Platform-agnostic Cloud-based Orchestration

Introduction

One could view all software systems as a collection of a rather large set of functions[1].

What if you could leverage value from this collection of functions to build your next software product? This paper will discuss how a lightweight platform-agnostic cloud-based orchestration tool can be applied to encourage reusability while providing some additional value that may surprise you.

First, a little history for awareness of advances that got us to the here and now...
When object-oriented programming (OOP) was born it was thought that the design phase of a software system can be disseminated into a specific set of required functionality by looking at a given system from the top-down and then from the bottom-up. It's in this bottom-up view where one gains an awareness of how a given system is comprised of a specific set of functions. Viewing a new system from the top-down can reveal the hierarchical nature of how functions depend on each other along with insight on the desired order of their invocation. It was also thought that top-down and bottom-up awareness provides one with the appropriate frame of mind to build a system that results in a set a reusable functions by reducing or eliminating the tightly coupled nature of a given function with it's consumer.

The idea of making use of reusable functions in theory sounds great but, in practice one might ask how does one actually do this when the frameworks, libraries, specifications and platforms all seem to evolve and shift in a way that inhibit reuse of proven functionality going forward? Another way to refine this question is to ask how does one make use of a layer of abstraction that will isolate the application layer from it's framework to increase the likelihood of constructing a new system from reusable functions.

Theoretically, at least, reusability seems achievable as it's just a matter of developing functions that perform a distinct operation that can be loosely coupled from their consuming mechanism and framework. Many of the design patterns that were introduced in 1994 out of the work by the Gang of Four[2] have been designed to provide a protective layer of abstraction between each of the application layer functions and the surrounding environment (i.e. platform, framework and utilized libraries) to strengthen reusability and robustness. These patterns include the Facade pattern, Adapter Design pattern, Inversion of Control pattern and the Bridge Design Pattern among others.

A resulting issue still remains however, where there is this set of code that defines the order of events to be performed. The consumers of each of the reusable functions that drive their invocation. Imagine that an application can be set up in such a way that the order of events that are invoked is completely data driven (i.e. not hard-coded).

---

1 For the sake of this discussion we'll refer to *functions* to also include *member functions* of a given OOP *class.* Within the context of this document it is thought that the difference with the utilization of member functions is just an implementation detail in regards to how objects are instantiated and invoked. OOP techniques by their nature are well suited for reuse through the adherence of the 'Composition over Inheritance' principle where classes are designed to achieve polymorphic behavior and code reuse by their composition (i.e. by containing insances of other classes to achieve the desired functionality).

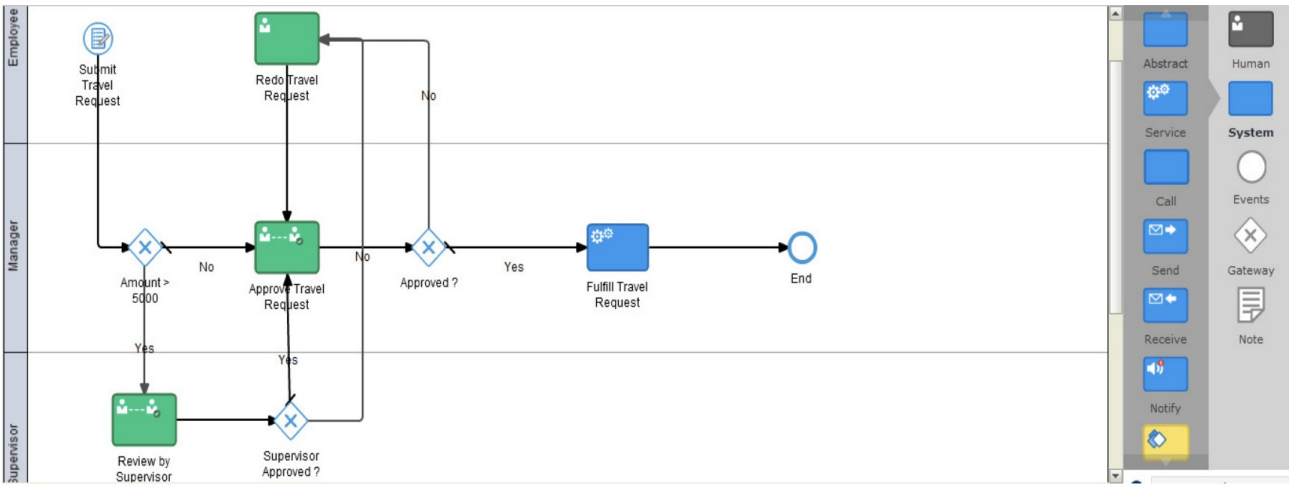
2 The *Gang of Four,* Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, are the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". This important book describes various development techniques and pitfalls in addition to providing twenty-three object-oriented programming design patterns.

It turns out that there are orchestration frameworks that predominantly reside in the Business Process Management (i.e. BPM) space that are designed to invoke a set of business processes from a data driven orchestration engine. These orchestration frameworks include a process flow design tool that can be used to define the order of events to be invoked.

A second piece of software, the orchestration engine, will then make use of the data generated from the orchestration design tool to actually perform each function in the order represented.

Each shape as depicted in figure 1, for example, represents a given process that are connected with one way arrows from a given starting point to a given ending point in response to a triggered event (e.g. travel reimbursement request application).

Figure 1: Oracle's Cloud-based orchestration tool, Process Cloud Service



The classic BPM example chosen in figure 1 depicts a flow of events at a relatively high business process level instead of orchestrating each application layer function that a given component is comprised of to achieve the corresponding process. In figure 2, you'll find an example of what an application layer work flow might look like that invokes reusable application layer functions.

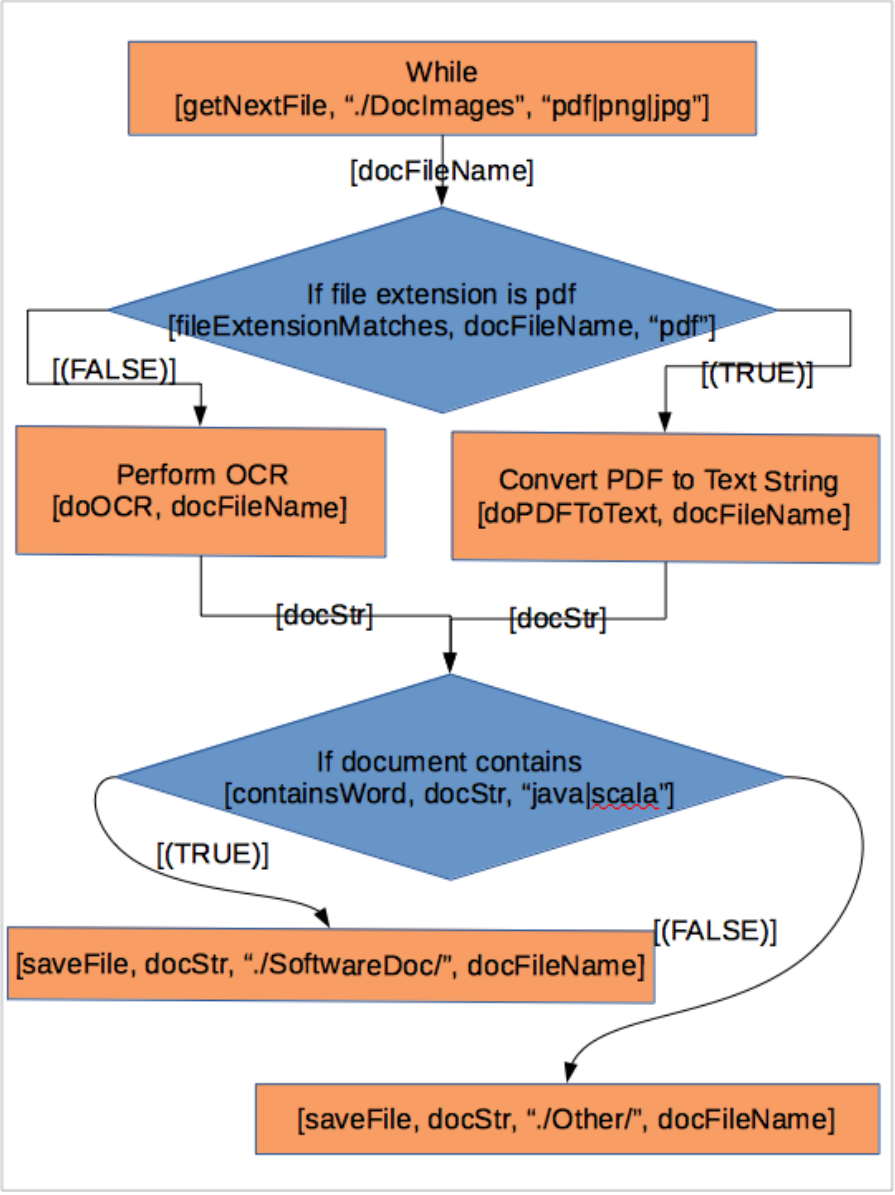


Figure 2: Function level orchestration

What would it look like if an orchestration tool was utilized to drive microservices?

As you can see, it does not take much imagination to see how a given set of services could be orchestrated on a distributed network with an orchestration platform. The messages being passed between each microservice and their corresponding invocations can both be illustrated and defined with the work flow design tool while the orchestration engine can reside on the distributed network to invoke the corresponding microservice. The same advantages generated being that each microservice is decoupled from its' consumer. The only difference with a microservice platform over it's monolithic counterpart is that the microservice implementation includes additional specifications as to which functions can be performed in parallel and the focus of a good design emphasizes the messages being passed between each of the microservices to contribute to how the orchestration engine distributes the work flow across a set of nodes within a network cluster.

Before embarking on the endeavor of building a microservices platform it is assumed that one has answered fundamental questions as to why this might be a good idea. The reasoning might include, for example, a need to conform to the tenants of a reactive manifesto. That is, there's a need to increase responsiveness, provide scaleability in response to expected growth, a need for resilience from failure, a need for elasticity in response to peak demands over a given day/month/season or a required emphasis on a message driven persistence model where current state can be captured for a given moment in time. The approach might also be motivated with an interest in optimizing organizational efficiency where there's a desire to efficiently use autonomous groups of software developers responsible for a corresponding discrete set of services.

There is an organizational advantage of utilizing a Domain Driven Design, DDD, as its' referred where a given set of distinctly related microservices are setup with a layer of persistence to minimize interaction and dependence with other domains. One domain, for example, might be focused on providing a catalog of products while another domain is responsible for reporting on and maintaining the current inventory of products at a given location.

There are several approaches that one can take in determining where the lines are drawn that segregates a given domain from others within a microservices platform. There is a school of thought, for example, that proposes that there is an advantage with first defining the messages being passed in response to a given event to distinguish where domains boundaries are drawn when migrating from an existing monolithic application. This approach places emphasis on the messages as opposed to the commands that are being carried out with a given operation. The idea being that a command is a static function that carries with it the nature of a state. The state of 'will be done', 'being done' or 'has been done'. The messages being passed between each of these microservices on the other hand distinguishes how the data can be persisted into a given entity. There is also a school of thought that relegates that domain boundaries are drawn at context boundaries that might be distinguished by a set of data entities. A context boundary might also be drawn to a sphere of responsibility as it relates to organizational lines as to how they relate with a given set of operations. There is also the converse of this last approach where domain boundaries are drawn to accentuate the efficiency of a given operational flow of events that may intentionally be designed to circumvent organizational lines.

DDD in itself does not fix the perception of spaghetti code within a given domain without the use of an orchestration platform. The spaghetti view being the interwoven dependencies that dictate both the messages being passed between each microservice and the predefined order of events performed in response to a given event. Placing the encompassing set of microservices into a distinct set of domains will only postpone the inevitable spaghetti perception as it can be thought of as simply dishing out a large bowl of spaghetti into a smaller set of bowls where there is less spaghetti within a given domain. Making use of an orchestration tool that precisely represents the work flow with message schema specifications between each of the microservices will illustrate the current state of your domain in a clear on concise manner without the periodic need to familiarize yourself at the code layer to unscramble the otherwise perceived spaghetti.

You mentioned a surprise of some kind. What's that about?

Making use of an orchestration tool has some inherent benefits as mentioned (i.e. promotes reusability while providing a representative illustration of the flow of events within your application) but, there's some surprising advantages. From a developers' perspective you'll be given a design tool where you'll build a template of shapes that each represent a given function or process within your work flow. The act of development using the orchestration tool provides a clear and precise illustration of the current

design where you can go from developing a given function to an awareness of how this function fits within the global flow of events. In other words, you'll be given both the top-down and bottom-up understanding of the system as you're developing each component part. The surprising thing that happens with utilizing a tool of this nature is that you'll be made aware of the current set of all functions that one can make use of while viewing the exact representation of the current design as it evolves. One could, for example, build the intended design of the entire system where stubs are created that represent missing functionality. Working from the initial design in this way will provide an understanding as progress is made when functionality is added.

One can organize the shapes with their representing functions into categories to quickly locate reusable functions. Organized categories of functions can include a description to reveal the input and output specifications along with a drill-down feature where a listing of corresponding source code is quickly revealed.

Advantages in summary:

- You'll be building a system with a framework that promotes continuous awareness of the top-down and bottom-up approach to promote the awareness of advantages of building reusable functions that are decoupled from their consumer.
- You'll be presented with an appreciation of how a given function fits into the overall architecture.
- The workflow diagram represents the exact state of development instead of becoming obsolete as the application evolves.
- One can easily set up unit tests where a given function can be tested in isolation by simply defining a DAG that verifies boundary conditions that are defined within the incoming and outgoing connections.
- A break point can be created on a given shape to quickly set up a debug session.
- A simulation can be set up where corresponding shapes are highlighted as they're invoked to illustrate the flow of events for a given set of tests.
- Bottle necks in performance can be easily revealed by collecting elapsed time of execution for a given set of functions by simply turning on the record performance feature.

Glossary:

https://bpm.com/bpm-today/in-the-forum/6267-are-microservices-the-future-of-bpm
Dr. Alexander Samarin writes "Only one step is necessary – the vendors of BPM-suite tools must destroy their monolith approach to BPM."

https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design
Top-Down Bottom-Up software