How Composable is the Web?
CSCI724 - Web Services and Service Oriented Computing
Group 5 Course Project

Original Paper - https://ieeexplore.ieee.org/document/9885779
GitHub Repository - https://github.com/JoshiArchit/CSCI724-Course_Project

Authors -

1. Archit Joshi (aj6082@rit.edu)
2. Athina Stewart (as1896@rit.edu)
3. Chung-An Huang (ch1949@rit.edu)

Table of Contents

1. Introduction

This technical report describes the attempt at implementing the approach to determine the composability of web services that are specified using openAPI/swagger specification. This document is intended to give you an overview of the technical stack and a brief description of the task each script performs. Most of the code utilizes the dictionary data structure as inherently API specifications follow a json specification which are key:value pairs.

The basic idea for determining web composability in layman terms is to determine if the produced data from one schema can be consumed by another schema. This would in theory make it possible to create a chain (or composition) of web-services where every web-service would feed data to every subsequent web-service which can consume it.

2. Requirements

1. Python 3.0 or greater with support for json, PyYaml libraries.
2. Internet connection for dataset collection.
3. For the final submission we are exploring incorporating MongoDB as a document-storage solution.

Note - External libraries that need to be pip-installed have been listed down in requirements.txt file in the repository. Before running the program use `pip install -r requirements.txt` to install them.

3. Execution Instructions and Control Flow

1. Data Collection needs to be performed before any scripts from the repository are executed. This is a time consuming operation that needs to be performed by itself (See Section 4 below).
2. Each script can run as its own individual stub. But to make things easier perform_experiment_from_paper.py has been provided as a single-entry point that will perform the entire experiment from data cleaning, parameter extraction, data storage and the matching process.
3. For the final submission we are attempting to explore document-based storage using MongoDB that might speed up indexing and lookups when performing the matches.
4. Code execution takes approximately 5 hours to complete. An additional hour or 2 for data extraction.

4. Data Collection

Note - Data collection is not included in the scripts themselves as it's a resource and time consuming operation. This step of the experiment needs to be performed by itself before proceeding to data cleaning.

Resource for data collection -
https://www.assetnote.io/resources/research/contextual-content-discovery-youve-forgotten-about-the-api-endpoints

As per the author's instructions we utilize the resource mentioned above which has instructions on how to compile and download the data which gives us about ~71,000 API descriptions for our experiment. Given the limited resources and expensive caching and indexing solutions, Currently we focus on a small subset of the dataset which contains the API specifications collected from APIs.Guru (https://apis.guru/).

We execute the following commands in order to download the dataset as mentioned in the original article -

```
wget https://api.apis.guru/v2/list.json ; cat list.json | jq -r
'.[]["versions"][]["swaggerUrl"]' > swagger-to-scrape
wget -i swagger-to-scrape
```

5. Data Cleaning - cleanDataSet.py

The authors have strict requirements so as not to bias the composability results. Data cleaning portion of this experiment is focused towards reinforcing these rules against each API specification and removing any specifications that do not adhere to them. The script cleanDataSet.py houses helper functions.
We implement the following checks -

1. **GET/POST Parameter check** - APIs need to have at least one endpoint with a description of schema of consumed (GET) or produced data (POST). To satisfy this condition we scan through the data and check if there is a 'get' or 'post' request in the paths. This condition ensures that the results are not biased towards mismatches as there can be requests which are just notification requests.

```
def extractFeatures(data):
    """
    Authors note - APIs need to have at least one endpoint with a description of schema
of consumed
    or produced data. To satisfy this condition we scan through the data and check if
there is a
    'get' or 'post' requests in the paths.

    :param data: JSON data of the API
    :return: True if the API has at least one 'get' or 'post' request in the paths
    """
    try:
        if 'paths' in data.keys():
            paths = data['paths']
            for path in paths:
                if 'get' in paths[path] or 'post' in paths[path]:
                    return True
    except KeyError as e:
        # print("Error in extractFeatures")
        return False
```

2. **Valid Endpoints** - APIs need to have at least 3 paths in order not to bias the average composability rate. To satisfy this condition we scan through the data and check if there are at least 3 paths in the API description.

```
def get_num_paths(data):
    """
    Authors note - APIs need to have at least 3 paths in order not to bias the average
composability rate. To satisfy this condition we scan through the data and check if there
are at least 3 paths in the API description.

    :param data: JSON data of the API
    :return: True if the API has at least 3 paths
    """
    try:
        if 'paths' in data.keys():
            if len(data['paths']) > 2:
                return True
    except KeyError as e:
        return False
```

3. **Realistic URL -** APIs need to have at least one realistic server (OAS v3) and base path (Swagger) URL. To satisfy this condition we scan through the data and check if there is a 'host' or 'servers' key in the data. We have continued the data cleaning with the assumption that if a base-path key exists its default value is held to be '/'.

```
def get_server(data):
    """
    Authors note - APIs need to have at least one realistic server (OAS v3) and base path
(Swagger)
    URL. To satisfy this condition we scan through the data and check if there is a
'host' or
    'servers' key in the data.

    :param data: JSON data of the API
    :return: True if the API has a 'host' or 'servers' key
    """
    try:
        if 'swagger' in data.keys():
            if data['host']:
                return True
        else:
            if 'servers' in data.keys():
                for server in data['servers']:
                    if 'url' in server.keys() and server['url'] != '':
                        return True
    except KeyError as e:
        return False
```

6. Parameter Extraction - extractMethodParameters.py

As discussed in the introduction section, web service composition requires one web service to produce data which can be consumed by every subsequent web service. This makes parameter extraction a crucial component for performing matches. We also need to ensure that matches are performed between one set of POST parameters and another set of GET parameters.

The authors provide a list of parameters that do not contribute towards composability. These include - *description, title, default, example, readOnly, writeOnly, example, title, enum, required, minimum, maximum, multipleOf, minItems, maxItems.* Adhering to these restrictions listed down by the authors we extract the parameters from each schema and create two separate specifications (GET and POST). The new specifications have a structure as follows -

GET operation	**POST operation**
api_path { get_operationNumber_parameters { parameter_1 : value, parameter_2 : value, parameter_3 : value } }	api_path { post_operationNumber_parameters { parameter_1 : value, parameter_2 : value, parameter_3 : value } }

Pseudocode for GET operation-

```
1. Parse JSON object into dictionary using json library in python.
2. new_schema = empty dictionary that will hold the new specification
3. exclude = list of keys to exclude as per the restrictions
4. if 'parameters' key exists in the original schema:
       if parameters contains attribute $ref its a complexType or referred attribute:
              Extract the key name using string operations -> key = key.split('\')[-1]
              Go to 'schema' key to find values:
                     if key is not in exclude:
                            add key:value pair to new_schema
                     else:
                            continue to explore the next key
       else its a normal list of parameters:
              for key in parameters:
                     if key is not in exclude:
                            add key:value pair to new_schema
                     else:
                            continue to explore the next key
```

Pseudocode for POST operation-

```
1. Parse JSON object into dictionary using json library in python.
2. new_schema = empty dictionary that will hold the new specification
3. exclude = list of keys to exclude as per the restrictions
4. if 'parameters' key exists in the original schema (parameters passed in url):
       if parameters contains attribute '$ref' its a complexType or referred attribute:
              Extract the key name using string operations -> key = key.split('\')[-1]
              Go to 'schema' key to find values:
                     if key is not in exclude:
                            add key:value pair to new_schema
                     else:
                            continue to explore the next key
       else its a normal list of parameters:
              for key in parameters:
                     if key is not in exclude:
                            add key:value pair to new_schema
                     else:
                            continue to explore the next key
5. if 'requestBody' key exists in the original schema (parameters passed as http body):
       repeat steps listed in Step 4 loop.
```

7. Determining Composability - perform_experiment_from_paper.py

The final step is to perform the composability test based on certain parameters and drawing statistics and conclusions from the experiment numbers. Perform_experiment_from_paper.py serves as a driver script to execute the entire experiment. Code for each individual matching criteria can be found in the scripts denoted in the corresponding sections.
The authors have listed down a few criterias which help determine the composability of two web services. In our current iteration we explore two of those approaches and plan to extend our experiment to more approaches before our final submission.

Data-Type matching (data_type_match.py) -
For two APIs to have a data-type match, the keys in which they share must have the same type. For example, in the description below, the operation 1 GET matches with the operation 2 POST but does not match with operation 3 POST due to a data type mismatch for the age attribute.

Operation 1	**Operation 2**	**Operation 3**

`operation_1 {`
`    get_0_operation {`
`        name : string,`
`        age : int,`
`        height : float`
	`operation_2 {`
	`    post_0_operation {`
	`        name : string,`
	`        age : int,`
	`        height : float`
		`operation_3 {`
		`    post_0_operation {`
		`        name : string,`
		`        age : float,`
		`        height : float`

`    }`
`}`
	`    }`
	`}`
		`    }`
		`}`

Table 1 - Operation examples for matching algorithm

```python
def compare_dicts(dict1, dict2):
    """
    This function compares the values of the keys in the dictionaries. For all the keys
    that are common to both dictionaries, if the values are the same, return True
    :param dict1: post request dictionary
    :param dict2: get request dictionary
    :return: True if the values of the keys in the dictionaries are the same
    """
    # Get the set of common keys
    common_keys = set(dict1.keys()) & set(dict2.keys())

    # Check if values for common keys are the same in both dictionaries
    for key in common_keys:
        if dict1[key] != dict2[key]:
            return False
    return True


def match_data_types(doc_1, doc_2):
    """
    This function compares the data types of the parameters in the post
    request of doc_1 with the data types of the get request of doc_2
    :param doc_1: first API description
    :param doc_2: second API description
    :return: match count
    """
    total_match_count = 0

    for method_type in doc_1:
        for operation_type in doc_1[method_type]:
            if "post" in operation_type:
                post_parameters = doc_1[method_type][operation_type]
                for method_type_2 in doc_2:
                    for operation_type_2 in doc_2[method_type_2]:
                        if "get" in operation_type_2:
                            get_parameters = doc_2[method_type_2][operation_type_2]
                            # for all the keys that are common to both post_parameters
and
                            # get_parameters, if the values are the same, increment match
count by 1
                            if compare_dicts(post_parameters, get_parameters):
                                total_match_count += 1
    return total_match_count
```

Property-Name matching (property_name_match.py) -
For two APIs to have a property-name match, the keys must be exactly the same. Here, we do not consider the values, aka, the data types of the keys. For example, in the same example above (Table 1), we have a property match between Operations 1, 2 & 3.

```
def compare_dicts(dict1, dict2):
    """
    This function compares the keys of the parameters in the post requests of
    doc_1 against the keys of the parameters in the get requests of doc_2. If the
    keys of the post parameters are exactly the same as the keys of the get
    parameters, return True
    :param dict1: post request dictionary
    :param dict2: get request dictionary
    :return: True if the keys of the post parameters are exactly the same as the keys of
the get parameters
    """
    # if the keys of dict1 are exactly the same as the keys of
    # dict2, return True
    if dict1.keys() == dict2.keys():
        return True


def match_property_names(doc_1, doc_2):
    """
    This function compares the names of the properties of the parameters
    in the post requests of doc_1 against the names of the properties
    of the parameters in doc_2
    :param doc_1: first API description
    :param doc_2: second API description
    :return:
    """
    total_match_count = 0

    for method_type in doc_1:
        for operation_type in doc_1[method_type]:
            if "post" in operation_type:
                post_parameters = doc_1[method_type][operation_type]
                for method_type_2 in doc_2:
                    for operation_type_2 in doc_2[method_type_2]:
                        if "get" in operation_type_2:
                            get_parameters = doc_2[method_type_2][operation_type_2]
                            # compare the keys of the post parameters with the keys
                            # of the get parameters
                            # if all the keys of the post parameters are in the get
                            # parameters, increment the match count by 1
                            if compare_dicts(post_parameters, get_parameters):
                                total_match_count += 1
    return total_match_count
```

Results and conclusions

Currently we are fine tuning the model to perform stricter cleaning and validations in order to get as close as we can to the results proposed by the authors. As the datasets are updated daily and we are only using a fraction of the same due to hardware constraints it would be difficult to obtain those exact numbers.

Despite this constraint we try to answer the following questions proposed by the authors -

1. Are most APIs compatible on a syntactic or PropertyName Level?

   After our analysis, we found that 54.04% of APIs are compatible. On the contrary, the authors found 92.6% of APIs to be compatible. This could be attributed to the fact that we did not use the entire dataset as the authors did.

2. How compatible are Web APIs?

   While we plan to fully answer this question by determining the percentage of matches that are formed on both data-type match and property-name match, we can for now say that at least half of the API's are compatible on a property-name match. This would still require manipulation of the data types in the middleware.

3. On which level are most API compatible?

   The authors claim that there are less APIs which use the same property names than the ones that share the same property type. We have come to the opposite conclusion as 54.04% were compatible on the property-name level and only 0.31% were compatible on the data-type level. Additionally, we make the claim that APIs are most compatible on the first level of their parameter descriptions. The deeper we go into the document, we would risk losing some property level matches.

4. What is the likelihood that a random pair of APIs has at least one compatible endpoint?

   Though the initial sample size was large, after cleaning we only used 2,908 API descriptions. From this, we then extracted 58,417 GET operations and 240,354 POST operations. This is a relatively small sample size and given the small number of data-type matches, the number of APIs that satisfy both types of matches is likely to be small. Therefore, the likelihood of a random pair of APIs having at least one compatible endpoint is not zero, but it is low. This is only the result from our experiment, and does not necessarily negate the work and findings of the original authors.