A Framework For Behavioural REST APIs

CO620 Research Project

Morgan Mahan

mrm45@kent.ac.uk

School Of Computing

University of Kent

United Kingdom

Abstract

Behavioural REST APIs often need particular behaviours to occur in a particular sequence or workflow, but this is often not documented, being enforced by developer knowledge alone. Constraining API interactions to desired behaviours ensures that the API can only be interacted with in a valid sequence. Currently, there is no API description format that includes behavioural information. Having a REST API specific workflow description format allows behavioural information to be included with other API documentation and allows the generation of tools that can govern API interactions based on the API description, such as a run-time monitor for API interactions. Interactions with an API can then be forcefully limited by an orchestrator service that sits in front of the API, restricting out of sequence calls. This ensures that API behaviours are limited to only those that are desired.

A framework consisting of three operations, prerequisite, postrequisite and exclusive, allows each API endpoint to specify its relation to other endpoints in a way that allows detailed specification of API behaviour and allows for a simple specification format. This has been integrated into Swagger's OpenAPI, an extension that is being called Behavioural-OpenAPI. To forcefully limit API interactions to those allowed by the specification, a Node.js module known as Swagger-Orchestrator has been created to be used as server middleware, rejecting API calls that are not part of a workflow specified in the Behavioural-OpenAPI specification. Both OpenAPI and Node.js are heavily used in industry and Swagger-Orchestrator has been tested with Express.js, the standard server framework for Node.js, ensuring that the framework could easily be applied to REST APIs in industry.

1. Introduction

1.1 Problem

Many REST APIs have sets of behaviour that require multiple API interactions from the client, and these behaviours require the interactions to happen in a certain order or workflow, such as adding an item to a basket, and then checking out. These orderings are often implicitly enforced and rarely documented anywhere, which causes uncertainty for developers that have not interacted with a particular API before, as well as potential for invalid interactions and behaviours that cause errors.

1.2 Approach

Creating a framework that can be used to specify ordering of API interactions, which can then be used to limit API behaviour, using current industry technologies. The framework consists of three parts: the operations that specify an API endpoints relationship to others, for example /endpoint1 must come before /endpoint2, the specification that implements these operations, and an orchestrator, which is some software that rejects API interactions that are invalid according to a specification. To ensure the practicality of the framework, a focus was put on simplicity and intuitiveness, making use of commonly used and easily understandable syntax, as well as scalability, ensuring that the framework can be used on APIs with any number of endpoints. In regards to the orchestrator, there should be no extra work required for this to be generated from a valid specification. To verify the framework's robustness, it was built to support and was tested against the workflows in Amazon Web Services' Implementing Workflow Patterns (n.d.).

1.3 Existing Work

There are existing systems that look to provide a way of modelling or specifying workflows, such as DCR Graphs (DCR Solutions, 2015), but these are not specific to REST APIs and cannot easily be included in existing API documentation, nor can they be used to generate any kind of software enforcing them. Similarly, existing API documentation formats, such as Swagger's OpenAPI (2011), do not include behavioural information. Even the GOV.UK guide for writing API reference documentation (2019) does not mention including any behavioural information in API documentation, behavioural information being that which specifies requirements and relationships between multiple API endpoints. This report looks to compare its framework with existing works in detail in section 6.

2. Operations

The framework consists of a set of operations that specify an endpoints relationship to other endpoints. Rather than specifying workflows as a separate entity, building workflows by specifying endpoint relationships through use of these operations allows the framework to be scalable and simple, which will benefit both use of the framework and conciseness of API documentation in which it is used. These operations are newly created for this framework, taking some inspiration from DCR Graphs (2015), and whilst alone they have no purpose further than understanding a workflow, as part of the overall framework they are integral to specifying the workflows in a specification as well as being understood by software that restricts APIs to these workflows.

2.1 Prerequisite

An endpoint can specify those endpoints that must be called before itself, known as prerequisites, either specifying that all those in its prerequisite set must be called before itself, or that at least one must be. This depends on the prerequisite sets “all rule”, which is a boolean property of the set. If the all rule is true, then all endpoints in the set must have been called before the current endpoint can be. If the all rule is false, then the current endpoint can be called once at least one endpoint in the prerequisite set has been called.

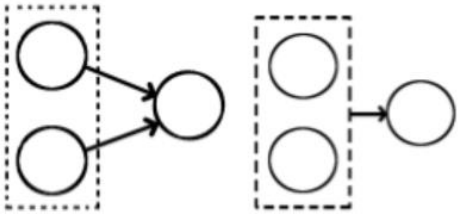


Figure Pre1
Prerequisite, all

Figure Pre2
Prerequisite, any

Prerequisites are useful for endpoints that resolve a concurrent sequence, specifying that the last endpoints in the respective sequences must be called before the resolving endpoint can be. For example, when booking a holiday online, reserving a hotel and a flight may be two separate API calls that execute concurrently after selecting a particular package. Payment cannot be taken until the hotel and flight are reserved and the client can be sure they are available and ready for booking, so /reserveHotel and /reserveFlight, would be prerequisites to /takePayment, and all prerequisites would be required. This small part of what would likely be a larger workflow would be represented as Figure Pre1, above.

2.2 Postrequisite

An endpoint can also limit the endpoints that can be called after itself, meaning no endpoints other than those in the current endpoints postrequisite set can be called after the current endpoint. Much like prerequisites, the all rule can be used with postrequisites. If it is true, all endpoints in a postrequisite set must be called, and if it

is false, at least one endpoint in the postrequisite set must be called before the workflow can continue.

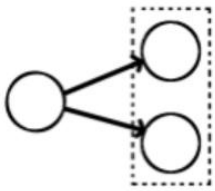


Figure Post1
Postrequisite, all

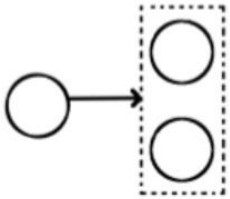


Figure Post2
Postrequisite, any

Postrequisites are the main relation that will be used to build workflows, being useful for building sequences, concurrent calls and when paired with the exclusive operation (2.3), specifying choices that can be used to choose between different subsequent workflow sequences. Using the previous example of booking a holiday online, after taking payment, both the hotel and flight that have been reserved need to be confirmed. /confirmHotel and /confirmFlight would both be postrequisites to /takePayment, and all postrequisites would be required. This part of the booking workflow would be represented as Figure Post1, above.

2.3 Exclusive

Figure Excl1
Exclusive

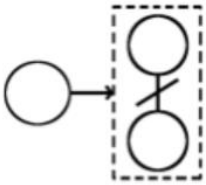


Figure Excl2
Postrequisite, exclusive any

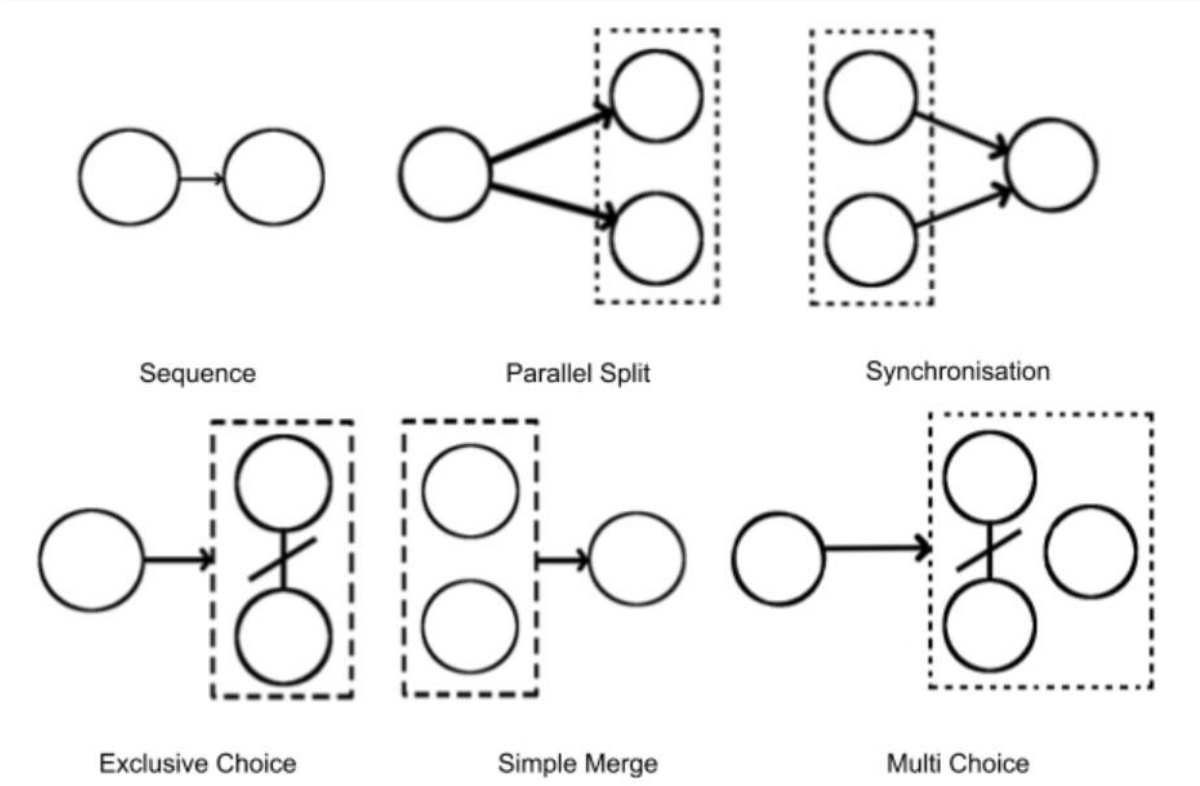
An endpoint can specify which endpoints it cannot be called in the same workflow as. If any endpoint in the current endpoints exclusive set has been called previously in the current workflow, then the current endpoint cannot be called.

Exclusives should mainly be used in conjunction with other operations, being particularly useful for building choices into a workflow, as mentioned in 2.2, Postrequisite. For example, when ordering a product online, a debit card could be pre-authorised to ensure there are sufficient funds available before reserving the product. After the pre-authorisation, the order can either be confirmed, if there is sufficient funds, or cancelled, if there is not, but not both. In the case of the order being confirmed, the workflow would likely continue into a further sequence. In the case of cancellation, this would likely be the end of the workflow. This part of the workflow would be represented as Figure Excl2, above.

2.4 Building the operations

When initially creating the set of operations, I took a look at a number of individual workflow patterns to understand the different ways that an overall workflow could be constructed, such as those in Workflow Patterns (van der Aalst, ter Hofstede, Kiepuszewski and Barros, 2003) and Implementing Workflow Patterns (Amazon Web Services, n.d.). The workflow patterns that the operations were developed to support

are all those in Implementing Workflow Patterns (n.d.), which is a subset of those in Workflow Patterns (2003). These workflows are the following:



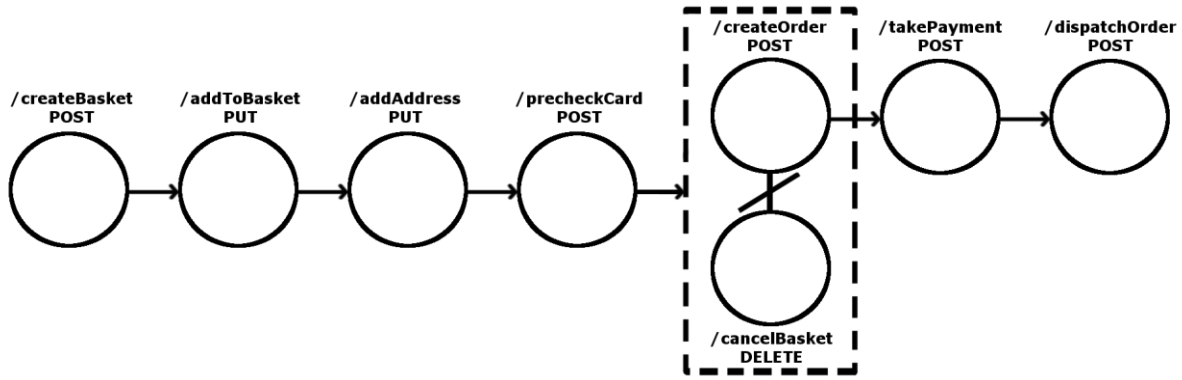
These are appropriate as they are small workflow patterns that can easily be chained together to create larger full workflows, being able to create the workflows that were not initially considered from van der Aalst's Workflow Patterns (2003), as well as being workflow patterns that can easily be specified by a single endpoint, as each only contains two 'steps', that being either two individual endpoints or an individual endpoint and a set of endpoints.

Operations being specified by a single endpoint is an important factor for scalability and conciseness, as this ensures that, in API documentation, operations can be documented with other endpoint information, rather than separately as, say, an entire workflow. Building workflows by specifying endpoint relationships rather than specifying entire workflows as their own entity is a far more scalable approach. This approach was inspired, along with the reasons outlined above, by the approach taken in DCR Graphs (2015), in which each activity has 'connections' to other activities. Each type of connection being a different type of relationship to other activities makes the workflow creation process intuitive and allows the workflow to grow to any size, since it is a scalable approach.

3. Building Workflows

The operations outlined in section 2 are used together, through chaining endpoints with the operations, to build workflows. Simple sequences can be built using just postrequisites but building more complex workflows requires use of all three operations.

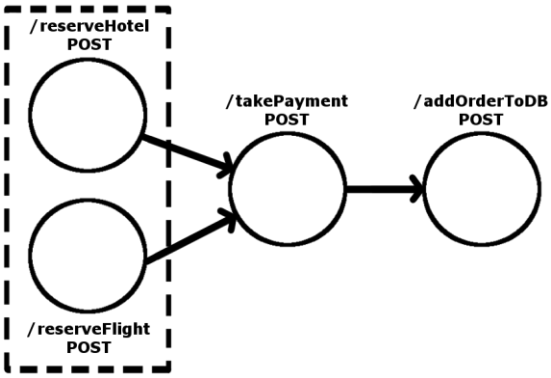
3.1 Online order workflow



The above workflow is what an entire online product order workflow could potentially look like, represented using the frameworks operations. This workflow is implemented in the Online Order Server API that is included in this project's corpus. The workflow has two separate sequences, created by using postrequisites, from /createBasket to /precheckCard and /createOrder to /dispatchOrder, which is what the large majority of workflows will be made up of. The sequences are simple to create, made up of single postrequisites requiring either all or any, it doesn't matter when there is only one postrequisite endpoint.

The exclusive choice in the middle is not as trivial as the sequences on either side of it. This is created using both a postrequisite and an exclusive. /precheckCard has two postrequisites, /createOrder and /cancelBasket, requiring any of them to be called, meaning either one or both of them need to be called for the operation to be satisfied. This is then made a choice by the exclusive. /cancelBasket is exclusive to /createOrder and vice versa, meaning that now only one of the postrequisites is allowed to be called, as once one has been called, the other cannot be. If /cancelBasket is called, the workflow completes and the only valid call is then one that starts a workflow, for example /createBasket or the initial state of another workflow. If /createOrder is called then the second sequence begins and the workflow continues. In this case, the workflow then completes on /dispatchOrder, and the next valid call must be the initial state of a workflow, such as /createBasket.

3.2 Package holiday workflow



The previous online order workflow was rather large, looked at from the perspective of the entire booking process that may be part of a monolith API. This workflow takes the perspective of microservices, where responsibilities are spread between multiple APIs and as such, workflows are much smaller, which is often preferable.

The above workflow is a package holiday booking workflow. This workflow would be executed once the customer has confirmed their choice of holiday. Both /reserveHotel and /reserveFlight are initial states in this workflow, and they must both be called before the workflow can continue. These two endpoints are parallel, not necessarily meaning they are executed at the same time but that they can be executed in any order as long as they are executed together. These endpoints are both prerequisites, all of which are required, to /takePayment. /takePayment then has one postrequisite, /addOrderToDB, which can be any or all required since the postrequisite set consists of a single endpoint. /addOrderToDB is the final endpoint in the workflow, completing the workflow once it has been called. Once the workflow has completed, the only valid next call is a workflow's initial state.

4. Behavioural-OpenAPI Specification

The frameworks operations are used to define the workflows in an API. These workflows can be specified in an extension of Swagger’s OpenAPI (2011) that was created for this project called Behavioural-OpenAPI.

4.1 Why OpenAPI?

OpenAPI is an industry standard format for creating API reference documentation. After reading an OpenAPI specification, a developer should have a complete understanding of how to interact with an API, but unfortunately, I find this not to be the case currently due to the lack of relational information between endpoints. When looking at endpoints individually this is the case, but in reality, some API behaviours require multiple endpoint interactions to be made in a certain order. OpenAPI currently has no way of describing these orderings.

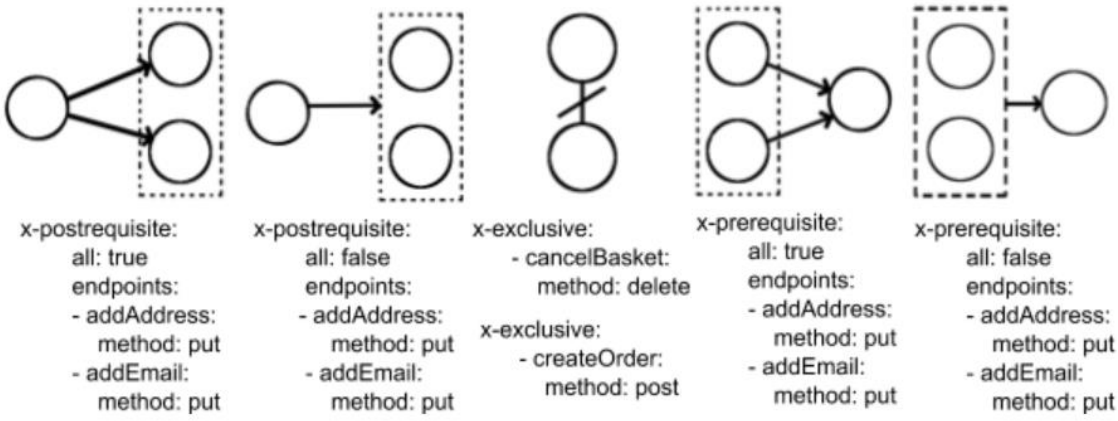
OpenAPI is an extensible description format (Swagger, n.d.), meaning that new properties can be added to the description that aren’t part of the standard OpenAPI

supported properties. This allows the Behavioural-OpenAPI extension to be added without needing to make a pull request that changes anything in Swagger's OpenAPI.

Being the industry standard API description format and being extensible, it makes sense to extend OpenAPI for the specification of this framework, since existing APIs can use the framework without having to use a new tool, and new APIs can use the framework without being forced to make a choice between the industry standard of OpenAPI and a specific tool for this framework, or using both, which would cause their API documentation to lack cohesion. Extending OpenAPI to include this framework ensures that all documentation around API interaction is together. Having separate API documentation would likely cause confusion and less use of the framework.

4.2 Syntax

In Behavioural-OpenAPI, each of the operations has its own representation, each of which follows the standard for the framework to make it simple and intuitive to create a Behavioural-OpenAPI specification. The following graphic shows how each operation is implemented in Behavioural-OpenAPI.



As well as the above properties for operations, there are an additional two properties required to specify a full workflow in Behavioural-OpenAPI.

x-initialState: true
This property indicates that a particular endpoint is the first in a workflow. When not currently in a workflow, i.e. if it is a client's first call to the API or a workflow has just been completed, only endpoints with the x-initialState property can be called. Once an initial state has been called, then a workflow can begin.

x-returnToInitialState: true

This property indicates that a particular endpoint is the end of a workflow. If an endpoint with this property is called, it completes the current workflow. After a x-returnToInitialState endpoint has been called, only an initial state can be called to start another workflow.

5. Orchestrator

An orchestrator is a program that restricts interactions with an API to only those allowed by a given specification, in this framework specifically a Behavioural-OpenAPI specification. An orchestrator monitors all incoming API calls and ensures that they are valid calls that do not violate the workflows in the specification. If a call is invalid, it will be rejected with an error message being sent back to the caller, explaining why the call was rejected.

An API monitored by an orchestrator is similar to a monitored network as outlined in Monitoring networks through multiparty session types (Bocchi, L. et al, 2017), although in this case only one orchestrator is present for an API. Much like a monitor in a monitored network, the orchestrator evaluates incoming calls, ensuring that they do not violate the current workflow as set out in the Behavioural-OpenAPI specification.

4.1 Why an orchestrator?

For this project I created a JavaScript orchestrator to prove the concept, but different implementations of an orchestrator could be created. An orchestrator has many benefits, from making an API less susceptible to attacks, providing greater understanding of API interactions, making the development experience of the API much easier, and protecting the API from crashes through bugs may have occurred if the orchestrator was not present, for example if a prerequisite had not been called and defensive programming against this had not been implemented in the requiring endpoint. In fact, an orchestrator negates the need for a lot of defensive programming, since often this is to protect against incorrect or missing data that would have been provided by a prerequisite endpoint. Defensive programming should never be neglected, but an orchestrator could be even more beneficial to error prone APIs that do not follow a defensive design.

4.2 Sessions

One orchestrator should be present per session. Whilst it is unlikely that this will be how the orchestrator is implemented, this is how it should seem to be implemented. There should be no crossover between sessions communicating with an API, which could create undesired behaviour, allowing workflows to potentially cross over and

allow invalid calls to be made. It is important that an orchestrator can acknowledge multiple sessions for use in a production API, since there is rarely going to be just one session present at any one time. Using the online product order example used previously, the ability to only handle one session would make an orchestrator completely unusable to a company implementing that workflow, since it is highly likely that more than one order will be taking place at any one time. Handling of multiple sessions is paramount for an orchestrator to be usable in a production scenario.

4.3 Security Benefits

As the orchestrator receives incoming API calls before the API itself does, the orchestrator is able to evaluate incoming API calls before they reach an endpoint, allowing the orchestrator to reject calls before ever being processed by the API. This has an added security benefit.

Whilst not specifically related to APIs, rather Internet Of Things devices, the Department For Digital, Culture, Media and Sports Secure By Design report (2019) makes a strong point that the growing number of connected devices increases the attack surface through which attackers can compromise IoT systems. This point also applies to APIs. The higher the number of endpoints available, the greater the attack surface through which an attacker may be able to compromise the API. By implementing an orchestrator that receives and evaluates API interactions before they can be processed by the API, rejecting any interactions that are invalid according to the specification, the attack surface is greatly reduced. Without an orchestrator, attackers have the freedom of interacting with any endpoint present on the API at any time.

4.4 Incorrect behaviour

Reducing the number of available endpoints at any one time to only those that are valid according to the specification reduces the number of incorrect and out of sequence calls that can possibly be made to zero, providing the specification is correct. This greatly reduces the overall number of incorrect and potentially fatal behaviours that can occur within the API.

5. Swagger-orchestrator

Swagger-orchestrator is the orchestrator created specifically for this project to prove the concept of an orchestrator.

5.1 Technologies

The orchestrator developed for this project, known as swagger-orchestrator, is written in Node.js, a JavaScript framework that is the most popular backend web framework according to the most recent Stack Overflow developer survey (2019). This makes the orchestrator very easy to use in an API, requiring three lines of code to be set up, and gives it the ability to be used in a lot of APIs, since it is using a popular technology. The orchestrator was tested using Express.js, the most common web server framework used with Node.js, but could be used with alternatives, and is used as Express middleware.

5.2 Limitations

This orchestrator does have some limitations with how it has been implemented, specifically surrounding sessions and completeness of framework operations. Currently, swagger-orchestrator can implement all operations but does have some limitations with continuing from any required postrequisites, namely with continuing with the postrequisites of those in the set. Currently, any required postrequisite sets only work when there are two endpoints in the set and they are exclusive. Postrequisite endpoints where all are required work correctly as long as they instantly resolve.

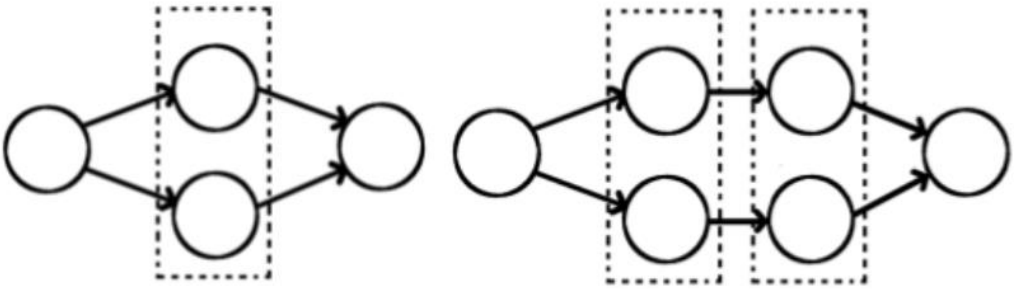


Figure A
A valid concurrent workflow

Figure B
An invalid concurrent workflow

Swagger-orchestrator does have issues with concurrent sequences that do not instantly resolve. For example, if an endpoint had two postrequisites and both of them were required, and those postrequisites both had their own respective postrequites, and then those postrequisites both resolved back into a single endpoint, by the single endpoint having the two postrequisites as prerequisites. This would not be supported, as the concurrent sequence goes on for more than one set of endpoints. This workflow is shown in Figure B, above. A valid concurrent workflow, one that instantly resolves, is shown in Figure A.

Regarding sessions, currently only one session can be active on the server at a time. Multiple sessions are necessary functionality for a production API but unnecessary to

prove the concept of an orchestrator and due to time constraints, this feature was left out of swagger-orchestrator.

5.3 Internals

Swagger-orchestrator checks requirements of workflows sequentially, ensuring that all are satisfied. This means that if multiple rules are broken, i.e. the call is not in the current postrequisites and one of its exclusives has already been called, the orchestrator does not need to check them all, as the call will reject on the first broken rule. This saves minimal time when there is low traffic to the API, but with a lot of traffic, say even a Denial of Service attack, this will be beneficial. It ensures that invalid requests are dealt with and do not leave the API busy.

The first thing that the orchestrator checks is if a workflow is in progress, by checking its own state. In the case of there being no workflow started, the orchestrator will reject any calls that do not have the property x-initialState in their Behavioural-OpenAPI specification. This ensures that any API interactions that are made are part of a workflow. Without this feature, providing a workflow had not been started, any endpoint could be called at any time, which would open up the opportunity for misuse such as entering a workflow halfway through its flow, if the rules would allow it, for example if an endpoint had no prerequisite requirements, only postrequisites. Ensuring that the first endpoint in a workflow is specified and calls are restricted against that when a workflow has not been started ensures that workflows always follow their desired sequence.

For prerequisites, the orchestrator stores the calls that have previously been made. When an endpoint is called that has prerequisite requirements, the orchestrator checks the previous number of calls equal to the number of endpoints in its prerequisite set. For example if there were two prerequisite endpoints, the orchestrator would check the two previous calls that had been made. If all prerequisites are required, then both of the two previous calls would need to be the same as the two prerequisites. If any are required, at least one of the previous calls would need to be the same as either of the prerequisites.

If an endpoint is called and it has postrequisite requirements, the orchestrator will store those postrequisites in a variable called nextCalls and await the next interaction. The next call that is made will then be checked against these nextCalls. If an endpoint is not specified in nextCalls, it will be rejected. If it is, that endpoint will be pass through the orchestrator and then be removed from nextCalls. If nextCalls specifies that all are required then no endpoints other than those in nextCalls will be

allowed to be called until nextCalls is empty, meaning all of its endpoints have been called.

Exclusive requirements take advantage of the stored previous calls used by prerequisites. If an endpoint is called and it has exclusive requirements, the orchestrator will check that none of the endpoints in the previous calls match any of the endpoints in the exclusive requirements. If there is a match on any of them, then the current call is rejected.

Once a workflow is finished, signalled by the x-returnToInitialState property being present in an endpoints Behavioural-OpenAPI specification, the orchestrator will clear its state, meaning that the previous calls state will be emptied, as well as the nextCalls state being emptied. This ensures that different workflows do not conflict with each other. They should be singular, not crossing over with each other.

6. Comparison with other frameworks

6.1 DCR Graphs

There are a number of differences between this framework and DCR Graphs, although DCR Graphs did inspire the operations for this framework. The first and most obvious difference is that DCR Graphs are not specific to REST APIs, rather business processes. Secondly, DCR Graphs cannot be exported to a simple programmatically accessible format, such as JSON. This is required for the specification to be used in the orchestrator, the specification needs to be programmatically accessible to be used in the restriction of API interactions.

In terms of the operations themselves, there are multiple differences. Firstly, this framework uses sets to show which endpoints come after others. DCR Graphs instead simply splits the process to two or more separate processes using the spawn connection. Using sets shows more clearly the relationship between endpoints, whether that be the requiring endpoint or those in the set, and makes it easier for an orchestrator to handle concurrent operations.

DCR Graphs also have additional ‘connections’, the equivalent of operations, for conditions, which this framework does not have. In this framework, the orderings are all that matter. Particular choices that are made should be decided in the code of the API that the orchestrator is implemented in. Choices are simply implemented in the framework by ‘any’ postrequisites and exclusives instead.

DCR Graphs has an equivalent to exclusives, called exclude, which excludes an activity upon completion of another. It also has an include connection, of which there

is no equivalent in this framework. This connection includes an activity upon completion of another. I did not feel this was necessary as by creating sequences, new activities, in the case of a REST API it would be endpoints, are included simply by interacting with an endpoint before it in the workflow.

DCR Graphs also includes responses. This framework disregards responses, since how an endpoint responds to an interaction makes no difference to the workflow, except perhaps in the case of choices. In the case of a choice, as mentioned previously, these are implemented using ‘any’ postrequisites and exclusives. This framework does not use knowledge of outcomes within the API to determine what is and isn’t a valid workflow. Rather, it lists what could come next and allows the API to programmatically decide between the options given to it.

Many of the connections in DCR Graphs would not make sense in a REST specific workflow notation, mainly due to the fact that each endpoint needs to be invoked by a caller, whereas in business processes it is more common for activities to be invoked by other activities completion, which would be more similar to the behaviour of a function calling multiple other functions in an API.

6.2 Business Process Model and Notation

Business Process Model and Notation (BPMN, n.d.) is a format for creating flow charts for business processes. Obviously the first difference here is that this format is not specific for REST APIs, but it also would not be appropriate due to a number of differences with this framework. A lot of the differences between BPMN and this framework are quite similar to the differences between DCR Graphs and this framework, which likely comes from the fact that both of their desired purposes are for modelling business processes.

Firstly, as with DCR Graphs, activities automatically trigger other activities, whereas REST APIs need each activity, in an APIs case each endpoint, to be invoked individually by the caller. As such, the foundations of how to construct a BPMN chart is flawed in the context of REST APIs. As mentioned in section 6.1, it is much more similar to how a function would call multiple other functions rather than how an external entity interacts with an API.

BPMN is a very standardised way of creating business process models, even having multiple versions released, the latest being version 2.0 (Object Management Group, 2011). These standards are purely visual however, there is no standard way of creating a BPMN model in a text format, nor being able to export a BPMN graph to a text format. That makes it completely unfit for the purposes needed in this paper, as

an orchestrator would have no way of understanding a BPMN model. Whilst it is plausible that REST API workflows could be specified with BPMN, all be it not very well, BPMN could not be used with an orchestrator due to its lack of text representation.

6.3 OpenAPI

Unlike the works mentioned in section 6.1 and 6.2, OpenAPI is both not a process modelling tool and is actually used as part of this project. The former being the first issue and why OpenAPI had to be extended as part of this project, because there is no way of modelling processes. This could be seen as an advantage from the perspective of this project however, as it allowed the integration of a completely new framework for specifying API workflows, without overlapping in purpose with some existing, previously integrated process modelling tool.

OpenAPI does provide what BPMN and DCR Graphs do not however, a REST API specific documentation format, which gave this project a basis on which to build upon, adding behavioural information to an existing format that has the same purpose, REST APIs, rather than creating a separate documentation tool specifically for this framework, meaning a reference documentation format would need to be used as well as this frameworks tool. It is a much better approach for the two to be integrated with each other, since they both cover the same domain, how to interact with a REST API.

OpenAPI was the perfect tool to build upon for this project. Making a new specification format did not make sense, it would cause overall API documentation to lack cohesion and conciseness, forcing those who wanted to use this framework to use two separate formats. The lack of existing behavioural information in OpenAPI allowed this project to have a clean slate implementing an extension, allowing there to be two clear individual purposes for the Behavioural-OpenAPI extension and OpenAPI as it currently exists.

7. Future Work

7.1 Using the framework in a production environment

There is extra work that could be done to extend the work undertaken as part of this project directly, mostly in further development of swagger-orchestrator. Whilst this was created to prove the concept, it could be used in a production API with a few improvements.

Firstly, the issues described in section 5.2 should be addressed. Any postrequisites are not fully implemented. I thought that this could be implemented using a second set of ‘nextCalls’, which are those calls that will come after the current set of nextCalls. In the case of any postrequisite, this would mean that if an endpoint is called, first nextCalls would be checked as normal. If the endpoint was not presented, the second set of nextCalls could be called. If it was present here, then the set of postrequisites that it belongs to could become the current nextCalls, and the second set of nextCalls could be emptied. This allows as many current postrequisites to be called as the client likes without losing the postrequisite sets of the current postrequisite endpoints.

The second issue described in section 5.2 is the limited level of concurrency that swagger-orchestrator provides. Currently, only concurrent sequences that instantly resolve are supported. An example of what I mean by an instantly resolving concurrent sequence, /endpoint1 has two postrequisites, all required, /endpoint2 and /endpoint3. Then there is /endpoint4 which has two prerequisites, all required, which are /endpoint2 and /endpoint3. Concurrent sequences should be able to be much longer than a single endpoint per thread in practice. This is certainly some necessary functionality that would be needed if a swagger-orchestrator was to be used in a production API.

The third issue from section 5.2 is handling multiple sessions. This is another improvement required for using swagger-orchestrator in a production API. Currently, different sessions would conflict with each other, likely causing confusion for callers and disallowing calls that should be valid. The orchestrator should be adapted to be able to handle multiple sessions, keeping their particular workflow progressions, their state, separate from each other. From an outside viewpoint, it should seem as though an orchestrator is assigned to each session. This is a possible implementation, but I imagine it would be simpler and easier to have one orchestrator for the whole API that handles every session, storing each session's state separately from each other, and only allowing a particular session to access its own state.

Due to how the orchestrator has been developed, ensuring each check within it are completely separate, ensuring the responsibility of each line of code is clear, the above continuations of the work should be trivial to implement.

7.2 Extending this framework

To extend this framework further, there are a few improvements that could be made. Firstly, there is currently no way to visually represent workflows without creating them in some graph creation software. It would be very useful if the automatic generation

of a physical representation of workflows could be integrated with Behavioural-OpenAPI, such that when a Behavioural-OpenAPI specification has been created, the interactable documentation that OpenAPI creates includes visual representations of all of the workflows in the specification. This would allow developers to understand the workflows in an API much more easily, as they would be simply represented alongside other API reference documentation.

Another improvement to this framework would be to simplify the case in which a set requires another set. In the example where there is an any postrequisite set and that set has an all postrequisite set itself, the only way to specify this currently would be to have every endpoint in the any postrequisite set to have all of the endpoints in the all postrequisite set as all postrequisites. It would be far easier and simpler to have the option for a set to specify its postrequisites, rather than every endpoint in that set needing to. This would also need an accompanying visual representation, as currently a set pointing to another set would mean an ‘any’ postrequisite.

7.3 Client-Side Orchestrator

A new piece of software that could be created to extend this framework would be an orchestrator for the client-side. This would be the same as the orchestrator specified in this paper, which is on the server-side, but would ensure that a client understands how to call a server, restricting outgoing calls to only those that are valid according to the specification. This would ensure that the client is interacting correctly with the server, which would lead to the server-side orchestrator needing to reject far less calls. If there was an orchestrator for both the client and the server, the reasoning behind implementing an orchestrator, ensuring that APIs are called correctly according to their specifications, would be fulfilled even more.

8. Conclusion

The idea of specifying the specific workflows that an API uses to perform certain behaviours and restricting an API to those workflows seems to have a lot of promise. It means API behaviours must be more clearly documented, making reference documentation more detailed. It prevents invalid API calls from even reaching an API and greatly decreases the attack surface for potential hacking attempts against an API, as well as decreasing the likelihood of bugs occuring in an API, due to the correct ordering of interactions being forced. Whilst the orchestrator created for this project did have some missing features due to time constraints, I believe an orchestrator is an effective addition to an API that requires its endpoints to be called in particular orders.

Unfortunately, there does not seem to be much other work that looks into restricting API interactions to workflows, but I believe it is an idea that could be very beneficial to a lot of production APIs, being most applicable to APIs that facilitate purchases, such as holiday bookings or product purchases. These APIs need to be secure and often have very specific workflows they need to follow, two of the biggest benefits of using an orchestrator.

To anyone else approaching a similar piece of work, I would say the most important thing is to bear in mind the realistic scenarios that would implement your work. This project was largely based on my past experience developing APIs in a commercial setting, which allowed me to see the needs and benefits that real companies could get from software like an orchestrator.

Acknowledgements

I would like to acknowledge my supervisor Laura Bocchi, who was integral to the completion of this project.

Bibliography

Amazon Web Services. n.d. *Implementing Workflow Patterns*. [online] Available at: <https://docs.aws.amazon.com/amazonswf/latest/awsrbflowguide/programming-workflow-patterns.html>.

Bocchi, L., Chen, T., Demangeon, R., Honda, K. and Yoshida, N., 2017. *Monitoring Networks Through Multiparty Session Types*. [online] Available at: <https://www.sciencedirect.com/science/article/pii/S0304397517301263?via%3Dihub>.

BPMN, n.d. *Business Process Model and Notation*. [online] Available at: < http://www.bpmn.org/>

DCR Solutions. 2015. *DCR Graphs*. [online] Available at: <https://wiki.dcrgraphs.net/dcr-graphs-the-knowledge-workflow-gpsx/>.

Department For Digital, Culture, Media and Sport, 2019. *Secure By Design: Improving The Cyber Security Of Consumer Internet Of Things Report*. [online] Available at: <https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/775559/Secure_by_Design_Report_.pdf>.

GOV.UK. 2019. *Writing API Reference Documentation*. [online] Available at: <https://www.gov.uk/guidance/writing-api-reference-documentation>.

Object Management Group. 2011. *Business Process Model And Notation Version 2.0*. [online] Available at: <https://www.omg.org/spec/BPMN/2.0/PDF>

Stack Overflow. 2019. *Developer Survey 2019*. [online] Available at: <https://insights.stackoverflow.com/survey/2019#technology>.

Swagger. 2011. *What Is Openapi?*. [online] Available at: <https://swagger.io/docs/specification/about/>.

Swagger. n.d. *Openapi Extensions*. [online] Available at: <https://swagger.io/docs/specification/openapi-extensions/>.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. and Barros, A., 2003. *Workflow Patterns*. [online] Available at: <http://eprints.qut.edu.au/9950/1/9950.pdf>.