transformed **Araport Language output** will contain two records for each of the previous records:

```
{
    "status": "success",
    "message": "",
    "result": [
        {
            "class": "locus_relationship",
            "reference": "TAIR10",
            "locus": "At2g26230",
            "type": "coexpression",
            "related_entity": "Solyc11g006550",
            "direction": "undirected",
            "score": [
                {
                    "correlation_coefficient": 0.1768
                }
            ],
            "source": "tomato"
        },
        {
            "class": "locus_relationship",
            "reference": "TAIR10",
            "locus": "At2g26230",
            "type": "similarity",
            "related_entity": "Solyc11g006550",
            "direction": "undirected",
            "score": [
                {
                    "similarity_percentage": 68
                }
            ],
            "source": "tomato"
        },
        {
```

```
            "class": "locus_relationship",
            "reference": "TAIR10",
            "locus": "At2g26230",
            "type": "coexpression",
            "related_entity": "Glyma20g17440",
            "direction": "undirected",
            "score": [
                {
                    "correlation_coefficient": 0.5264
                }
            ],
            "source": "soybean"
        },
        {
            "class": "locus_relationship",
            "reference": "TAIR10",
            "locus": "At2g26230",
            "type": "similarity",
            "related_entity": "Glyma20g17440",
            "direction": "undirected",
            "weight": [
                {
                    "similarity_percentage": 67
                }
            ],
            "source": "soybean"
        }
    ]
}
```

The complete code (in Python) for the adapter can be seen at Adama github repository.

In pseudo-code, the function `process` of the module `main.py` can be described as:

```
def process(args):
    # <extract 'locus' from the JSON object 'args'>
    # <send request to the expressologs service>
    for result in # <results from expressologs>:
        obj = # <convert result to Araport format (for type "similarity")>
        print obj
        print '---'

        obj = # <convert result to Araport format (for type "coexpression")>
        print obj
        print '---'

    print 'END'
```

This is the code which the developer of the adapter for the third party data source will create. The developer can test this function interactively in the Python interpreter, independently of any interaction with Adama, as in:

```
>>> import main, json
>>> main.process(json.dumps(
...   {"query": {"locus": "At2g26230"},
...    "countOnly": false,
...    "pageSize": 100,
...    "page": 1}))
...
```

If successful, the function will print to standard output a sequence of JSON objects separated by the lines `'---'`, and

it will print the `'END'` string when the stream of results is exhausted. After a successful testing, the developer will upload this module to Adama by posting to `$ADAMA/register`, for example with the (Python) command:

```
>>> requests.post('$ADAMA/register',
...               data={'name': 'expressologs_by_locus',
...                     'version': '0.1',
...                     'url': 'http://bar.utoronto.ca/webservices/get_expressologs.php',
...                     'requirements': 'requests'},
...               files={'code': ('main.py', open('main.py'))})
...
```

Note that we are assuming the `main.py` module (as in the example on the github repository) is using the module `requests` to access the expressolog service. For such reason, we include it in the `requirements` field, so it is properly installed in the Adama workers. This example can be found already built in the examples directory of Adama github repository.

On success, Adama will return the full identifier for this service (`expressologs_by_locus_v0.1`), and it will start workers ready to attend queries.

User Role Quickstart

After a developer registered the expressolog service as in *Developer Role Quickstart*, a user can query the service by posting to `$ADAMA/query`:

```
>>> requests.post('$ADAMA/query',
...               data=json.dumps({
...                    'serviceName': 'expressologs_by_locus_v0.1',
...                    'query': {'locus': 'At2g26230'}}))
...
```

On success, the response will return a JSON object with the results in the format described in *Developer Role Quickstart*. As mentioned in *Adapter API*, Adama will start streaming the results as soon as they are available from the data source. Users can start reading the results using any of the available increamental JSON parsers.

1.6 Provenance

Starting in version 0.3, Adama allows declaring the provenance for each microservice response.

Provenance for a microservice can be specified in the field `sources:` of the metadata. Its content is a list of objects with the following fields:

`title:` (mandatory) Name of the source

`description:` (mandatory) Human readable description of the source

`language:` (optional) Language (following RFC 4646)

`last_modified:` (optional) Date of last modification. ISO format preferred (e.g.: "2015-04-17T09:44:56"), but it's more permissive. Most unambiguous formats are supported.

`sponsor_organization_name:` (mandatory) Sponsor organization

`sponsor_uri:` (optional) Sponsor URI

`provider_name:` (mandatory) Provider name

`provider_email:` (optional) Provider email

`uri:` (optional) Agent URI

license: (optional) License

sources: (optional) A list of nested sources containing the above fields (recursively including more sources, if necessary). By default it is the empty list.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search