SOS – Simple Orchestration of Services

Ricardo Queirós[1] and Alberto Simões[2]

1 **ESMAD, Polytechnic of Porto, Porto, Portugal**
`ricardoqueiros@esmad.ipp.pt`
2 **Centro Algoritmi, University of Minho, Braga, Portugal; and Instituto Politécnico do Cávado e do Ave, Barcelos, Portugal**
`asimoes@ipca.pt`

Abstract

Nowadays, we continue to write redundant code which can often be reused from the Web. Reusing programming tasks is beneficial since it speeds up the process of creating applications and reduces the errors related with the task creation from scratch. At the same time, the demands of our applications are increasing, leading to a simple problem having to be solved through several tasks. With the advent of the cloud, there are countless Web services that proliferate on the Web. One solution for developers is to use these Web Services. However, the process of mastering and coordinating all these services manually is time-consuming and error-prone.

This paper presents SOS, a Simple Orchestration of Services. The ultimate goal of this tool is to act as a service composer while promoting the separation of concerns for two typical actors in this realm: the developer and the business analyst. The developer must define a service as a SOS task based on a JSON schema and submit it in a Web specialized editor. The business analyst uses the SOS editor, in an interactive way, to chain the required tasks to solve a specific problem. Then, the developer, uses a a simple client API – a SOS engine wrapper – to load a SOS manifest and to iterate over all tasks, without the need to dominate any bureaucratic aspects related with HTTP clients and messages. As a case study, several tasks are instantiated and aggregated in order to generate a composite service for a mobile app whose goal is to give an translated description of a picture taken with a mobile phone.

1998 ACM Subject Classification H.3.5 [Online Information Services] Web-based Services

Keywords and phrases Web services, Service Composition, Orchestration

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.13

1 Introduction

Service Oriented Architecture (SOA) emerged in the early 2000s, offering a way to develop new business services by reusing components from existing programs rather than writing redundant code from scratch and developing new infrastructures to support them [3].

SOA can be defined as an approach to develop enterprise systems by loosely coupling interoperable services – small units of software that perform discrete tasks – from separate systems across different business domains [2].

A crucial aspect of SOA is service composition. Service composition is the process of create a composite service using a set of available Web services in order to satisfy a user request or a problem that cannot be satisfied by any individual Web service [2]. The service composition can be defined from a global perspective (choreography) or using a central component which coordinates the entire process (orchestration).

However, despite all the software that implements those concepts, there are few that can be used, in a very simple way, and focused on the new paradigm of cloud services, where the

JSON specification increasingly assumes a prominent role in the data exchange formalization within the client-server model.

In this paper we present SOS – Simple Orchestration of Services – as a tool for services composition. The tool is composed by two components: a Web editor and a client engine. The former allows the submission, aggregation, sharing and grading of small services defined as SOS tasks. The later, allows developers to iterate over all the service tasks, through a simple API.

The main advantage of this approach regarding the existent approaches is the simplicity and the separation of concerns. Firstly, a SOS task is formalized with a simple JSON schema. The aggregation is materialized on a JSON manifest that can be loaded from the cloud. Then, developers through a simple API can manage the execution flow of the process without worrying about HTTP clients and messages. Secondly, the SOS network fosters the separation of concerns by giving to the developer the mission to formalize and submit tasks and interact with the engine and to the business analysts the chance to compose the tasks for the creation of a single service. The chaining process will be very simple, using drag-and-drop techniques, and will automatically inform of invalid pairings based on the matching of the tasks' response/request types.

As a case study we present a composite service that receives a photo taken by a smartphone and returns a translation definition of the photographed object.

2 Services Composition

Service composition encourages the design and aggregation of services that can be reused in multiple scenarios. In this realm, two techniques are used: orchestration and choreography.

2.1 Orchestration vs. Choreography

Web service orchestration is a type of service composition where specific web service business processes are controlled by a central component. This component coordinates asynchronous interactions, flow control and business transaction management [3]. Typically, Business process modelling notation[1] (BPMN), maintained by the Object Management Group, is used to define a visual representation of the flow and business process execution language (BPEL) is used to write the code that executes the services.

Service choreography is a form of service composition in which the interaction protocol between several partner services is defined from a global perspective. This could be mapped for the dance domain, where "dancers dance following a global scenario without a single point of control" [4]. In other words, at run-time each participant in a service choreography executes its role according to the behaviour of the other participants [2]. Several languages specifications appeared to model service choreographies: the Web Service Choreography Description Language[2] (WS-CDL) and the Web Service Choreography Interface[3] (WSCI). Both are XML-based specifications from the W3C for modelling choreographies. The BPMN version 2.0 includes diagrams to model service choreographies. Other academic proposals for service choreography languages include: Let's Dance [5], BPEL4Chor [1] and Chor[4].

[1] Link: `http://www.bpmn.org/`.
[2] Link: `https://www.w3.org/TR/ws-cdl-10/`.
[3] Link: `https://www.w3.org/TR/wsci/`.
[4] Link: `http://www.chor-lang.org/`.

A distinction is often made between orchestration (a local view from the perspective of one participant) and choreography (coordination from a global multi-participant perspective, without a central controller). Although, the service orchestration is the most used and plays an important part in a service-oriented architecture (SOA), the web service choreography, is also often used to address the typical issues of single point-of-failure that are often found using the orchestration paradigm [3].

2.2 Related work

The SOS tool does not intend to compete with similar tools. If we want to catalog the proposed tool, it can be defined as an automation tool made up of small services formalized through light specifications such as JSON. In the field of service automation tools several tools appeared in recent years. The best examples are IFTTT[5], Zapier[6] and Microsoft Flow[7]. IFTTT is a free web-based service that people use to create chains of simple conditional statements, called applets. Zapier connects existent apps for the automation of workflows. Lastly, Microsoft Flow is an automated actions service. All of them are more focused on tasks automation and are not focused on the composition and execution flow management of HTTP REST services.

3 SOS – Simple Orchestration of Services

This paper presents SOS as a simple system to help developers and business analysts to submit services as tasks and chain them in order to create composite services. The architecture of SOS is straightforward and is composed by the following components:

- The editor – Web-based component with a GUI for the submission, chaining and generation of composite services.
- The engine – client component responsible for the orchestration of the composite service.

3.1 The editor

The editor is a Web-based component for the submission of tasks and their aggregation in order to obtain a composite service. The next section will detail the main aspects of the editor, more precisely, the GUI component, the task schema and the service manifest.

3.1.1 The GUI component

The SOS editor is a Web-based component, based on HTML5 Canvas and D3.js[8], that will help users in the submission and aggregation of tasks. The final result is a new composite service as a SOS manifest that can be stored in the cloud or saved in the user's computer. A user can perform the following operations in the editor:

- Submit a new task.
- Create a new composite service (by aggregating tasks).
- Share and grade tasks and services.

[5] Link: `https://ifttt.com/discover`.
[6] Link: `https://zapier.com/`.
[7] Link: `https://flow.microsoft.com/pt-pt/`.
[8] Link: `https://d3js.org/`.

After the registration and, before any submission, developers can search for desirable tasks using tags. For instance, if we want to use a service for weather consumption, we insert the tag "weather" and the editor shows all the tasks with the related tag. If no services are returned, the developer can submit a new task. The submission of a task can be performed interacting with the GUI component or by submitting a valid JSON document.

After the selection/submission of a task, business analysts can aggregate new ones in order to compose a service. The aggregation of two tasks (for instance, Task A and Task B) is only possible if the type of the response of Task A complies with one of the types of the request of Task B. Outside of this rule are the initial and the final tasks. In the end, we will have a composite service as a SOS service manifest that can be stored in the private cloud of the user or downloaded for its computer.

Other feature of the editor is the capacity for sharing and grading tasks. This feature will allow a user to share a previously created tasks in the public space of the SOS community. With the grade feature, developers could score a given task taken into account the experience that they have with it. This grading will influence the results list after searching.

3.1.2 The Task schema

Developers should create a task as a JSON document. In order to submit a new task, developers should comply with the SOS official task schema formalized by the following JSON Schema:

Listing 1 SOS schema for a task.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
  description: "A schema to formalize a SOS task",
 type: "object",
 properties: {
   id: { description: "task identifier", type: "integer" },
   tags: { type: "array", items: {type: "string"},
           minItems: 1, uniqueItems: true },
   placeholders: { description: "Dynamic values in the request",
                   type: "array", items: {type: "string"},
                   uniqueItems: true },
   endpoint: { description: "Task endpoint", type: "string" },
   request: { description: "The request (template with placeholders)",
              type: "object" },
   pathQueryResponse: { description: "Queries for the response",
                        type: "array", items: {type: "string"} } },
   required: ["id", "tags", "endpoint", "pathQueryResponse"]
}
```

Using schemata is always a good approach for validating client-submitted data, specially with the absence of a GUI interface. A JSON Schema has several purposes, one of which is JSON instance validation. The JSON schema is composed by six important elements that must be included in a SOS task instance:

- id – is a numeric value that uniquely identifies a task;
- tags – is an array with keywords that characterise the task. Their inclusion is crucial for the task discovery process;
- placeholders – specifies places in the request where substitution may take place;

- endpoint – the task URL endpoint;
- request – a request template sent to the task implementer. The template can include placeholders that will be changed automatically by the previous task response in the chain or, manually, through the client API;
- pathQueryResponse – a set of queries executed against the task implementer response in order to obtain specific results. These results can be used as input for the following tasks.

One final note for the description property. This property can serve multiple purposes, such as, to assist users with the editor GUI operations and to automatically generate the API code documentation.

As an example, the next code shows the JSON document for the Ipify service[9]. This service allows to get your public IP address programmatically.

Listing 2 Task instance for Ipify service.

```
{
    "id": 1,
    "tags": ["ip", "machine"],
    "placeholders": ["json"],
    "endpoint": "https://api.ipify.org?format=$1",
    "request": {},
    "pathQueryResponse": ["ip"]
}
```

The JSON document for the Ipify service is very simple. We included two strings for the tags property. The property placeholders is fulfilled with one value which will be used by default. However this default value will only be used if the developer does not associate another value through the client API. A placeholder is marked with a sequential number preceded by an $. In this example, there isn't any request message to send to the service implementer, thus the request property is empty. Finally, the pathQueryResponse defines one query to be performed in the task response: {"*ip*" : "98.207.254.136"}. The value of this property adheres to the public specification json-query[10]. After submitting a task as a JSON document, it is validated through a validator. For the validation process, the editor uses the jsonschema validator[11] for Node.js. The validator uses the previous JSON schema and produces a result. The following code validates a specific submitted task instance:

Listing 3 SOS schema for a task.

```
var Validator = require('jsonschema').Validator;
var v = new Validator();
v.addSchema(mainSchema, '/SOSTaskSchema');
var result = v.validate(myInstance, mainSchema);
if(result.valid === true) { // actions for successful validation }
else { // iterate result.errors }
```

In case of validation errors, they will be appended to the result.errors array which also contains the success flag result.valid.

[9] Link: `https://api.ipify.org?format=json`.
[10] Link: `https://github.com/mmckegg/json-query`
[11] Link: `https://github.com/tdegrunt/jsonschema`.

Table 1 API functions of the SOS engine.

Function	Description
`SOService SOService.createService(Url manifest)`	Creates a new service based on a SOS manifest
`SOService.start()`	Init the service
`Task SOService.getCurrenatTask()`	Gets the current task
`List<PlaceHolder> task.getPlaceholders()`	Obtains the placeholders defined for the current task
`PlaceHolder.setValue(int index, Object value)`	Assigns a value for a specific placeholder
`Object task.run()`	Run a specific task
`Object SOService.executeAll()`	Execute all tasks automatically and returns the response object of the last task

3.1.3 The SOS manifest

The aggregation of tasks should be done through the GUI component (using drag-and-drop). The aggregation of two tasks should fail if the tasks are not eligible for pairing. The eligibility is granted only if the two tasks (for instance, Task A and Task B) have response and request types similar. In other words, we cannot pair task A that returns a date with task B that are expecting an integer as input.

After the chaining process, the editor will produce a manifest (a new JSON document) with all the selected task instances included. The manifest structure is very simple and can be accessed by the SOS engine through an URL endpoint or with a local reference.

3.2 The engine

The engine is a software component that acts as the orchestrator in the SOS realm. The main goal of the engine is to manage the execution flow of the service and to free the developer of the need to handle HTTP clients and parsing request and response messages.

In practical terms, the engine receives a SOS manifest (from a local reference or a cloud location), maps the JSON data to a set of objects and iterates over them. At the same time, it provides a simple API so the developer can manipulate all the tasks in the composite service. In Table 1, we present some of the API functions of the SOS engine.

The engine support two run modes:

- **Automatic mode** – executes all the tasks automatically, picking the response of the current task and inject it in the request of the following task. In the end, it returns the response of the last task. In order to trigger this mode, the developer must use the executeAll method;
- **Iterative mode** – performs one task at a time. For each task, the developer can inject values to map with placeholders. Use this mode when the inputs of the app users are crucial for the flow of the service. In order to trigger this mode, the developer must use the run method of the specific task, and then, use the setValue method of the PlaceHolder object to inject new values from the user's input;

The engine uses the Fuel library[12] as the HTTP client. In short, with Fuel we can make HTTP GET requests to specific endpoints and use the responseString() method to

[12] Link: `https://github.com/openstack/fuel-library`.

asynchronously get the responses. At the time of writing, a Java binding implementation of the engine is being created and can be used, in Android platforms (for instance), as a Gradle dependency.

4 Case Study: a Photo App

In this section we briefly detail a Photo mobile app as a case study for the SOS tool. The main goal of the app is to allow the user to take a photo and, automatically, receive a description of the photographed object. In order to accomplish these requirements we need three services: (1) a service to analyse an image and get the name of the photographed object; (2) a service that translate the given name to another language (e.g. Portuguese); (3) a service that takes the translated name and returns its definition.

After some research, we found three services to address these requirements, namely: Google Cloud Vision[13], Google Cloud Translation[14] and Dicionario-Aberto[15] services. The first two are part of the Google Cloud Machine Learning services that provides fast, large scale and easy access to machine learning services. The last one consists of a project with more than 100,000 entries of the Portuguese dictionary providing a RESTful "user-friendly" API which returns results in XML and JSON formats.

The first step was the submission of each service as a task in the Editor. In this case, we need to create a JSON document that complies with the JSON schema previously described. For reasons of compactness and readability, we only expose the JSON instance for the first service, the Cloud Vision service.

Listing 4 Task instance for Google Cloud Vision API.

```
{
  "id": 1,
  "tags": ["vision", "image manipulation"],
  "placeholders": ["DEFAULT_KEY", "<BASE64-ENCODED IMAGE DATA>"],
  "endpoint": "https://vision.googleapis.com/images:annotate?key=$1",
  "request": {
      {"requests": [{
        "image": {"content": $2},
        "features": [{"type": "LABEL_DETECTION"}]
      }]}
  },
  "pathQueryResponse": "responses/labelAnnotations[0]/description"
}
```

The request property of this task deserves an explanation. In order to send data to the API, one should create an HTTP POST request. The body of the request must be a JSON document containing the Base64-encoded image data. For label detection, the document must also have a features array containing the value LABEL_DETECTION. When we use the label detection feature, the Vision API returns a JSON document containing labels. Along with each label, we also get a score specifying how accurate the label is. In this case, we pick the first label since it will be the most accurate.

[13] Link: `https://cloud.google.com/vision/`.
[14] Link: `https://cloud.google.com/translate/`.
[15] Link: `http://dicionario-aberto.net`.

After all the tasks submitted, the business analyst can select the tasks and create a composite service. The SOS Editor will generate a cloud endpoint for the manifest. Then, the developer, using the SOS library, can load the manifest through the engine API and iterate all over the tasks of the composite service. In this example, the developer must inject the Vision API key (previously obtained) and a Base64 encoded string based on the photo taken. The following code shows how to handle the engine API in order to inject user's value in tasks' placeholders and to manage the execution flow of the composite service.

Listing 5 SOS engine execution flow.

```
Process process = SOS.createProcess("http://sos.com/manifests/123)
ArrayList<PlaceHolder> placeHolders = new ArrayList<PlaceHolder>();
placeHolders.get(0).setValue(1, VisionApiKey);
placeHolders.get(1).setValue(2, base64Data);
process.getCurrentTask().run();
```

In short, the user takes a photo. Then, the generated Base64 encoded string is used as input for the Cloud Vision service. This service returns a label that represents the photographed object. Then, the label is translated using the Translation service. With the label translated, we use the last service to obtain a definition of the label. This label is presented in a Snackbar GUI component in the bottom of the device screen.

5 Conclusions

In this paper we present SOS as a tool for service composition. The main idea is to use an intelligent Web editor to aggregate small services as tasks and package them in a composite service. These composite services can be loaded in an engine library that will manage the execution flow of the service and abstract the developer of all the bureaucratic aspects related to HTTP messaging management.

The main contributions of this work is the JSON schema to describe a task and the a client API that will be exposed by a service engine.

As future work we intend to create a prototype by choosing a specific domain and by implementing all these components. In the editor we intend to maintain it simple (to justify its name), but the inclusion of conditional and cyclic blocks are being considered.

References

1 Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: extending BPEL for modeling choreographies. In *International Conference on Web Services*, 2007.

2 Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based analysis of obligations in web service choreography. In *Advanced Int' Conference on Telecommunications and Int' Conference on Internet and Web Applications and Services*, page 149, 2006.

3 Yang Hongli, Zhao Xiangpeng, Cai Chao, , and Qiu Zongyan. Exploring the connection of choreography and orchestration with exception handling and finalization/compensation. In John Derrick and Jüri Vain, editors, *Formal Techniques for Networked and Distributed Systems*, pages 81–96, 2007.

4 Ashley McNeile. Protocol contracts with application to choreographed multiparty collaborations. *Service Oriented Computing and Applications*, 4(2):109–136, 2010.

5 Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur ter Hofstede. Let's dance: A language for service behavior modeling. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems*, pages 145–162, 2006.