Multiscale Complex Genomics

Project Acronym: MuG
Project title: Multi-Scale Complex Genomics (MuG)
Call: H2020-EINFRA-2015-1
Topic: EINFRA-9-2015
Project Number: 676556
Project Coordinator: Institute for Research in Biomedicine (IRB Barcelona)
Project start date: 1/11/2015
Duration: 36 months

Deliverable 4.5: Data Access API Specification and Implementation

Lead beneficiary: The European Bioinformatics Institute (EMBL-EBI)
Dissemination level: PUBLIC

Due date: 31/10/2017
Actual submission date: 31/10/2017

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 676556.

Document History

Version	Contributor(s)	Partner	Date	Comments
0.1	Mark McDowall	EMBL-EBI	29/09/2017	First draft
0.2	Andy Yates	EMBL-EBI	04/10/2017	Second Draft
0.3	Laia Codo	BSC	27/10/2017	Added Service Deployment
1.0	Mark McDowall	EMBL-EBI	30/10/2017	Final report
			31/10/2017	Approved by Supervisory Board

Table of Contents

1 EXECUTIVE SUMMARY

The following document describes the Data Management (DM) Application Programming Interface (API) and the changes that have been introduced since the reporting of D4.4. The DM API has been integrated into a RESTful API allowing physically separated clusters to be able to efficiently interact with our DM service. This has required work to develop an AAI layer that is able to quickly identify a user, authenticate them and provide them access to only files they have authorisation to see. Further extensions have been added to the RESTful DM layer, developed around the concept of microservices. These provide access, either selectively or completely, to the files that are stored within the DM layer. This access allows the VRE and COMPSs to make the right files available to pipelines and tools, but also to visualisers integrating the multitude of data types involved in understanding the folding of chromosomes.

2 INTRODUCTION

Resolving the architecture of chromosomes within the nucleus requires a large array of experimental methods, the result of which is a large number of files that need to be tracked and monitored. Keeping records about how files were generated, where they came from and what tools were run are important in tracking the provenance of a piece of knowledge. Knowing how the information was generated is crucial not only for accurate reporting of the work but also in knowing how to interpret the results. These form the principles of FAIR (findable, accessible, identifiable, reusable) data access.

On top of experimental data and metadata storage described in Deliverable 4.4 (D4.4) an Application Programming Interface (API) has been developed to allow the Virtual Research Environment (VRE) to communicate with a Data Management (DM) layer; a software package responsible for noting the physical location of data and associated metadata. MuG's adopted computation infrastructure means our VRE has the capability of being deployed across multiple cloud environments requiring the tracking of files across multiple disparate locations. We have developed a RESTful interface to provide a seamless layer for interaction with the DM API. With the introduction of a RESTful interface there is also the need for an authentication layer to ensure that users can only have access to files that they are entitled to view.

3 DATA ACCESS API

The DM API is based around interacting with the MongoDB data structure that was described in D4.4 section 4.1. The code is implemented in Python 2.7.10+, but is also compatible with Python 3.5. The API plays a dual role, firstly as a simple way for services to interact with the data, the second is to enforce the way in which the data can queried. The first part removes the complexities of needing to implement custom interactions with the MongoDB for every service that want to get information from the database. The second enforces that service that wish to interact with the database have a matching user_id to retrieve and store data removing the risk of exposing inappropriate files to another user.

3.1 Data Structure Updates

Since D4.4 there have been modifications made to the data structure to incorporate more information and ease the integration of the DM API within the existing VRE. Table 3.1.1 describes all of the primary parameters that are stored within the DM, the newly introduced parameters include path_type, parent_dir and size and are highlighted in blue. The path_type can be one of file, dir (directory) or Universal Resource Locator (URL). Using URLs means files that are not located on servers can be flagged as a file within the remit of the MuG consortium/VRE. This can include links to repositories or servers outside of the project that are hosting data of interest to analysis methods within the VRE.

One of the new requirements is for the ability for a VRE user to organise their files into a directory like structure independently from how they are organised on the host server. Directory structures are a well known and used paradigm for organising complex datasets in cloud environments e.g. Dropbox, Google Drive, ownCloud. The introduction of parent_id allows for the existence of a directory structure in combination with the path_type being set to 'dir'. To help the VRE best decide where to run a given pipeline a file size (bytes) parameter has also been included. This will help as it will allow the VRE to balance between running the pipeline local to the data and incurring the cost of moving data versus waiting for other processes to finish.

Along with each file getting a timestamp for the creation of an entry, within the metadata there is also an expiration_date. This is so it is possible to automatically remove old metadata files after a defined period of time to reduce old files getting left on the system. Currently the expiration time is not being enforced while still in the early stages of the project.

Parameter	Required	Description
user_id	YES	The unique user ID for whom the file is associated with
file_id		This is an auto-generated ID that is created when the data is entered. The ID is unique to the file and the user.
file_path	YES	Location of the file either within the file system or a URL to an archive or repository.
path_type		Defines if the file_path is a file or directory
parent_dir	DEPENDENT	If the path_type is a directory then this parameter defines the _id of the parent directory.
file_type	YES	File format. The current accepted file types are described in section 5.
size		Size of the file (bytes)
data_type		This describes the type of data that is in the file. This is helpful as there are several formats that can have different data. For example FASTQ data can be related to RNA-seq, MNase-Seq, ChIP-seq, WGBS, etc.
taxon_id	YES	The taxonomic ID of the species from which the sample data was taken.
compressed		Whether the file has been compressed. Type of compression used depends on the format in question.
source_id		List of the file IDs that were used during the creation of this file.
meta_data	DEPENDENT	There are cases where additional data is required for some files that is not relevant to other file types. Files that have been generated and are dependent on alignments require that the meta_data has an 'assembly' key with the assembly for which the alignment was made against
creation_time		This is the time inserted by the API and is not required from the user.

Table 3.1.1: *List of parameters stored by the data management API and whether these parameters are required. Those highlighted in blue are new since D4.4*

3.2 DM API Functions

The DM API covers all from from loading filtering and modifying and deleting.The code in block 3.2.1 highlights opening a DM API handle and inserting an entry into the DM using the API.

```
from dmp import dmp
dm_handle = dmp(test=True)
file_id = dm_handle.set_file(
    user_id='test',
    file_path='/tmp/test.fastq',
    path_type='file',
    parent_dir='',
    file_type='FASTQ'
    size=1234567890,
    data_type='hic',
    taxon_id=9606,
    compressed=False,
    source_id=[],
    meta_data={'assembly': 'GRCh38'}
)
```

Code Block 3.2.1*: Example code showing how to load a file record into the DM using the API*

Once a file has been logged within the DM it is possible to search for relevant files. All queries require the user_id as a minimum piece of information. All of the functions to filter the available files return a list of python dictionary objects detailing the metadata associated with a given file. The only exception to this is the get_files_by_id which returns only the dictionary object containing the metadata. The set of filter functions that are available are listed in table 3.2.1.

It is also important for the user to be able to update the information within the DM to either add additional information that was not available at the time of submission, correct already stored information or to remove files from the system. To allow for this there are the functions that are listed in table 3.2.2 to provide a level of data management.

Whenever there is a change to the data within the DM for a given file the entry is checked to ensure that all the required data is still defined. If a piece of information fails the validation check this raises an error and wont update the record unless a valid entry is submitted.

Function	Parameters	Description
get_files_by_id	user_id, file_id	Return the metadata about a single specific file
get_files_by_user	user_id	List all files associated with a given user
get_files_by_path	user_id, file_path	Return the files that have a specific file_path
get_files_by_file_type	user_id, file_type	List all files based on the file type (fasta, fastq, bam, pdf, lif, etc)
get_files_by_data_type	user_id, data_type	List all files that relate to a specific data type (chip-seq, rna-seq, etc)
get_files_by_assembly	user_id, assembly	List all files that have been aligned with a specific assembly
get_files_by_taxon_id	user_id, taxon_id	List all file that are related to a specific species irrespective of the assembly that they have been aligned with
get_file_history	user_id, file_id	List of all files that have been used to generate a given results file

Table 3.2.1: *List of functions available for filtering a user's files.*

Function	Parameters	Description
remove_file	user_id, file_id	Removing a file from the DM records.
add_file_metadata	user_id, file_id, key, value	Add additional metadata to an already existing record within the metadata. This allows a user to add extra information including if it has come from, or been included in a publication.
remove_file_metadata	user_id, file_id, key	If metadata has been erroneously added then it can be removed
modify_column	user_id, file_id, key_value	Allows for the subsequent modification of the main data columns listed in table 3.1.1.

Table 3.2.2: *List of functions for editing entries within the DM.*

3.3 Testing, Code Quality and Documentation

As the code base grows and the number of developers increases it is important to maintain high standards when it comes to the quality of the code, documentation and ensuring that the functions that have been written work as expected. This is vital when it comes to introducing changes to the code base and having confidence that this will not break other functions already in the API. To help with this there has been the adoption of of documentation standards defined as part of MS14, but these have also been expanded on and detailed within ReadTheDocs to define how code within the project should be written:

- http://multiscale-genomics.readthedocs.io/en/latest/coding_standards.html

To make sure that the functions in the API return valid results we have implemented pytest and written tests for each of the functions to ensure that they return valid results. To do this it uses the mongomock python module to simulate a MongoDB, which is then populated with mock entries and queries are made against this dataset. There are also example files kept with the Git repository and where the sample files would be too large there are generators that are able to create an example file of the matching file type. This testing has been automated using TravisCI. Everytime there is a push to the GitHub repository the full suit of tests are run by TravisCI to ensure that there are no failures.

The final approach to creating a clean and simple API has been the introduction of linting of the code. This is a process to check that the code that has been written conforms to standard, in the case of python this is PEP8 (https://www.python.org/dev/peps/pep-0008/). For the API we use pylint to check that the code matches the PEP8 standard. This can be done locally before committing, but is also run automatically by Landscape.io, which tracks if there has been a change in quality between one commit and the next when there has been a push to the GitHub repository.

One part of the PEP8 standard it to make sure that there is documentation for all publically available functions. This means that at minimum there needs to be a short description of the function so that it can display within the matching ReadTheDocs portal. Defined in the coding standards documentation there are examples about how to set out the documentation. There are also notes about how to apply the Apache 2.0 License to a new file or repository.

4 RESTful ARCHITECTURE

Having the VRE and compute as a centrally managed infrastructure means that it is possible to keep a tight control over the tracking of files. As the infrastructure grows, and compute and storage resources are introduced that are not local to the VRE it is important to be able to provide a mechanism to be able to update the DM records from physically separated machines. It is also a requirement that visualisers are able to access the results files that have been generated so that information can be integrated in an accessible manner for the user. The development of RESTful servers also removes the dependence of downstream services needing to be written in Python. This allows visualisation tools access to the data while using the most appropriate language for the task.

To facilitate this a RESTful approach has been taken to allow access to the DM and the files that are being tracked. The section describes the approach taken for developing a RESTful API, authentication of the users and what endpoints are available.

Development of each server is based on the HATEOAS (Hypermedia as the Engine of Application State) paradigm. Responses from the servers are self describing and links are provided to related endpoints. When no parameters are provided then a list of the required parameters are given.

4.1 Structure

The RESTful service has been designed around the concept of micro services. This relies on multiple, small, dedicated services that focus on serving a specific function. If a new function is required then a new service and matching endpoints are developed. Each service acts as its own independent server, if there are dependencies on another service then this is via RESTful calls to that service. This function is provided by the mg-rest-service server. It acts as a head server that has a list of the available services that it can ping to ensure that they are functioning, but also provide an up to date list of next level down endpoints.

Microservices allow our developers to choose the most appropriate language for a given service. However, given that the majority of experience is within Python this has been the default language as it allows easy integration with the DM API. We have chosen Flask as our default framework for developing microservices.

4.2 Authentication

As anyone can query a URL it is important to be able to authenticate a user is who they say they are to prevent one user see another user's files without permission. An OAuth server has been set up that can use either LDAP, Google IDs or LinkedIn. In the future we intend to allow access via a user's ELIXIR to identify. The authentication server issues tokens to the user, these are submitted in the header of a request to the RESTful server. The server then checks the token with the authentication server. If it is a valid token then the authentication server returns the ID of the user which can then be used to query the DM to retrieve the relevant information. If the token is invalid or not supplied then the request fails and raises the relevant HTTP error response code.

This means that the user_id is never used in the URL, users always have to provide a valid token on all requests where they are retrieving data and if a user_id is known it won't be used to access data as all requests are passed to the authentication server first. The tokens are also time limited to ensure that if there is a leaked token then this is only valid for a limited time period. Likewise a user is also able to manually regenerate a token if a previous token has been exposed.

The authentication of tokens with the authentication server is performed using Python decorators around the respective GET, PUSH, DELETE and PUT functions (see Code Block 4.2.1), in the case of the MuG RESTful services it is called “@authorized”. To access the @authorized decorator for validating tokens and returning the user_id each RESTful API needs to import the wrapper from the mg-rest-util repository.

```
from flask_restful import Resource
from rest.mg_auth import authorized
class example_endpoint(Resource):

    @authorized
    def get(self, user_id):
        return True
```

Code Block 4.2.1: *Example use of the @authorized decorator*

Decorators work by performing a function before the decorated function subsequently passing the values that they acquire, in the case of code block 4.2.1 this is the user_id. The passing of an authentication token needs to be done in the header of an HTTP request, an example if this is shown in code block 4.2.2 using an example endpoint from table 4.3.1.

```
curl -H "Authorization: Bearer <token_string>" --url
"<url_root>/mug/api/dmp/files?assembly=GRCh38"
```

Code Block 4.2.2: *Example use of including a validation token in the header of an HTTP request.*

4.3 RESTful Interaction With The DM API

The main RESTful server for interacting with the DM API is the mg-rest-dm server. This maps all of the filtering, submission, modification and deletion functions to the respective endpoints while relaying the standard outputs of the DM API to the user. As described in section 4.2.1 it relies on a valid token to be passed with each request to authenticate the user as well as knowing who the file should belong to when querying the DM API. The main endpoints within this service are listed in table 4.3.1.

Endpoint	Parameters	Description
/mug/api/dmp		Lists the available endpoints that can be requested.
/mug/api/dmp/ping		Returns the name of the service, version, author, license, description, link to the root for the service and the status.
/mug/api/dmp/files	region, assembly, file_type, data_type, by_user	There are 5 filter types to retrieve files for a given user. Region uses the positional HDF5 meta index described in D4.4 section 5.1.1 and requires the assembly parameter as well. Querying can be done by assembly , file_type or data_type. by_user returns all files for the current user.
/mug/api/dmp/file_meta	file_id	Returns a python dictionary of the selected file
/mug/api/dmp/file_history	file_id	Returns a list of the files required to generate the specified file_id.

Table 4.3.1: *List of the endpoints available within the mg-rest-dm server. With the exception of the root URL and the ping endpoint all require a valid authentication token in the header to identify the user_id.*

4.4 Data Retrieval

There are 3 services so far that have been developed to handle the exchanges of data between the DM and the user. Below we further describe the services that have been generated so far.

4.4.1 mg-rest-file

This is the main service for streaming files to the user, the endpoints have been listed in table 4.4.1.1. Once the user has identified the files that they wish to acquire, whether this is to download, visualise or use as part of a pipeline, this service acts to handle the passing of that data from where it is located to the user. The reason for having this interface for the RESTful service is to provide a single gateway access to files that may be located on multiple servers.

Endpoint	Parameters	Description
/mug/api/dmp/file		Lists the available endpoints that can be requested.
/mug/api/dmp/file/ping		Returns the name of the service, version, author, license, description, link to the root for the service and the status.
/mug/api/dmp/file/whole	file_id	Streams the whole file to the user
/mug/api/dmp/file/region	file_id, chrom, start, end	For genomic position file types (bed/bigBed, wig/bigWig and GFF3/Tabix) it is possible to return only the features in a given region.

Table 4.4.1.1: *List of the endpoints available within the mg-rest-file server. With the exception of the root URL and the ping endpoint all require a valid authentication token in the header to identify the user_id.*

4.4.2 mg-rest-adjacency

This is a specific service for serving adjacency matrix data that has been generated by TADbit and saved in the HDF5 file format described in D4.4 section 5.1.2. The available endpoints are listed in table 4.4.2.1.

Endpoint	Parameters	Description
/mug/api/adjacency		Lists the available endpoints that can be requested.
/mug/api/adjacency/ping		Returns the name of the service, version, author, license, description, link to the root for the service and the status.
/mug/api/adjacency/GetDetails	file_id	Lists the chromosomes and resolutions in the dataset
/mug/api/adjacency/GetInteractions	file_id, chrom, start, end, res, limit_chr, limit_start, limit_end	Lists all of the interactions for a given region. This can be limited to a given subset on another chromosome or the same chromosome.
/mug/api/adjacency/GetValue	file_id, res, pox_x, pos_y	Lists the peak count for a given resolution for a given x and y position listed in the output from the GetInteractions endpoint.

Table 4.4.2.1: *List of the endpoints available within the mg-rest-file server. With the exception of the root URL and the ping endpoint all require a valid authentication token in the header to identify the user_id.*

4.4.3 mg-rest-3d

This is a specific service for providing access to the 3D model predictions that have been made by TADbit. The output is in the form required by TADkit for display with the browser. The data is stored within the HDF5 file format described in D4.4 section 5.2.1. The available endpoints are listed in table 4.4.3.1.

Endpoint	Parameters	Description
/mug/api/3dcoord		Lists the available endpoints that can be requested.
/mug/api/3dcoord/ping		Returns the name of the service, version, author, license, description, link to the root for the service and the status.
/mug/api/3dcoord/resolutions	file_id	List all resolutions that models have been calculated for
/mug/api/3dcoord/chromosomes	file_id, res	For a given resolution list the chromosomes that models have been generated for
/mug/api/3dcoord/regions	file_id, res, chrom, start, end	List regions within a given chromosomal area that have structural predictions
/mug/api/3dcoord/models	file_id, res, region	Return all models from a region at a given resolution
/mug/api/3dcoord/model	file_id, res, region, model_id	Return a single model for a given resolution from a specific region

Table 4.4.2.1: *List of the endpoints available within the mg-rest-file server. With the exception of the root URL and the ping endpoint all require a valid authentication token in the header to identify the user_id.*

4.5 Testing, Code Quality and Documentation

For each service that is developed the code matches the same standards that are defined in section 3.3 for coding standards, documentation and testing. Each service has a repository name of "mg-rest-*" with documentation made available from the ReadTheDocs service. TravisCI and Landscape.io are also used to ensure that the code quality is as high as possible. For testing it uses pytest, and queries the endpoints in a similar manner to the tests described in section 3.3 for the DM API.

5 SERVICE DEPLOYMENT

As the project is developing, the distributed infrastructure that underlies the compute platform for the VRE is coupled with matching MuG data repositories. The RESTful architecture of DM services make them fully accessible to VRE, who is responsible of centralizing metadata registration and staging data across MuG infrastructures.

The updated DM data structure is fully compatible with VRE needs, and the metadata management is now capable of being handled by the DM API. VRE is a client for the microservice, and it is granted authorization via the access token of the end user.

MuG infrastructures are currently deployed in BSC, IRB and EMBASSY clouds. The DM streaming file service (mg-rest-file) offers a uniformed entry to infrastructure storage facilities, allowing them to be securely and remotely accessed by VRE, at the same time that provides Uniform Resource Identifiers (URI) for MuG resources. Internally, these resources are locally made available to the virtualized tool instances by means of a dynamic contextualization system based on the network file system (NFS) protocol, as detailed in-depth in the D5.2.

Demanding challenges have arisen with the development of a distributed data system, not only data accessibility, privacy policies, hardware heterogeneity and public identifiers, mainly addressed in our current DM plan, but also issues more intrinsically related to data transfer, such as data synchronization. The implementation of an abstraction layer able to transparently deal with data replication and coherence is under study, strategy that could definitely allow user data staging in and out on the selected MuG infrastructure in an secure and effective way. In particular, the IRODS technology, used already by EUDAT e-infrastructure is being tested to take care of lower level data transfer layer.

6 CONCLUSIONS

The development of the DM API acts to support the work of the VRE and launching pipelines within the COMPSs infrastructure. To serve the VRE, provide results for job submission and be responsive in a RESTful environment it is crucial to have a dependable API. The DM API has been built in collaboration with WP3 and WP5 so that it is best able to serve their needs. With openly available documentation, continuous integration testing and automated code quality checking we have tried to minimise the technical debt for future users and developers.

The development of a RESTful interface allows for the expansion of the MuG compute facility outside of the current infrastructure. It also allows the development of external visualisation tools to view a user's data. Allowing for access by external tools opens up the potential for community developers to create new visualisations that aid in the understanding of experimental data.

7 ANNEXES

7.1 Abbreviations

DAC:	Data access committee
EGA:	European Genome-phenome Archive
ENA:	European Nucleotide Archive
FISH:	Fluorescence *in-situ* hybridisation
MINSEQE:	Minimum Information about a high-throughput Sequencing Experiment
OME:	Open Microscopy Environment
PDB:	Protein Data Bank
PMES:	Programming Model Enactment Service
SRA:	Sequence read archive
VRE:	Virtual Research Environment
WGBS:	Whole Genome Bisulphate Sequencing