ADT Generator Rest API

API specification version 1.0.0

- Compile
- Compose

Repository: https://github.com/UoW-CPC/adt-generator-api.git

Prepared by: The University of Westminster

The link for editing the document if it is required:

https://docs.google.com/document/d/17BnZmhsPvmcwem9EyXSTUissXRAz9NW3ornpWKea3qc/edit?usp=sharing

Platform/tools: Python and Flask

Introduction:

The REST API is a part of the ADT generator for the DigitBrain project that provides different functionalities by using HTTP requests to access and use data across the whole components of the ADT generator.

In the core architecture of the API, compile and compose modules are included in order to deal with the metadata. The metadata is expected to be in the form of files or payload. In the current version of the API, three Db assets are considered: algorithm, infrastructure and microservice. Each of which are embedded within a specific function in the Python programming language.

Compile:

Endpoint	api/v1/.../compile
Request payload (JSON)	- Asset type (Microservice, Algorithm, Infrastructure) - Metadata key/value pair
Handlers/functions	Calling and verifying compile libraries
Response (JSON)	Description template (Preview as YAML through the publishing interface)
Error Handling	Raising exception in Python functions and jsonify error considering appropriate status codes

Table 1 - A high-level overview of the compile procedure

A high-level overview of the compile module is shown in table 2 that specifies the endpoint for each specific compile function, shown in table 2. Each function in the compile section, needs to invoke proper libraries and produces a response as a JSON that must be previewed as YAML via the publishing interface. Indeed, in the current version of the API, the error handling is not being considered.

Three compile functions for each type of metadata are presented in table 2. Each metadata has a specific route/endpoint with HTTP requests GET and POST.

function	Route / endpoint	methods
infrastructure_dt ()	api/v1/idt/compile	“GET”, “POST”
microservice_dt()	api/v1/mdt/compile	“GET”, “POST”
algorithm_dt()	api/v1/algodt/compile	“GET”, “POST”

Table 2 - Compile functions

The responsibility of the compile functions is to get the metadata, and identify it based on the type of request. In other words, there could be multiple metadata with corresponding types and for each type, there could be multiple instances of all

metadata. Hence, considering the input as a payload, it is possible to get all metadata and for each of which invoke the compile library. Indeed, when one of the functions is invoked inside a particular route/endpoint, the API must extract all the information regarding the metadata and identify what kind of metadata is.

compose:

The compose function (see table 3) of the REST API gets all metadata and asks it to the publisher interface and then creates all different templates and packs them all as an archive (CSAR) file. Similar to the compile function, the compose has its specific endpoint that interacts with the publishing interface through the POST request. As it has been already mentioned, the results of invoking appropriate compile libraries based on the type of requests are packed as a CSAR archive file.

Endpoint	api/v1/.../compose
Method	POST
Request payload (JSON)	DMA Tuple (Reference to three types of metadata mentioned in table 2) **(Probably IDS for different asset metadata)**
Response (JSON)	Description template (Preview as YAML through the publishing interface) **(Probably ID of the nexus asset)**
Handlers	Workflow Engine
Response (JSON)	Serialized CSAR Archive
Error Handling	Raise exception in Python and return response as a status code (The message should be produced by the compile library)

Table 3 - Comose

It is noted that when the DMA Tuple is composed by the DMA composer the Publishing Interface makes a request to the ADT Generator. Accordingly, the request payload includes all the IDs of the various assets that make up the DMA Tuple. Then, the ADT

Generator uses the IDs to retrieve metadata from the Db Asset Metadata Registry and creates CSAR archive as a nexus artefact.

Logging operations:

The REST API for the ADT generator benefits the logging functionality for all operations. This process is done by loading and initializing the configuration. The logging configurations are shown in Table 4. Logging in Flask uses the same standardized Python logging framework. The benefit of having a logging framework by standardized library module in Python is that all Python modules can interact and participate during logging. The messages are logged using app.logger and the Flask object instantiating is done with the same variable as that of app.name. Clearly speaking the application utilized different levels of logging such as DEBUG, INFO, and ERROR based on the logging principles of FLASK.

Logging levels	Functionality
logger.debug	used to find the reason in case the program is not working as expected or an exception has occurred.
logger.info	used to log the information in case that the program is working as expected.
logger.error	used to log the information in case that the program is not working as expected.

Table 4 - Logging levels in API

The compile and compose functions are expected to get necessary configurations from the config dictionary which is usually a YAML file.

The log history for all the operations in the REST API is stored in an structured way in a log file once the application is executed. The details of the settings are described in Table 5.

	Functionality
Log formatter	log formatters allow to specify the layout of the messages when the logger writes them
basicConfig()	The basicConfig() configures the root logger. It does basic configuration for the logging system by creating a stream handler with a default formatter. The debug(), info(), warning(), error() and critical() call basicConfig() automatically if no handlers are defined for the root logger.
Streamhandler()	StreamHandler sends log records to a stream. If the stream is not specified, the sys.stderr is used.
FileHandler()	Creates a log file which includes the summary of the logging operations (Usually a high level represenations)

Table 5 - Logging in API