Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA E SISTEMAS

Service Workflow Engine – Cloud Evolution

Relatório de Estágio para a obtenção do grau de Mestre em Engenharia Informática
Especialização em Desenvolvimento de Software

Autor

Alexandre Jorge Fernandes de Pinho

Orientador

Jorge Barbosa

Supervisores na empresa Altice Labs

Carlos Manuel Rodrigues

Gilberto Matos

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, Junho de 2023

Agradecimentos

Gostaria de agradecer em primeiro lugar à minha família por todo o apoio, carinho e compreensão que me deram ao longo de todos estes anos e, dizer também, que sem eles teria sido tudo mais difícil. A todos eles, uma palavra de agradecimento.

Agradeço ainda a todos os meus amigos, que me deram durante estes últimos anos muitas alegrias e muitos momentos de felicidade. A todos eles, uma palavra de apreço desejando-lhes muitas felicidades.

Agradeço ao meu orientador do ISEC, o Professor Jorge Barbosa do Departamento de Engenharia Informática e Sistemas do Instituto Superior de Engenharia de Coimbra (ISEC), pelo tempo dispensado e pela ajuda prestada durante este período.

À Inova Ria pela atribuição de uma bolsa de investigação.

Por último, mas não menos importante, aos meus orientadores da Altice Labs, Carlos Manuel Rodrigues e Gilberto Matos, e também a restante equipa, o meu agradecimento pela disponibilidade e atenção prestada que generosamente me dedicaram.

Resumo

Atualmente, as empresas têm muitos processos que são executados diariamente. Para reduzir custos e tempo utilizam-se os designados motores de execução de fluxos que permitem gerir, eficientemente, os processos de negócio de forma guiada. Também ajuda a prevenir erros, permite a tomada de decisões entre várias alternativas possíveis mais rapidamente e os funcionários podem colaborar de forma mais produtiva e expedita.

A principal vantagem do uso de motores de execução de fluxos é o de permitir qualquer pessoa, que não tem experiência em programação, de desenhar fluxos com diversas capacidades.

A Altice Labs já possui um motor de execução de fluxos que permite executar fluxos de forma guiada, mas o mesmo possui algumas limitações. O objetivo principal deste trabalho de estágio, realizado na empresa Altice Labs, foi investigar aplicações similares existentes e aferir-se se as mesmas se enquadram no modelo de negócio da empresa. Caso não se enquadrassem, como se verificou, proceder-se-ia à implementação de um novo motor de execução de fluxos adequado às necessidades atuais da Altice Labs.

Com esse objetivo, começou-se por realizar um estudo comparativo entre dois motores de execução de fluxos muito conhecidos e utilizados, o *Netflix Conductor* e o *Uber Cadence*. Para se melhor aferir a sua validade e adequação à empresa efetuou-se uma análise prática, na qual estes dois motores foram instanciados e colocados à prova no mesmo ambiente e com os mesmos testes. Conclui-se que os mesmos têm limitações relativamente ao pretendido. Com este estudo, também se recolheram informações importantes, nomeadamente quais os pontos fortes de cada um deles de modo a incorporar essas mais valias detetadas na implementação do novo motor a desenvolver.

Depois de concluído o estudo e como se conclui que nenhum dos motores existentes se adequava às necessidades procedeu-se à implementação de um novo motor de execução de fluxos para colmatar as lacunas existentes no atualmente existente.

O novo motor desenvolvido encontra-se às necessidades da Altice Labs e, desse modo, todos os objetivos propostos para este estágio foram cumpridos com sucesso.

Palavras-Chave: *atividades*, *fluxos, motor de execução de fluxos, ordens, verticles*

Abstract

Nowadays, companies have many processes that are performed daily. To reduce cost and time, workflow execution engines are used to efficiently manage business processes in a guided manner. It also helps to prevent errors, it enabled decision-making between multiple possible alternatives more quickly, and employees can collaborate more productively and expeditiously.

The main advantage of using workflow execution engines is to allows anyone, who has no programming experience, to design workflows with different capabilities.

Altice Labs already has a workflow execution engine that allows running workflows in a guided manner, but it also has some limitations. The main goal of this internship project carried out at Altice Labs, was to investigate the existing similar applications and assess whether they fit into the company's business model. If they didn't fit in, they would implement a new workflow execution engine suited to the current needs of Altice Labs.

With these goals, a comparative study was carried out between the two execution engines very well know Netflix Conductor and Uber Cadence. To better assess its validity and suitability for the company, a practical analysis was carried out, in which the two engines were instantiated and put to the test in the same environment and with the same tests. This test concluded that they have limitations with what was intended. With this study, important information was also collected, namely what are the strengths of each one of them to incorporate these added values detected in the implementation of the new engine to be developed.

After completing the study and as it was concluded that none of the existing engines suited the needs, a new workflow execution engine was implemented to fill in the existing gaps in the current one.

The new engine developed meets the needs of Altice Labs and, therefore all the objectives proposed for this internship were completed.

Keywords: *activities, orders, verticles, workflow execution engine, workflows*

Índice

Lista de figuras

Lista de tabelas

Definições e Abreviaturas

API – *Application Programming Interface*.

BPMN – *Business Process Model and Notation* – São um conjunto de anotações que ajudam a organizar os processos de negócio e melhoram a comunicação.

Endpoint - É o caminho utilizado por uma conexão *http* para realizar um pedido de informação a um servidor.

HTTP – *HyperText Transfer Protocol* – Protocolo de comunicação utilizado e baseado na *web*.

ISEC – *Instituto Superior de Engenharia de Coimbra*.

JSON – *JavaScript Object Notation* – Formato de comunicação, através de uma norma de representação de informação.

Payload – É o conjunto de dados, por exemplo em *JSON*, que é enviado num pedido *http*.

SCRUM – É uma abordagem utilizada para o desenvolvimento de *software* e para a gestão desses projetos numa perspetiva ágil e de um modo iterativa e incremental.

Software – Conjunto de instruções que devem ser seguidas e processadas por um computador.

Sprint – Termo utilizado na abordagem SCRUM e que se refere a um ciclo de desenvolvimento. Um *sprint* decorre, geralmente, durante o período de uma a quatro semanas.

SWE – Service Workflow Engine.

UML – *Unified Modeling Language* – é uma linguagem de modelagem unificada e principalmente usada na criação de diagramas de classes.

1 Introdução

O desenvolvimento e uso de orquestradores nas empresas tem benefícios como o tempo e o custo associado à realização de tarefas.

Este relatório decorre da realização do estágio curricular relativo ao Mestrado em Engenharia Informática do DEIS/ISEC/IPC que o autor realizou no departamento de Sistemas de Suporte às Operações (SSO) da Altice Labs. Este estágio realizou-se sobre a orientação do Eng. Carlos Rodrigues e Eng. Gilberto Matos por parte da Altice Labs e do Doutor Jorge Barbosa por parte do DEIS/ISEC/IPC.

Como o atual motor possui algumas limitações, foi realizado um estudo comparativo de dois motores existentes no mercado: o *Netflix Conductor* e *Uber Cadence*. Apesar destes dois motores terem as suas vantagens também têm limitações e por isso procedeu-se à implementação de raiz de um novo motor de execução de fluxos.

No subcapítulo 1.1, será abordado o âmbito de todo o projeto de estágio. Em 1.2, será apresentada a empresa de acolhimento onde foi realizado o estágio (Altice Labs). Em 1.3, será apresentado a instituição de ensino que permitiu a realização do estágio, o ISEC. Em 1.4, será apresentado a entidade responsável pela atribuição da bolsa de investigação, a Inova Ria. Em 1.5, será explicado como ocorreram as reuniões de acompanhamento. Em 1.6, serão apresentados os problemas e o fundamento da proposta de estágio. Em 1.7, serão explicados os objetivos do estágio de forma sintetizada e, em 1.8, será descrita a organização de todo o documento.

1.1 Âmbito

A unidade curricular em que se enquadra este estágio tem como principal objetivo a integração dos alunos num ambiente empresarial para aquisição de competências e conhecimentos. Este objetivo é alcançado através da exposição e respetiva resolução de problemas não triviais e com abordagens novas que exijam o estudo e a análise de soluções, aliada ao desenvolvimento de uma solução do problema e de acordo com as boas práticas em uso no domínio em que se insere.

A Altice Labs é uma empresa que desenvolve soluções de software e de hardware e está organizada em seis diferentes departamentos distintos: Sistemas de Suporte às Operações (SSO), Serviço de Redes e Plataformas (SRP), Estratégia de Inovação e Tecnologia (EIT), Desenvolvimento de Sistemas de Rede (DSR), Desenvolvimento e Suporte do Negócio (DSN) e Digital, Internet e Televisão (DIT).

O estágio teve um regime presencial/híbrido, isto é, o estagiário desenvolveu o mesmo umas vezes na própria Altice Labs e noutras fora da Altice Labs. Havia a obrigatoriedade de, pelo menos, num dia por semana estar nas instalações da Altice Labs em Aveiro. Este estágio teve início no dia 1 de setembro de 2021 e o seu término ocorreu no dia 31 de julho de 2022.

1.2 Altice Labs

A Altice Labs, anteriormente denominada PT Inovação, foi fundada em 1999 e está sediada na Rua Engenheiro José Ferreira Pinto Basto em Aveiro. É uma empresa do

grupo Altice e cujo "*core business*" é o desenvolvimento de novas soluções tecnológicas inovadoras. [1].

1.3 ISEC

O ISEC é uma escola de ensino superior politécnico que está integrada no Instituto Politécnico de Coimbra (IPC).

O ISEC tem como principal missão transmitir conhecimento e preparar os seus alunos para o mundo empresarial e industrial. Ministra atualmente 39 cursos, sendo 12 licenciaturas, 9 mestrados e os restantes 18 são referentes a Cursos Técnicos Superiores Profissionais (CTeSP) [2].

1.4 Inova Ria

A Inova Ria é uma entidade sem fins lucrativos fundada em 2003 que tem como objetivos a "criação e consolidação de um cluster na área das Tecnologias de Informação, Comunicação e Eletrónica, com especial enfoque nas telecomunicações, centrado na Região de Aveiro" [3].

A Inova Ria foi responsável pela atribuição de uma bolsa de investigação ao estagiário, através do programa Genius que é direcionado para todos os que se encontram a terminar licenciaturas ou mestrados [3].

1.5 Reuniões de Acompanhamento

Durante a realização do estágio houve um acompanhamento por parte do Professor Jorge Barbosa bem como por parte dos orientadores da Altice Labs. Este acompanhamento foi realizado através de reuniões quinzenais/mensais dependendo do progresso realizado em termos de estágio e da disponibilidade de ambas as partes. Mensalmente era realizada uma reunião entre todas as partes, orientadores do ISEC e da Altice Labs.

1.6 Descrição do Problema

Como já foi referido, a Altice Labs já possui um motor de execução de fluxos, conhecido internamente como *Service Workflow Engine* (SWE). No entanto o mesmo apresenta algumas limitações, pelo que foi determinado pela empresa depois de um estudo prévio e prospetivo realizado também no âmbito deste estágio, que o estagiário desenvolvesse um novo motor de execução de fluxos que melhor respondesse às necessidades atuais da empresa e dos seus parceiros de negócio.

Estas limitações consistem em não ser possível executar atividades em paralelo e executar *sub-workflows*.

Um motor de execução de fluxos consiste num orquestrador de tarefas e operações a serem executadas em determinado contexto, e tem como objetivo escalonar esse conjunto de operações sem a necessidade de uma interação humana constante.

O operador numa interação inicial parametriza as relações das tarefas, nomeadamente definindo prioridades e sequenciações e depois o motor de execução de fluxos analisa

essa parametrização e constrói uma solução de execução das mesmas de uma forma otimizada e coerente.

Atendendo às limitações do *SWE*, concluiu-se ser mais vantajoso desenvolver um novo motor de "raiz" do que melhorar/refatorar o atual. Isto deve-se ao facto de as alterações necessários exigirem bastante tempo para serem implementadas devido à complexidade existente no *SWE*. Para além disso, no desenvolvimento de um novo motor de execução de fluxos pode-se incorporar novas tecnologias o que é também uma mais valia e vantagem. No entanto teve-se em atenção a experiência adquirida com a utilização do antigo motor de modo a não se cometer os mesmos erros nem de ter as mesmas limitações que o *SWE* apresenta.

1.7 Objetivos do Estágio

A proposta de estágio (Anexo A), apresentada pela Altice Labs ao ISEC, tem como foco os seguintes objetivos:

- Avaliação de motores de execução de fluxos existentes na comunidade *open source*;
- Em caso de se optar pela implementação de um novo motor, realizar a definição do modelo de dados de suporte aos workflows e do modelo operacional;
- Definição das APIs de CRUD e de execução dos workflows, seguindo a metodologia Secure by design;
- Atividades - Criação das atividades que a ferramenta deveria oferecer como unidades básicas de execução;
- Execução - Desenvolvimento do motor de execução de atividades;
- Testes Unitários e de Integração - Montar cenário de testes de integração integrado na *cloud* e implementação dos testes unitários.

A execução de cada um dos objetivos por ordem cronológica, ao longo do estágio, é apresentado na Tabela 1.

Tabela 1 - Objetivos do Estágio

Tarefas	Set.	Out.	Nov.	Dez.	Jan.	Fev.	Mar.	Abr.	Maio	Jun.	Jul.
Estado da arte											
Definição do modelo de dados											
Definição das *APIs* de *CRUD* e de execução											
Implementação das atividades											
Desenvolvimento do motor de execução de atividades											
Testes unitários e de integração											

1.8 Estrutura do Documento

Além deste capítulo introdutório, o presente documento encontra-se dividido em mais 6 capítulos e incorpora também cinco anexos:

- Capítulo 2 – Neste capítulo descreve-se a realização de um estudo comparativo, baseado na utilização e análise de duas ferramentas de execução de fluxos diferentes. Este estudo permitiu concluir que nenhuma das ferramentas existentes e analisadas se mostrou adequada às necessidades da Altice Labs e como tal era necessário desenvolver uma outra de raiz que respondesse a essas necessidades. Este estudo também serviu para se obter informação sobre as vantagens de cada uma das ferramentas analisadas e perspetivar a sua inclusão no motor a desenvolver;
- Capítulo 3 – Apresenta uma contextualização sobre o protocolo *Business Process Model and Notation* (BPMN);
- Capítulo 4 – O contexto tecnológico envolvido na realização deste estágio bem como as tecnologias e as ferramentas utilizadas são descritas neste capítulo;
- Capítulo 5 – Apresenta-se aqui a metodologia de desenvolvimento de *software* seguida e os respetivos processos de desenvolvimento adotados;
- Capítulo 6 – Apresenta-se e detalha-se o trabalho desenvolvido relativo ao desenvolvimento do novo motor de execução de fluxos. São também referidos

e enquadrados os motores existentes que foram considerados para o estudo prospetivo que foi realizado bem como os conhecimentos e funcionalidades decorrentes desse estudo e que poderiam vir a ser incorporados no novo motor a desenvolver. Para concluir apresenta-se o resultado final e as vantagens do novo orquestrador que foi desenvolvido, realizando para isso uma análise comparativa com o na altura em funcionamento na Altice Labs.

- Capítulo 7 – Neste capítulo é analisada a conclusão tirada sobre o trabalho realizado e apresentam-se as principais contribuições e dificuldades encontradas. Também se perspetiva trabalho futuro a realizar de modo a poder melhorar-se o motor desenvolvido tendo em consideração a experiência adquirida no seu desenvolvimento e novas funcionalidades que da sua utilização se evidenciaram como necessárias e que não estavam consideradas na proposta inicial de desenvolvimento.

Na parte final deste documento, são indicadas as referências bibliográficas utilizadas na elaboração do trabalho realizado bem como os anexos produzidos ao longo do mesmo:

- Anexo A – Proposta de estágio da Altice Labs ao ISEC;
- Anexo B – Comparação entre motores de execução de fluxos (documento elaborado para a Altice Labs);
- Anexo C – Diagrama de classes de dados;
- Anexo D – Exemplo de *payload* para a criação de um novo fluxo;
- Anexo E – Exemplo de *payload* relativo a um pedido de execução de um fluxo.

2 Motor de Execução de Fluxos

O conceito de motor de execução de fluxos aqui apresentado descreve a comparação realizada entre os motores *Netflix Conductor* e *Uber Cadence*. As vantagens do novo motor de execução de fluxos que foi desenvolvido comparativamente ao atualmente existente, anteriormente implementado pela Altice Labs, são também apresentadas.

2.1 Motor de Execução de Fluxos

Um motor de execução de fluxos é uma ferramenta desenhada para facilitar a gestão de processos nas empresas. Processos que podem ser executados de forma automatizada, eficiente, eficaz e consistente. O uso de motores de execução de fluxos não há necessidade de uma interação humana. Por exemplo, um processo para ser concluído, tem de ser analisado por um conjunto de pessoas tendo para isso de passar por vários departamentos. Esta metodologia quando mal feita, pode ter consequências para as empresas como perda de tempo e dinheiro. Para colmatar ou minimizar estas eventuais ineficiências utilizam-se os motores de execução de fluxos que têm como principais funcionalidades escalonar de um modo otimizado a execução desse conjunto de atividades. Como referido anteriormente, uma das principais vantagens é a possibilidade de criar fluxos sem ter conhecimentos em qualquer linguagem de programação.

Um fluxo pode ser constituído por um conjunto de atividades. Estas atividades podem ser de diversos tipos:

- *Start*, atividade que indica o início do fluxo, ou seja, a primeira atividade a ser executada;
- *End*, atividade que indica a última atividade a ser executada;
- *Decision*, quando é pretendido fazer uma validação;
- *Http*, atividade usada para fazer pedidos *http*;
- *Fork*, atividade responsável pela execução de atividades em simultâneo, ou seja, bifurcação das atividades;
- *Join*, atividade responsável pela junção das atividades em paralelo;
- *Sub-workflow*, estas atividades são usadas para executar outros fluxos dentro do fluxo principal;
- *Error*, é uma atividade que é usada para terminar a execução do fluxo abruptamente. Há semelhança da atividade *end* esta também é uma atividade final;
- *Event* é uma atividade usada quando é pretendido aguardar uma notificação de um evento externo;
- *Notification* é uma atividade para informar/notificar que algo aconteceu;
- *Manual*, atividade que indica a necessidade de uma intervenção manual por parte de um utilizador;
- Entre outras atividades customizadas ou criadas estendidas pelos clientes.

Tanto os fluxos como as suas atividades possuem *inputs* e *outputs*. Os parâmetros dos *inputs* são introduzidos pelo utilizador, podendo ser um valor final ou um *json path* para realizar pesquisas no objeto. Cada tipo de atividade possui um modelo definido de *inputs* e *outputs*, ou seja, uma atividade do tipo *Http* tem diferentes parâmetros de *input* e *output* que uma atividade *Decision*.

As atividades também possuem políticas de *retry* e de *timeout*. A política de *retry* nas atividades é usada para indicar o número de vezes que a atividade é reexecutada em caso de ocorrer uma falha durante a sua execução. A política de *timeout* nas atividades é usada para estabelecer o tempo máximo que cada atividade tem para terminar a sua execução, caso contrário a atividade é terminada em erro.

2.2 Comparação entre motores da *Netflix* e *Uber*

O estudo consistiu na análise e comparação entre dois motores de execução de fluxos: o *Netflix Conductor* e o *Uber Cadence*. O motivo da escolha destes motores foi o facto de serem gratuitos e *open source*. Foram também escolhidos por serem motores com muitas capacidades e que vão ao encontro de alguns requisitos e necessidades da Altice Labs. Foi também uma sugestão por parte da empresa por ter sido efetuada uma breve análise e terem concluído que estes dois motores têm grande capacidade de execução de fluxos, conseguindo executar centenas por segundo. Este é um dos requisitos da *Altice Labs*, que consiste ter a capacidade de executar fluxos na ordem das centenas.

Para realizar a comparação entre as duas ferramentas é necessário criar diferentes fluxos, com propósitos diferentes, e executá-los nos dois orquestradores para se obterem conclusões. O objetivo desta comparação foi perceber se estes dois motores possuem as condições necessárias para ser usado e integrado na Altice Labs.

2.2.1 Preparação do estudo realizado

Antes de dar início à análise dos dois motores foi necessário, numa primeira fase, instalá-los. Para isso, os motores foram instanciados usando *Docker* e usando as imagens oficiais dos respetivos motores.

De seguida, foi definido um plano para analisar o potencial dos dois motores. O plano era constituído por todas as funcionalidades que deveriam ser colocadas à prova e foram as seguintes:

- Identificadores dos fluxos e das execuções;
- Comportamento de um sub-workflow;
- Comportamento no cancelamento de uma execução;
- Paralelismo;
- Capacidade de reexecutar um fluxo;
- Capacidade de pausar e retomar execução;
- Comportamento das atividades do tipo *wait*;
- Capacidade de fazer *skip* a uma atividade;
- Falha durante a execução de uma atividade;
- Comportamento de eventos e sinais;
- Capacidade de manipular variáveis do fluxo durante uma execução;
- Comportamento dos *timeouts* nos fluxos e atividades.

As conclusões deste estudo encontram-se na secção 2.5.

Para terminar a fase de preparação foi necessário desenhar e criar um conjunto de fluxos em cada um dos motores. Os fluxos criados tiveram de ser os mesmos, para os dois motores, para que o estudo seja realizado usando os mesmos dados de entrada.

A Figura 1 é um exemplo de um dos fluxos criados para a realização do estudo.

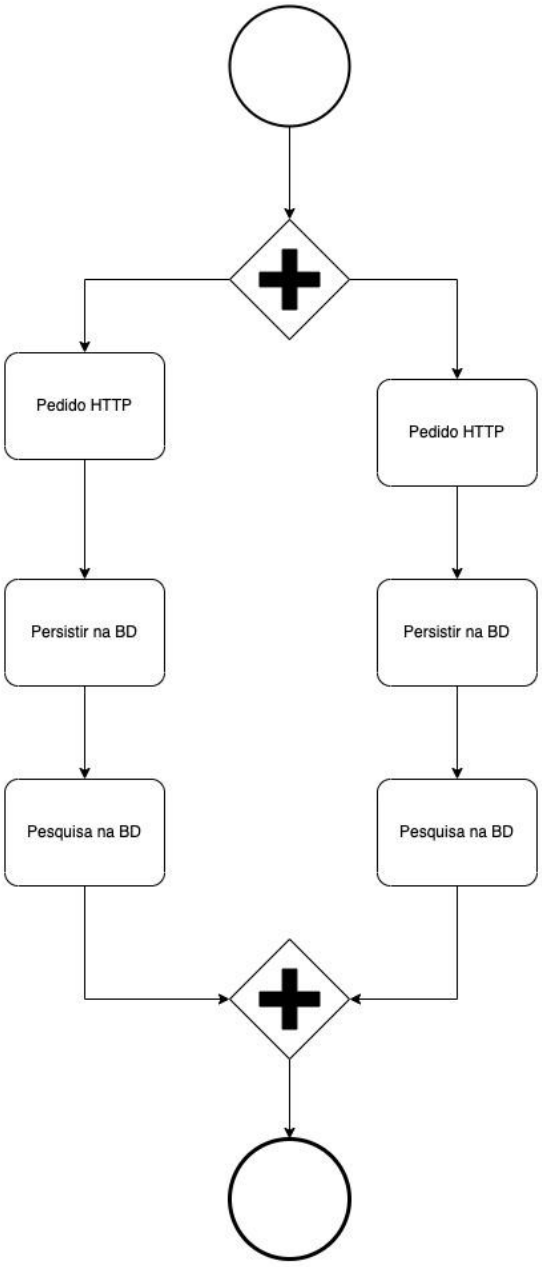


Figura 1 - Fluxo para analisar paralelismo

Fonte: Criado pelo autor

Este fluxo foi usado para compreender o comportamento dos fluxos na existência de paralelismo, ou seja, quando existem atividades a executar em simultâneo.

2.3 Netflix Conductor

O *Netflix Conductor* é um motor de execução de fluxos usado e desenvolvido pela *Netflix*. Foi desenvolvido em *Java* e também possui uma *interface web* para acompanhar a execução de um fluxo. Este motor disponibiliza uma *API* para fazer a gestão dos fluxos e o controlo das execuções. Os fluxos são definidos e criados usando *JSON*.

2.4 Uber Cadence

O *Uber Cadence* também é um motor de execução de fluxos usado e desenvolvido pela *Uber*. Foi desenvolvido em *Go* e também possui uma interface web para acompanhar a execução de um fluxo. A gestão e controlo dos fluxos e atividades é feita através de código. Para tal é preciso usar a *framework* da *Uber* para realizar estas operações.

2.5 Comparação entre os dois motores

Os dois motores foram instanciados e testados. Como referido, os testes realizados tiveram como objetivo recolher informação das diferentes abordagens usadas pelos dois motores e dos seus pontos fortes e fracos. A informação obtida nos testes foi usada na implementação do novo motor de execução de fluxos.

2.5.1 Identificadores dos fluxos e suas execuções

2.5.1.1 Netflix Conductor

Os fluxos são constituídos por um identificador único. Quando um fluxo inicia a execução o motor gera um *id* para cada uma das atividades. Estes *id* das atividades são usados quando se pretende reexecutar um fluxo a partir de uma certa atividade ou fazer o *skip* da mesma.

2.5.1.2 Uber Cadence

Os *fluxos* são constituídos por dois identificadores: *workflow id* e *run id*. O *workflow id* é o identificador do fluxo e só pode existir um fluxo com o mesmo *id* em execução ao mesmo tempo. Se houver uma intenção de executar um *fluxo* com um *id* igual a um *fluxo* que esteja em execução, o que estiver a ser executado é terminado e o "novo" é executado.

O *run id* é o identificador de execução. Quando existe um fluxo que terminou a execução e de seguida pretende-se reexecutá-lo, o novo fluxo irá ter o mesmo *workflow id,* mas é gerado um novo *run id*, ou seja, pode- se executar um fluxo já executado (fluxo com o mesmo *id*) mas o *run id* vai ser diferente.

2.5.2 Sub-Workflow

2.5.2.1 Netflix Conductor

Quando um *sub-workflow* é executado, o fluxo principal fica à espera que o *sub-workflow* termine.

2.5.2.2 Uber Cadence

Quando um *sub-workflow* é executado, o fluxo principal dá continuidade à execução das próximas atividades.

2.5.3 Cancelar execução de fluxo

2.5.3.1 Netflix Conductor

Quando o motor recebe instrução para cancelar a execução de um fluxo, ele é terminado de imediato. Caso exista uma atividade em execução quando for realizado um pedido de cancelamento, o fluxo é terminado e a resposta da atividade é "descartada" pelo motor.

2.5.3.2 Uber Cadence

Quando o motor recebe uma instrução para cancelar a execução de um fluxo, vai ser desencadeado um pedido de cancelamento da atividade que está a ser executada (caso exista). Como a execução do fluxo só termina quando a execução da atividade terminar, a atividade tem de usar um mecanismo de *heartbeat* para verificar o estado do fluxo, ou seja, tem de usar este mecanismo periodicamente para fazer esta validação e, porventura, terminar a execução da atividade de forma abrupta.
Heartbeat – processo para assinalar que a atividade ainda está em execução. Também é usado para guardar o progresso da atividade, ou seja, se a atividade falhar ela pode ser executada novamente recomeçando na etapa em que se encontrava. Este mecanismo é usado para detetar falhas relativas a erros específicos. Neste caso é relativa ao cancelamento de um fluxo.
Se não existir nenhuma atividade em execução, o fluxo é terminado imediatamente.

2.5.4 Paralelismo

Possibilidade de ter duas ou mais atividades em execução ao mesmo tempo. Esta funcionalidade funciona de igual forma nos dois motores.

2.5.5 Reexecutar fluxo

2.5.5.1 Netflix Conductor

Existem três mecanismos para efetuar-se a reexecução de fluxos:

- Reiniciar um fluxo que já esteja terminado e que se encontre num estado final. Necessário indicar *id* do fluxo;
- Reexecutar fluxo a partir da última atividade que falhou. Este mecanismo utiliza os mesmos valores de entrada do fluxo e da atividade que falhou. Necessário indicar *id* do fluxo*;*
- Reexecutar fluxo a partir da atividade que pretendemos. É possível introduzir dados relativos à entrada do fluxo e ao começo da atividade. É necessário indicar o *id* do fluxo e o *id* da atividade para realizar esta operação.

Não é possível reexecutar um fluxo, se este ainda estiver a ser executado.

É possível reexecutar um fluxo se este estiver terminado e que se encontre num estado final. A nova execução irá ter o mesmo *workflow id*.

2.5.5.2 Uber Cadence

Existem duas possibilidades de reexecutar um fluxo:

- Reexecutar a partir do início do fluxo usando os mesmos parâmetros de entrada. Necessário indicar *id* do fluxo;
- Reexecutar a partir da última atividade que falhou usando os mesmos parâmetros de entrada do fluxo e da atividade. Necessário indicar *id* do fluxo.

Não há a possibilidade de reexecutar o fluxo com dados de entrada diferentes dos definidos inicialmente.

Se uma atividade tem o parâmetro "número máximo de *retries"* definido e a atividade ultrapassa esse número de tentativas, o fluxo é imediatamente terminado em erro. De seguida, se reexecutar o fluxo a partir da última atividade que falhou vai gerar novamente o mesmo erro. É possível concluir que o contador do número de vezes que a atividade foi executada não fez *reset* e, por isso, sempre que for pretendido reexecutar o fluxo a partir da última atividade vai dar o mesmo erro.

Se for reexecutado um fluxo (com certo *id*) e este ainda estiver em execução, o fluxo atual é terminado e o novo será iniciado.

2.5.6 Pausar/Continuar execução

2.5.6.1 Netflix Conductor

Existência de dois comandos para pausar e retomar a execução de um fluxo.

2.5.6.2 Uber Cadence

O *Uber Cadence* não possui um mecanismo para pausar e/ou continuar execução. A única possibilidade para realizar estas operações é através do envio de sinais. Por exemplo, ter uma variável interna ao fluxo que é manipulada (por exemplo um booleano) sempre que recebe um sinal. Com a manipulação desta variável é possível controlar o fluxo.

2.5.7 Atividades do tipo *wait*

2.5.7.1 Netflix Conductor

Existe uma atividade do tipo *Wait* que fica com status IN_PROGRESS à espera que seja terminada por um evento externo.

2.5.7.2 Uber Cadence

Utilizando o método *await* da biblioteca do *Uber* é possível esperar até que determinado parâmetro seja "verdadeiro", ou seja, que vá ao encontro do requisitado por quem cria o fluxo. Para esse parâmetro sofrer alteração e o fluxo continuar a execução são usados sinais para manipular o valor do parâmetro.

2.5.8 *Skip* de uma atividade

2.5.8.1 Netflix Conductor

Existência de comando que permite "saltar" a atividade, ou seja, a atividade não é executada. Se a atividade que for "saltada" (*skipped*) estiver em execução, o fluxo executa a próxima atividade e descarta os resultados retornados pela atividade que foi "saltada". É necessário indicar o nome da atividade que se pretende fazer o *skip*.

2.5.8.2 Uber Cadence

Não possui esta funcionalidade, mas há a possibilidade de usar um sinal para não executar uma certa atividade. Não é eficaz porque teria de existir um parâmetro para cada atividade. E, caso seja enviado um sinal para saltar uma determinada atividade e essa atividade estiver em execução esta ação não terá efeito. Ou seja, o fluxo conclui que a atividade foi executada.

2.5.9 Falha de uma atividade

Quando uma atividade falha durante a sua execução, o motor pode fazer um pedido para executar novamente essa atividade. Para isso, a atividade tem de ter definida o parâmetro "número máximo de *retries*" (maior que zero) e o tempo máximo da sua execução. Este comportamento é igual nos dois motores.

2.5.10 Modificar variáveis do fluxo em execução

2.5.10.1 Netflix Conductor

Existência de uma atividade do tipo "*SET_VARIABLE*" que permite criar, inicializar e modificar variáveis no fluxo. Este tipo de atividade, à semelhança de todos os outros tipos de atividades, tem de ser definida no fluxo.

Durante a execução do fluxo não é possível criar/alterar variáveis através de sistemas externos, como acontece no *Uber Cadence*.

2.5.10.2 Uber Cadence

Através de sinais é possível modificar variáveis do fluxo. As variáveis podem ser alteradas durante a execução do fluxo através do envio de sinais.

2.5.11 Timeouts

Possibilidade de definir vários *timeouts*, que são:

- Tempo máximo para a execução de um fluxo (antes do fluxo transitar para o estado "*TIMED_OUT*");
- Tempo máximo para a execução de uma atividade (antes da atividade transitar para o estado "*TIMED_OUT*");
- Número máximo de *retries*;

Existe também a possibilidade de definir o tempo de espera antes de realizar um *retry* num fluxo ou atividade. É um intervalo de tempo desde que o fluxo ou atividade falha até ser reexecutado.

Este comportamento é igual nos dois motores.

2.6 Conclusão da Comparação

Após a realização dos testes no *Netflix Conductor* e *Uber Cadence* é possível concluir que ambos os motores têm os seus pontos fortes e fracos, mas nenhum se adequa às necessidades da empresa. Na opinião do autor o *Netflix Conductor* é o mais competente porque apresenta diversas e diferentes abordagens e configurações para definir e executar os fluxos e as suas atividades, ou seja, é mais dinâmico e versátil. Na Tabela 2 é apresentado um resumo da comparação entre os motores *Netflix Conductor e Uber Cadence*. Esta tabela também é constituída por uma coluna que indica a expetativa da *Altice Labs* sobre cada funcionalidade testada, ou seja, o comportamento desejado.

Embora não exista um motor de execução de fluxos perfeito que vá ao encontro do modelo de negócio da Altice Labs, recolheu-se informação sobre as diferentes abordagens usadas e as diferentes funcionalidades de cada motor de execução de fluxo. Estas abordagens e funcionalidades servirão de apoio na implementação do novo motor. Um dos requisitos na implementação do novo orquestrador é seguir a norma *Business Process Model and Notation* (*BPMN*).

Tabela 2 - Comparação entre os motores Netflix Conductor e Uber Cadence

	Netflix Conductor	Uber Cadence	Expetativa da Altice Labs
Identificação dos fluxos e suas execuções	O fluxo e execução possuem o seu próprio identificador. As atividades também possuem identificador gerado no momento da execução do fluxo.	O fluxo e execução possuem o seu próprio identificador. Não permite executar o mesmo fluxo em simultâneo.	Cada fluxo e execução ter o seu próprio identificador. O identificador das atividades é o nome da atividade.
Atividade Sub-workflow	Aguardar que esta atividade sub-workflow termine a execução e só depois prosseguir com a execução das	O fluxo não aguarda que esta atividade termine a sua execução e prossegue com a execução das	Aguardar que esta atividade sub-workflow termine a execução e só depois prosseguir com a execução

	restantes atividades.	próximas atividades.	das restantes atividades.
Cancelar	A execução é terminada imediatamente independentemente se ainda existir alguma atividade em execução.	A execução só é cancelada quando já não existirem atividades em execução.	A execução ser colocada num estado "CANCELLING" mas só é terminado quando já não existir nenhuma atividade em execução.
Atividade Paralelismo	Permitir executar atividades em simultâneo.	Permitir executar atividades em simultâneo.	Permitir executar atividades em simultâneo.
Reexecutar fluxo	Permite reexecutar fluxo a partir da última atividade que falhou ou de uma determinada atividade selecionada pelo utilizador.	Apenas permite reexecutar fluxos a partir da primeira atividade ou da última atividade que falhou.	Reexecutar fluxo a partir de uma determinada atividade.
Pausar/Continuar Execução	Permite pausar e continuar uma execução.	Não permite executar estas operações.	Permitir pausar e continuar uma execução.
Atividades do tipo *Wait*	Aguarda notificação de um evento externo. Enquanto aguarda este evento a atividade fica em estado "IN_PROGRESS".	Aguarda notificação até uma determinada variável, definida pelo utilizador, for manipulada.	Aguardar notificação de um evento externo. Enquanto aguarda este evento a atividade fica em estado "WAITING".
Skip de uma atividade	Permite fazer *skip* de atividades mesmo que estas estejam em execução.	Não possui esta funcionalidade. Existem alternativas, mas não são eficazes.	Permitir fazer *skip* de atividades, mas estas não podem estar num estado final (terminado) ou em execução.
Falha de uma atividade	Existência de políticas de *retries*.	Existência de políticas de *retries*.	Existência de políticas de *retries*.
Modificar variáveis do fluxo em execução	Existência de uma atividade que permite definir variáveis que depois podem ser usadas no fluxo. Não	As variáveis podem ser definidas no fluxo. Existe possibilidade de redefinir os valores das variáveis	O fluxo poder ter como dados de entrada um conjunto de variáveis. Não ser possível, durante a

	permite modificar os valores destas variáveis durante a execução.	durante a execução do fluxo.	execução, manipular estes valores.
Timeouts	Permite definir um tempo máximo que as atividades e fluxos têm para terminar a sua execução.	Permite definir um tempo máximo que as atividades e fluxos têm para terminar a sua execução.	Permitir definir um tempo máximo que a atividade tem para executar.

3 BPMN

BPMN é um protocolo que foi desenvolvido com o objetivo em criar uma linguagem comum para modelagem de processos de negócio [4].

O BPMN é constituído por um conjunto de símbolos e notações que facilita a criação de fluxos. De seguida, são apresentados os símbolos usados durante o projeto e que são fundamentais para a criação de um fluxo.

3.1.1.1 Início de Evento

O símbolo de início de evento indica o primeiro passo de um processo, ou seja, a primeira atividade a ser executada. Este símbolo é representado conforme é mostrado na Figura 2.

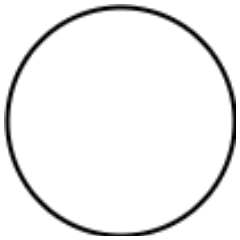


Figura 2 - Símbolo de Início de Evento

Fonte: Criado pelo autor

3.1.1.2 Final do Evento

O símbolo de final do evento é referente ao último passo de um processo, ou seja, é a última atividade a ser executada. Este símbolo é representado conforme é amostrado na Figura 3.

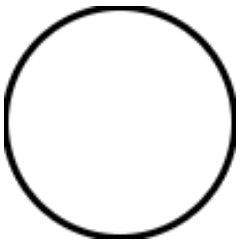


Figura 3 - Símbolo de Final de Evento

Fonte: Criado pelo autor

3.1.1.3 Exclusivo

O símbolo exclusivo realiza um conjunto de validações e com base nas condições separa o fluxo em dois ou mais caminhos mutuamente exclusivos. Este símbolo pode ser equiparado a um *if*. Este símbolo é representado conforme é amostrado na Figura 4.

Figura 4 - Símbolo de Exclusividade

Fonte: Criado pelo autor

3.1.1.4 Paralelo

O símbolo do paralelo não depende de condições ou eventos, mas de representar duas ou mais tarefas em simultâneo. Este símbolo é usado quando é pretendido executar atividades em paralelo, isto é, ao mesmo tempo. Normalmente é usado em situações de *FORK* e *JOIN*. O símbolo é representado conforme é amostado na Figura 5.

Figura 5 - Símbolo de Paralelismo

Fonte: Criado pelo autor

3.1.1.5 Erro

O símbolo de erro é usado quando ocorre um erro no início, meio ou fim do processo. Quando este evento é acionado o processo é terminado em erro. É considerado uma atividade final semelhante ao símbolo de final do evento. Este símbolo é representado conforme é amostrado na Figura 6.

Figura 6 - Símbolo de Erro

Fonte: Criado pelo autor

3.1.1.6 Atividade

O símbolo da atividade é usado quando é pretendido executar uma tarefa. Um exemplo da utilização deste símbolo é uma atividade *http* quando pretende-se realizar um pedido *http*. Este símbolo é representado conforme é amostrado na Figura 7.

Figura 7 - Símbolo da Atividade

Fonte: Criado pelo autor

4 Contexto Tecnológico

Neste capítulo são apresentados as tecnologias e ferramentas utilizadas durante o período do estágio.

4.1 Tecnologias

4.1.1 Java

Java é uma linguagem de programação orientada a objetos e foi usada no desenvolvimento do motor de execução de fluxos. A escolha desta linguagem de programação em vez de outra foi o facto de ser a linguagem usada noutros projetos e a mais indicada para o novo motor de execução.

4.1.2 JSON

JSON é o formato usado na serialização/*desserialização* de dados. Normalmente usado nos *http requests*.

4.1.3 Quarkus

Quarkus é a *framework* Java usada no desenvolvimento da ferramenta. O *Quarkus* é focado no desenvolvimento em *cloud* e tem como vantagens os tempos de execução menores em comparação com outras *frameworks*. [5]

4.1.4 Vert.x

Vert.x é uma *framework* idealizado para a construção de aplicações reativas e é facilmente integrável em aplicações que são executadas na JVM (*Java Virtual Machine*). O Vert.x contém bibliotecas para fazer a conexão à base de dados, comunicação entre brokers, comunicação HTTP entre outras [6].

Uma aplicação reativa é um sistema sem bloqueios, assíncrono e orientado a eventos.

O *event bus* do *Vert.x* permite que diferentes partes de uma aplicação comuniquem entre si. Existe apenas uma instância de *event bus* por cada instância de *Vert.x* e por *default* existem duas *threads* de *event loop* por cada *core* existente no processador. Os *event loop* tem como objetivo estar constantemente “à escuta" de novos eventos no *event bus* e sempre que surge um novo evento este é entregue à instância que sabe tratar do mesmo. A Figura 8 representa a estratégia de *Event Loop* no *Vert.x* [6].

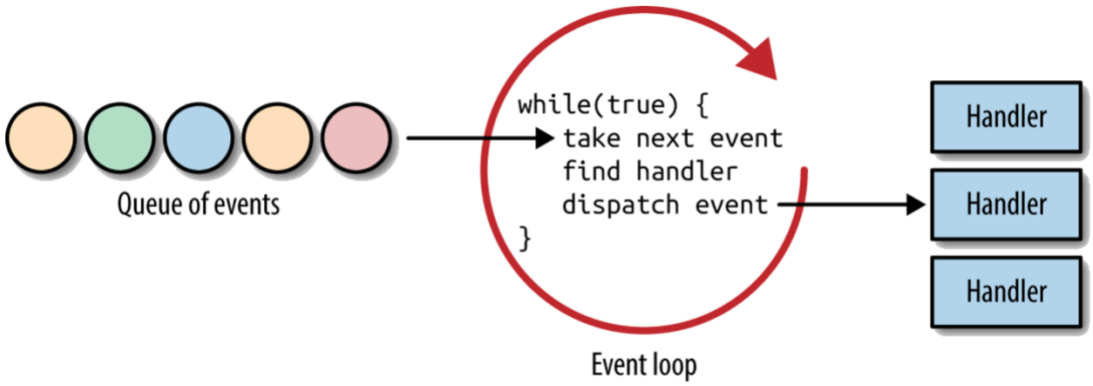


Figura 8 - Representação do Consumo de Eventos

Fonte: https://www.deliver-better.com/post/eclipse-vert-x-basics, 10/08/2022

O Vert.x tem um conceito conhecido como *Verticle*. Um *verticle* é uma classe que tem acesso aos objetos da plataforma *Vert.x* e é basicamente utilizado para organizar código em unidades independentes e modulares. Existe também facilidade em escalar verticalmente a aplicação através da manipulação do número de instâncias para cada *verticle*. Cada instância dos *verticles* é executada numa *thread* em separado. Os *verticles* podem comunicar entre si (na mesma instância) ou com outros (em instâncias diferentes). Esta comunicação é feita através do *event bus* do *Vert.x* [6].

É possível criar uma solução na plataforma *Vert.x* utilizando uma ou mais linguagens de programação. Atualmente, é compatível com *Java*, *Kotlin*, *JavaScript*, *Groovy*, *Ruby* e *Scala*, ou seja, permite a comunicação entre *verticles* mesmo que estes estejam em linguagens diferentes [7].

Finalizando, o *Vert.x* permite a criação de aplicações leves, modernas e de alta *performance*.

4.1.5 Mutiny

Mutiny é uma biblioteca de programação orientada a eventos. É baseado em *Reactive Streams*, tem I/O não bloqueantes e atua de forma assíncrona à semelhança do *Vert.x* [8].

4.1.6 Relação entre *Vert.x* e *Mutiny*

A principal diferença entre a API do *Vert.x* e a variante *Mutiny* é que a API do *Vert.x* aciona a operação assim que o método é chamado enquanto no *Mutiny* é necessário subscrever o evento para que a operação seja acionada.

O *Vert.x* pode ser usado em dois formatos diferentes: utilizando a API *standard* do próprio *Vert.x* ou utilizando a variação com o *Mutiny*. A junção entre estas duas tecnologias permite ter uma experiência perfeita com outras *APIs* reativas oferecidas pelo *Quarkus*.

O *Mutiny* também pode ser usado de forma independente, ou seja, sem uma integração com *Quarkus* e *Vert.x*. Porém, o *Quarkus* faz uso, por baixo, das duas bibliotecas [8].

4.2 Ferramentas

4.2.1 IntelliJ

O *IntelliJ* foi o *IDE* (*Integrated Development Environment*) utilizado para desenvolver o *software*. O autor escolheu este *IDE* como ferramenta de trabalho pois já estava familiarizado com o *IntelliJ*.

4.2.2 Postman

A ferramenta *Postman* é usada para realizar testes em API's. O Postman foi usado para realizar testes nas operações de CRUD, como o POST, GET, PUT e DELETE. Esta ferramenta foi a escolhida por ser das mais conhecidas e acolhidas pelos desenvolvedores de software devido às suas capacidades.

4.2.3 MongoDB

O MongoDB é uma base de dados *NoSQL* orientada a documentos usada para armazenar um grande volume de dados. Esta base de dados faz uso de *collections* e *documents*. Os fluxos e as suas execuções são guardados no Mongo. A escolha desta base de dados não relacional foi realizada em equipa.

4.2.4 MongoDB Compass

O *MongoDB Compass* é uma ferramenta que facilita a visualização de dados da base de dados. Esta ferramenta foi usada para visualizar os fluxos e execuções persistidos no *MongoDB* com mais facilidade.

4.2.5 Swagger

O *Swagger* é uma ferramenta usada para definir API's e modelo de dados. Facilita na visualização e permite testar/usar a API.

5 Metodologias de Desenvolvimento de Software

A equipa na qual o autor esteve inserido tem implementada como metodologia de desenvolvimento de software o *Kanban* usando também alguns artefactos do *SCRUM*. O *Kanban* é uma metodologia ágil e tem como objetivo melhorar a entrega de tarefas e melhorar o fluxo de trabalho. Esta metodologia prioriza a produtividade e organização dos projetos. [9]

O *Kanban* utiliza cartões para indicar o progesso dos processos produtivos, eliminando qualquer tipo de atividade que não agregue valor à empresa. [10]

As tarefas são separadas em colunas diferentes, que são:

- *To Do* (para fazer): nesta coluna são colocadas as tarefas que não foram iniciadas; [10]
- *Doing* (em desenvolvimento): na segunda coluna encontram-se todas as tarefas que estão em desenvolvimento no momento; [10]
- *Done* (Terminado): na última coluna são colocadas as tarefas que foram concluídas. [10]

Apesar do Kanban não preconizar diferentes "*roles*", a equipa na qual o autor se insere faz uso de alguns artefactos do *SCRUM* e, por isso, existem três diferentes tipos de funções na equipa:

- **O *Líder da Equipa*:** É responsável por supervisionar o processo durante a *sprint*. Normalmente é o elemento com mais experiência na equipa;
- **O *Product Owner*:** Elemento responsável pelo planeamento, priorização de tarefas e comunicação com os clientes;
- **Os desenvolvedores:** Constituída pelos restantes elementos da equipa e responsável pela realização das tarefas.

A equipa também segue as reuniões preconizadas pelo *SCRUM*, que são:

- **As *daily*:** São curtas reuniões realizadas diariamente. Toda a equipa está presente e cada elemento tem de responder às questões: *O que eu fiz ontem? O que vou fazer hoje? Problemas encontrados?*
- **A reunião da retrospetiva:** Reuniões realizadas no final da sprint e toda a equipa está presente. Consiste em responder às questões: *O que correu bem? O que correu menos bem? O que podemos fazer de diferente?*
- **O planeamento:** Reuniões realizadas no início de cada *sprint*. O objetivo principal é definir o objetivo da *sprint* como também as tarefas que serão realizadas para alcançá-lo (*backlog*). Na reunião toda a equipa está presente;
- **A demo:** Esta reuniões ocorrem no final da *sprint* e consistem em explicar e demonstrar o que foi realizado na *sprint* anterior. Está presente toda a equipa e também todos os *stakeholders*.

Com esta metodologia, o trabalho é dividido em ciclos de desenvolvimento chamados *sprints*. As *sprints*, normalmente, tinham uma duração de duas semanas.

A Figura 9 é mostrado um exemplo de um *board Kanban* constituída por três colunas.

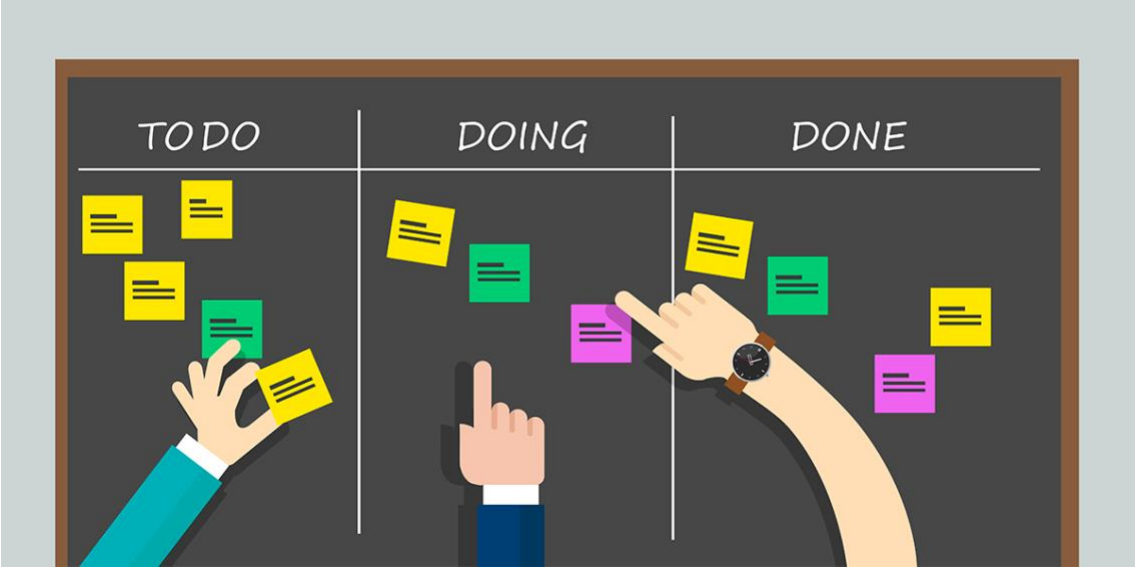


Figura 9 – Exemplo de uma board Kanban

Fonte: https://blog.acelerato.com/artigo/melhoria-continua-com-kanban/, 19/03/2023

A Altice Labs encontra-se, atualmente, a mudar o seu processo de desenvolvimento. Estão a ser migradas algumas ferramentas de trabalho, tais como a migração para *GitHub*, *Docker* e *Kubernetes*.

5.1.1 GitHub

Durante a realização do estágio a Altice Labs fez algumas alterações nas suas ferramentas de gestão de projeto. Começando a migrar do *SVN* e *Jira Confluence* para o *GitHub*. O GitHub é uma plataforma de repositório de código, na qual é possível fazer um controlo de versões, criar *issues* e *pull request* e também criar páginas *wiki* à semelhança do *Jira*. O GitHub tem também as *GitHub Actions* que é uma plataforma de integração contínua e entrega contínua e permite automatizar a *build*, testes e a *pipeline* de *deployment*. É possível criar *workflows* que fazem a *build* e executam os testes em cada *commit*/*pull request* criado [11].

5.1.2 Docker Images

Os RPMs (*Red-hat Package Manager*) estão a ser migrados para *docker images*. O RPM é um sistema de gestão de pacotes de software enquanto uma imagem *docker* é um ficheiro usado para executar código num *docker container*. O *docker* permite com mais facilidade configurar as imagens e os *containers*.

5.1.3 Kubernetes

A Altice Labs encontra-se também num processo de migração do *Ansible* para *Kubernetes*. Estas ferramentas são usadas para fazer o *deploy* das aplicações. Com esta migração começou-se a usar manifestos de instalação em *Kubernetes*. O *Kubernetes* tem a capacidade de fazer a orquestração de *docker containers* e fazer o seu escalonamento dependendo das suas necessidades.

5.1.4 Cloud Native

Cloud Native é uma abordagem usada para criar e instanciar aplicações na *cloud*. Esta abordagem explora a flexibilidade, a escalabilidade e resiliência da computação em *cloud*. Tem como vantagens ser *stateless*, escalável (up e down) e rápido no arranque das aplicações. Aplicações *stateless* consistem em aplicações que não guardam informações sobre determinadas interações que foram realizadas no passado. Isto reduz problemas de memória.

Para seguir esta abordagem é necessário adotar, por exemplo, containers e Kubernetes e acelera o desenvolvimento de aplicações.

6 Desenvolvimento do Motor de Execução de Fluxos

Neste capítulo são apresentados os detalhes mais técnicos e detalhados do desenvolvimento do motor de execução de fluxos. É abordado o modelo de dados, o swagger, a implementação do motor de execução de fluxos, testes realizados, e um POC final. Também são apresentadas as limitações do motor atual que levaram ao desenvolvimento do novo.

6.1 Limitações do motor atual

O conceito do motor atual, SWE, é executar o fluxo como um "todo" e não de atividade em atividade. O objetivo do novo motor é a atividade passar a ser a unidade básica de execução.

O SWE possui algumas limitações que dificultam a sua manutenção e o escalar do motor com novas funcionalidades. As limitações mais importantes são a incapacidade de executar atividades em paralelo, ou seja, possuir atividades *Fork* e *Join*.

Outra limitação reside na não existência de atividades do tipo sub-workflow, na qual permitiria executar um novo fluxo (filho) e aguardando que este seja terminado.

Apesar do motor atual ter a capacidade de reexecutar um fluxo a partir de uma certa atividade, a abordagem usada não é a melhor. Quando esta ação é requisitada, em vez de reexecutar o fluxo começando pela atividade que se pretende, está-se a reexecutar o fluxo desde o início percorrendo a *timeline* de atividades até chegar à atividade pretendida. A abordagem usada atualmente no SWE é complexa, mais demorada e mais exigente a nível de processamento.

6.2 Modelo de Dados

Numa fase inicial é necessário definir e criar o modelo de dados. O modelo de dados foi sofrendo alterações conforme as necessidades e as reuniões realizadas com toda a equipa.

O modelo de dados está dividido em três partes:

- Definição dos fluxos e suas atividades;
- Definição da execução dos fluxos;
- Definição do modelo de execução de ordens.

6.2.1 Definição dos Fluxos e suas Atividades

Para a definição dos fluxos e suas atividades são criadas classes para o seu efeito. As principais classes criadas são:

- ***WorkflowDefinition***;
- ***ActivityDefinition***;
- ***RetriesPolicy***.

A classe ***WorkflowDefinition*** é constituída por um conjunto de campos:

- ***Id***, que representa o identificador do fluxo. Os fluxos têm identificadores únicos;
- ***name***, que representa o nome do fluxo;
- ***version***, que representa a versão do fluxo;
- ***operationalState***, que indica se é um fluxo para desenvolvimento, teste ou produção. Este campo é um *enum*;
- ***startActivity***, que representa a primeira atividade a ser executada no fluxo;
- ***activities***, que representa um mapa de atividades;
- ***rollback***, que indica o identificador do fluxo a ser executado para repor todas as configurações realizadas até ao momento. Normalmente, este campo é usado quando a execução falha e termina em erro;
- ***Input***, que representa os valores de entrada do fluxo;
- ***Output***, que representa os valores de saída do fluxo, ou seja, os valores esperados quando o fluxo termina a sua execução.

A classe ***ActivityDefinition*** é constituída pelos seguintes campos:

- ***name***, que representa o nome da atividade. Todas as atividades definidas num fluxo têm de ter um nome único. Este campo é usado como um identificador;
- ***i18NCode***, que representa o identificador do país. Este campo é usado para realizar traduções;
- ***displayName***, que representa o nome da atividade a ser mostrado dependendo do código do país inserido no campo *i18NCode*;
- ***type***, que representa o tipo de atividade. Por exemplo: *FORK*, *JOIN*, *DECISION*, *SUB_WORKFLOW*, *HTTP*, entre outras;
- ***next***, que representa o nome da próxima atividade a ser executada;
- ***ttl***, que indica o tempo máximo que a atividade tem para terminar a sua execução. Se o tempo de execução não for cumprido a atividade é terminada em erro;
- ***retries***, representa um objeto do tipo "*RetryPolicies*" e é constituído pelo número máximo de *retries*, que uma atividade pode realizar, e pelo *delay*, ou seja, o "tempo de espera" antes da atividade voltar a ser executada;
- ***input***, que representa o input da atividade. Cada tipo de atividade possui um input diferente;
- ***output***, que representa o output da atividade. Cada tipo de atividade possui um output diferente.

A classe ***RetryPolicies*** representa a política de *retries* e é constituída pelos seguintes campos:

- ***max***, que representa o número máximo de reexecuções que a atividade pode fazer;
- ***delay***, que representa o tempo de espera antes de reexecutar a atividade.

A Figura 10 representa o diagrama de classes da definição de fluxos em formato *UML*.

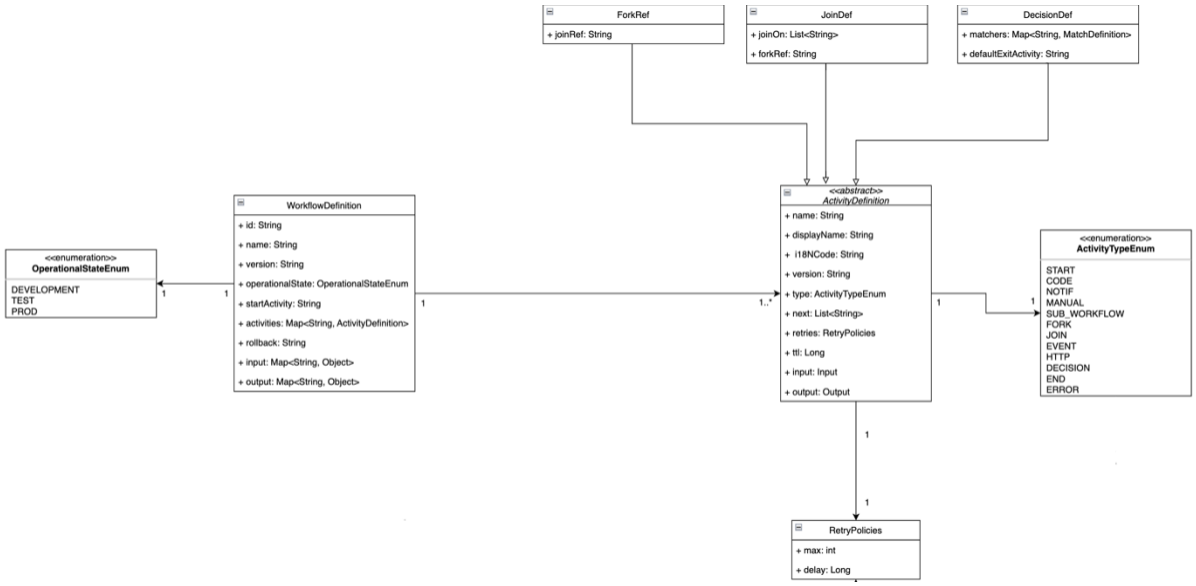


Figura 10 - Diagrama de classes referente ao WorkflowDefinition e ActivityDefinition

Fonte: Criado pelo autor

6.2.2 Definição da Execução de Fluxos

O modelo de dados responsável pela execução de um fluxo é diferente do da definição. Este é constituído por diferentes campos usados para acompanhar o estado de uma execução.

Para a execução dos fluxos e suas atividades são criadas classes para o seu efeito. As principais classes criadas são:

- ***WorkflowExecution***;
- ***AbstractActivity***;
- ***RetriesPolicy***.

A classe ***WorkflowExecution*** é constituída por um conjunto de campos:

- ***correlationId***, que representa o identificador externo, ou seja, o identificador da ordem;
- ***definitionId***, que indica o identificador do fluxo que é executado;
- ***executionId***, que representa o identificador da execução. As execuções têm identificadores únicos;
- ***Name***, que representa o nome da execução;
- ***version***, que indica a versão da execução;
- ***rollback***, à semelhança do *WorkflowDefinition*, este campo indica o identificador do fluxo a ser executado para repor todas as configurações realizadas até ao momento.
- ***startActivity***, que representa a primeira atividade a ser executada no fluxo;
- ***timeline***, que representa um mapa de todas as atividades que irão ser executadas;
- ***state***, que representa o estado atual da execução;

- ***createTime***, que representa o *timestamp* da criação da execução;
- ***startTime***, que representa o *timestamp* do momento do início da execução;
- ***stateTime***, que representa o *timestamp* do momento em que a execução mudou de estado;
- ***endTime***, que representa o *timestamp* do momento que a execução termina;
- ***Input***, que representa os valores de entrada do fluxo;
- ***Output***, que representa os valores de saída do fluxo, ou seja, os valores esperados quando o fluxo termina a sua execução.

A classe ***AbstractActivity*** é constituída pelos seguintes campos:

- ***name***, que representa o nome da atividade. Todas as atividades definidas num fluxo têm de ter um nome único. Este campo é usado como um identificador;
- ***displayName***, que representa o nome da atividade a ser mostrado dependendo do código do país inserido no campo *i18NCode*;
- ***type***, que representa o tipo de atividade. Por exemplo: *FORK*, *JOIN*, *DECISION*, *SUB_WORKFLOW*, *HTTP*, entre outros. Este campo é um *enum*;
- ***state***, representa o estado atual da atividade;
- ***next***, que representa o nome da próxima atividade a ser executada;
- ***startTime***, que representa o *timestamp* do momento do início da execução da atividade;
- ***stateTime***, que representa o *timestamp* do momento em que a atividade mudou de estado;
- ***endTime***, que representa o *timestamp* do momento que a execução da atividade termina;
- ***ttl,*** que indica o tempo máximo que a atividade possui para terminar a sua execução. Se a atividade não cumprir com o valor definido neste campo, a execução da mesma é terminada em erro;
- ***retries***, que representa um objeto do tipo "*RetryPolicies*" e é constituído pelo número de *retries* e pelo valor do *delay*;
- ***input***, que representa o input da atividade. Cada tipo de atividade possui um input diferente;
- ***output***, que representa o output da atividade. Cada tipo de atividade possui um output diferente.

A Figura 11 representa o diagrama de classes da execução de fluxos em formato *UML*.

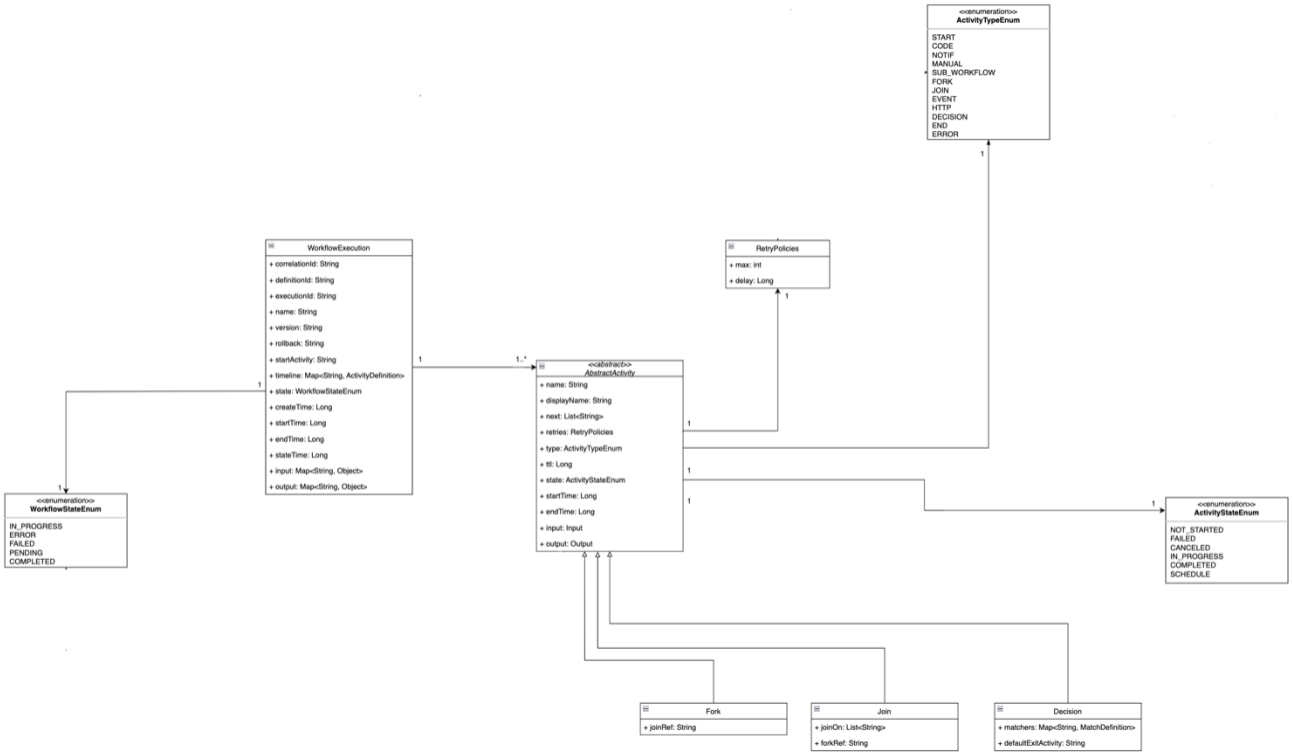


Figura 11 - Diagrama de classes referente ao WorkflowExecution e AbstractActivity

Fonte: Criado pelo autor

Antes de executar um fluxo, este precisa de ser criado e persistido na base de dados. No anexo C está representado o diagrama de classes completo.

6.2.3 Classe DTO

Uma classe *DTO* (*Data Transfer Object*) é um padrão usado para transportar dados entre diferentes sistemas. Este padrão consiste em agrupar um conjunto de atributos numa classe simples de forma a otimizar a comunicação.

Este padrão é usado na definição das execuções de um fluxo e permite ocultar propriedades confidenciais ou irrelevantes ao cliente. O uso destas classes tem também a vantagem de reduzir o tamanho do *payload* aquando da comunicação entre diferentes sistemas.

6.3 Execução

A execução de um fluxo é constituída por campos que são importantes no acompanhamento do estado da execução e também na realização de um *rollback* em caso de falha. De seguida são apresentados, com mais detalhe, cada um destes temas.

6.3.1 Estados da Execução

Uma execução é constituída por um campo *state* que pode ser de 11 diferentes tipos. Este estado representa a fase atual da execução. Os estados suportados são:

- *CREATED*, estado quando a ordem foi aceite e criada;
- *RUNNING*, estado usado quando a execução do fluxo é iniciada;
- *WAITING*, usado quando o fluxo não possui nenhuma atividade em execução, mas existe pelo menos uma atividade no estado *WAITING*;
- *CANCELING*, usado quando é realizado um pedido de cancelamento da execução, mas ainda existem atividades em execução. Nesta fase, a execução encontra-se em cancelamento;
- *CANCELLED*, estado aplicado quando a execução se encontra num estado *CANCELLING* e já não possui nenhuma atividade em execução;
- *PAUSING*, estado usado quando há um pedido para pausar o fluxo, mas ainda existem atividades em execução;
- *PAUSED*, estado aplicado quando a execução se encontra num estado *PAUSING* e já não existe nenhuma atividade em execução;
- *COMPLETED*, usado quando o fluxo termina com sucesso a sua execução;
- *FAILING*, estado usado quando uma atividade fica num estado *FAILED*, mas ainda existem outras atividades em execução;
- *FAILED*, estado aplicado quando a execução se encontra num estado *FAILING* e já não existe atividades em execução;
- *HALTED*, estado usado quando uma atividade fica num estado *HALTED* e as restantes estejam num estado diferente de *FAILED*.

A Figura 12 mostra o fluxo das transições entre estados de uma execução.

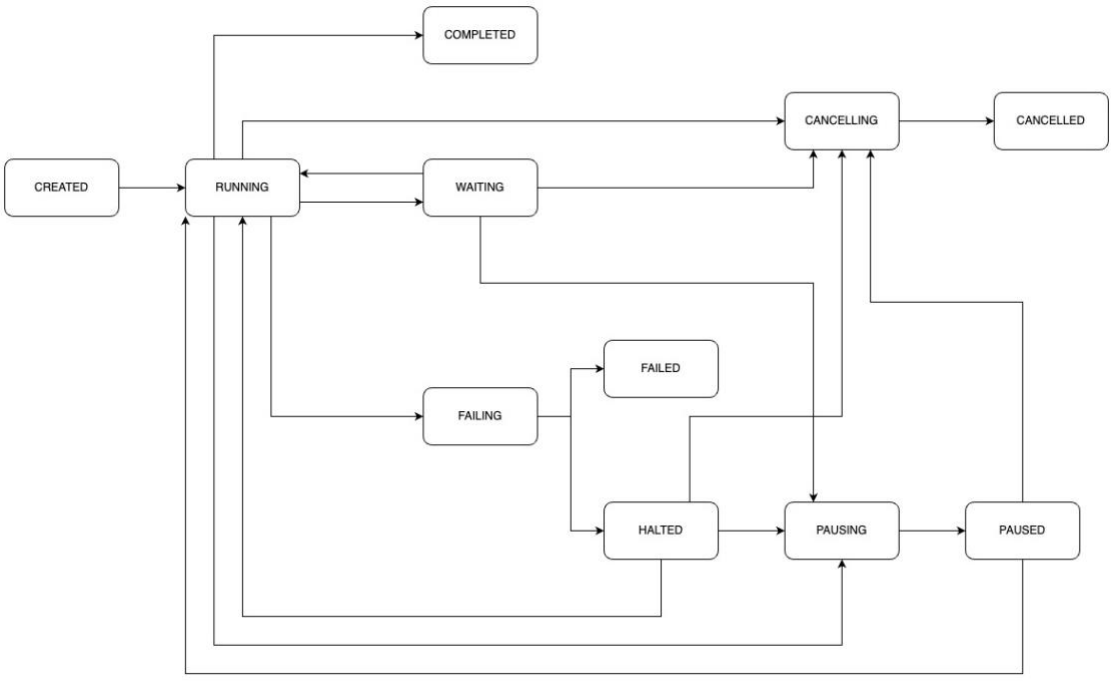


Figura 12 - Transição de estados de uma execução

Fonte: Criado pelo autor

Caso exista uma atividade em estado *FAILED* e outra em *HALTED*, o estado atribuído à execução é o *FAILED*.

6.3.2 *Rollback*

As execuções possuem um campo *rollback*. Este *rollback* é constituído pelo identificador do fluxo a ser executado. Quando uma execução tem uma falha e termina no estado *FAILED*, é executado o fluxo que está representado neste campo. Normalmente, este "*workflow*" *callback* tem como objetivo, por exemplo, reverter algumas configurações de dispositivos que tenham sido manipuladas antes da execução terminar em *FAILED*.

6.4 Atividades

Como já foi referido, as atividades podem ser diferentes e de diversos tipos. Considerando o tempo do estágio, começou-se por implementar as atividades mais básicas e importantes para a criação de um fluxo válido e funcional. Por isso, as atividades criadas durante o estágio foram o Start, Decision, HTTP, SubWorkflow, Fork, Join, End e Error.

Estas atividades possuem algumas particularidades importantes, como os diferentes estados e *allowed actions* que suportam, os seus inputs e o comportamento em caso de falha da atividade.

6.4.1 Estados das Atividades

Cada atividade é constituída por um estado que muda durante a sua execução e representa o estado atual da atividade. Considerando o modelo de negócio da Altice Labs, são definidos os 8 diferentes estados, que são:

- *NOT_STARTED*, quando a atividade ainda não foi executada;
- *RUNNING*, quando a atividade está em execução;
- *HALTED*, é um estado usado quando a atividade termina em erro, mas possui a *allowed action* que permite, por exemplo, reexecutar esta atividade.;
- *WAITING*, quando a atividade está à espera de ser notificada, por exemplo, quando está à espera de um evento externo;
- *SKIPPED*, quando o utilizador pretende que a atividade seja ignorada e por isso não é executada;
- *CANCELED*, é um estado final e indica que a atividade foi cancelada;
- *COMPLETED*, é um estado final que representa que a atividade terminou a sua execução corretamente sem dificuldades;
- *FAILED*, é um estado final que representa que a atividade terminou em erro.

A Figura 13 mostra o fluxo das transições entre estados de uma atividade.

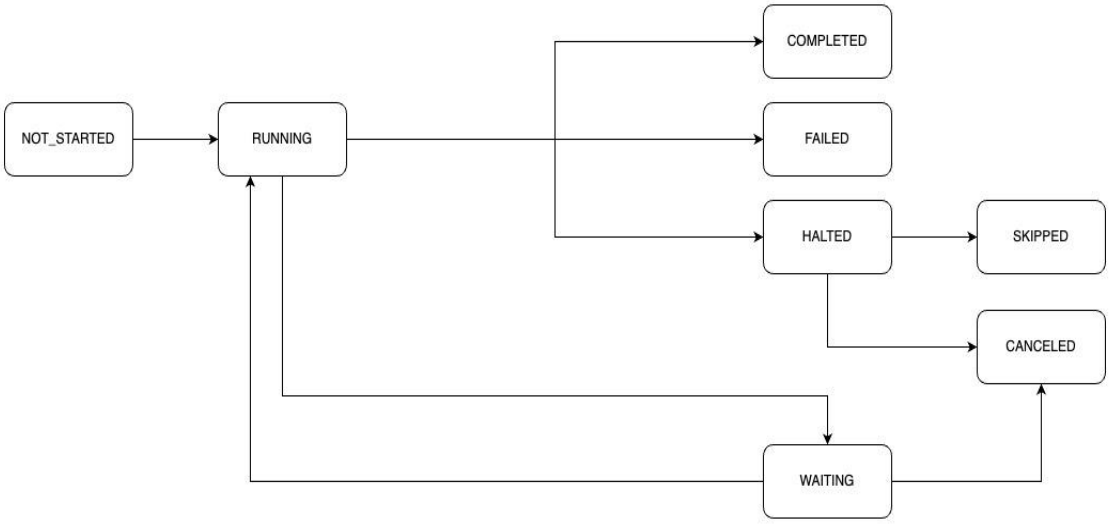


Figura 13 - Transição de estados de uma atividade

Fonte: Criado pelo autor

6.4.2 Allowed Actions

As *allowed action* são ações extra que as atividades podem ter. Estas ações podem ser de 3 diferentes tipos, que são:

- *Skip*, permite que a execução da atividade seja ignorada;
- *Restart*, permite que a atividade seja reexecutada. Esta allowed action é usada apenas quando a atividade está num estado "HALTED";
- *Cancel*, indica que a atividade pode ser cancelada. Após a atividade transitar para o estado "CANCELED", o estado da execução passa para o estado de "CANCELING" e só transita para "CANCELED" quando não existir nenhuma atividade em execução.

As *allowed actions* permitidas em cada atividade são definidas pelo utilizador que cria as atividades. Uma atividade pode ser constituída por nenhuma *allowed action*, apenas uma ou todas. Ou seja, permite ter todas as combinações possíveis. Por *default*, as atividades são constituídas por todas as allowed actions. Se o utilizador pretender retirar uma ação numa determinada atividade terá de preencher o campo "notAllowedAction". Este campo é uma lista de ações que a atividade não suporta. Este campo não se encontra definido no modelo de dados (secção 6.2.1) porque não foi implementado.

6.4.3 Inputs das Atividades

As atividades têm *inputs* que podem ser de dois diferentes tipos, um valor definido pelo utilizador ou um *json path* para encontrar o valor pretendido no *payload* da execução. Por isso, os campos de um *input* têm de seguir o seguinte *regex*: *$**type**(**value**)*. O type pode assumir valores como *String* ou *JsonPath*. *String* indica que o campo *value* é o valor introduzido pelo utilizador e usado sempre que este *input* for pretendido. O *JsonPath* indica que o que está presente no campo *value* é um *json path* e é necessário fazer a pesquisa no objeto da execução para obter o valor pretendido.

Esta abordagem também é usada nos *inputs* do fluxo.

6.4.4 Falha da Atividade

Durante a execução de uma atividade esta pode falhar. Sendo que os *retries* ainda não estão implementados, o comportamento assumido é o de terminar abruptamente a execução do fluxo.

As atividades do tipo *SubWorkflow* executam outro fluxo em paralelo. Se este falhar durante a sua execução, essa informação é propagada para o fluxo principal e o fluxo é terminado num estado de erro.

6.5 Swagger

Numa fase inicial realizou-se a criação do *swagger* em que foram definidos todos os *endpoints* da *API* e os modelos de dados da definição de um fluxo, da sua execução e um pedido de execução de um fluxo.

6.5.1 Endpoints

Os *endpoints* responsáveis pela definição de um fluxo são os seguintes:

- *GET* – getWorkflowById, usado para obter/pesquisar um fluxo pelo seu identificador;
- *POST* – createWorkflow, usado para criar um novo fluxo;
- *PUT* – updateWorkflow, usado para atualizar um fluxo já existente;
- *PATCH* – partialUpdateWorkflow, usado para fazer uma atualização parcial de um fluxo;
- *DELETE* – deleteWorkflow, usado para eliminar um fluxo através do seu identificador.

Os *endpoints* responsáveis pela execução de ordens e gestão de execução de fluxos são os seguintes:

- *POST – createExecution*, usado para receber pedidos de execução de fluxos;
- *GET – getExecutionById*, usado para obter o estado atual de uma execução de um fluxo;
- *DELETE – stopExecution*, usado para terminar abruptamente a execução de um fluxo que esteja em execução, através do seu identificador;
- *PUT – pauseExecution*, usado para pausar uma execução que esteja a decorrer através do identificador de execução;
- *PUT – resumeExecution*, usado para retomar uma execução de um fluxo que tenha sido pausado, através do seu identificador;
- *SKIP – skipActivity*, usado quando não é pretendido executar uma certa atividade, para isso é necessário indicar o identificador da execução em curso e o identificador da atividade que se pretende passar à frente.

Na Figura 14 está representado o conjunto destes *endpoints* no *swagger*.

Figura 14 - Representação dos endpoints da definição e execução de fluxos

Fonte: https://swagger.io/, criado pelo autor

6.7 Criação do motor de execução de fluxos

Inicialmente é criado as classes *POJO* que representam a definição de um fluxo, a sua execução e pedido de execução de ordem.

6.7.1 Ficheiro Configuração da Aplicação

O ficheiro de configuração "*application.properties*" é importante pois contém informação importante como o número de instâncias para cada *verticle*, o tipo de base de dados que se pretende usar (atualmente existe apenas suporte para MongoDB mas no futuro pode-se implementar outra base de dados como por exemplo *PostgreSQL*) e o nome das diferentes *collections* do mongo. Este ficheiro pode ser alterado pelos utilizadores dependendo das suas necessidades.

6.7.2 Verticles

Para o desenvolvimento da *API* e do motor de execução de fluxos foi usado *verticles* do *Vert.x*. Um *verticle* é um ator e cada *verticle* é constituído por pedaços de código responsável por cada funcionalidade. No fundo é uma unidade básica de execução.

A aplicação é repartida em 8 *verticles* que são:

- *Verticle* da API, responsável por receber todos os pedidos *http*. Os pedidos suportados encontram-se definidos na secção dos *endpoints* em 6.5.1;
- *Verticle* responsável por realizar operações de leitura na base de dados *MongoDB*;
- *Verticle* responsável por criar novos fluxos e execuções na base de dados;

- *Verticle* responsável por realizar operações de atualização na base de dados;
- *Verticle* para eliminar fluxos na base de dados;
- *Verticle* do *workflow*, responsável pela execução dos fluxos;
- *Verticle* da atividade, responsável pela execução de cada atividade;
- *Verticle* do *http*, este *verticle* é responsável pela realização de pedidos *http*.

Inicialmente, existia um *verticle* responsável por todas as operações à base de dados *MongoDB*. Por motivos de performance e maior controlo realizou-se a repartição de tarefas e atualmente cada *verticle* é responsável por um tipo de operação à base de dados.

Cada *verticle* pode ser instanciado diversas vezes dependendo da necessidade e da sobrecarga existente. Para isso, o *deploy* dos *verticles* é realizado na inicialização/arranque da aplicação. Relativamente ao número de instâncias que são usadas em cada *verticle*, estes valores são definidos no ficheiro "*application.properties*" e usados durante o *deploy* dos *verticles*.

Estes *verticles* comunicam entre si através do envio de mensagens no *event bus* do *Vert.x*, ou seja, os *verticles* enviam/consomem informação através dos endereços. Estes endereços são como canais. Quando um pedido é enviado para um determinado endereço, todos os *verticles* que estão à escuta, consomem o pedido.

Existe também a possibilidade de apenas uma instância de um *verticle* consumir a mensagem e para isso é considerado a primeira instância a fazê-lo. Normalmente a instância com mais disponibilidade é a que consome a mensagem do *event bus*. Esta é a abordagem usada porque apenas é pretendido realizar uma certa ação uma única vez, ou seja, consumir apenas uma vez.

É também possível fazer pedidos a endereços e esperar pela resposta de forma assíncrona, são os chamados *requests*. Depois do *verticle* realizar tudo o que pretende, pode fazer o *reply* para o *verticle* de origem (que fez o pedido).

Durante a comunicação entre os *verticles* existe a possibilidade de enviar uma mensagem. Esta mensagem pode ser uma simples *String* como pode ser um objeto composto.

A Figura 15 representa o diagrama de comunicação entre os diferentes *verticles*.

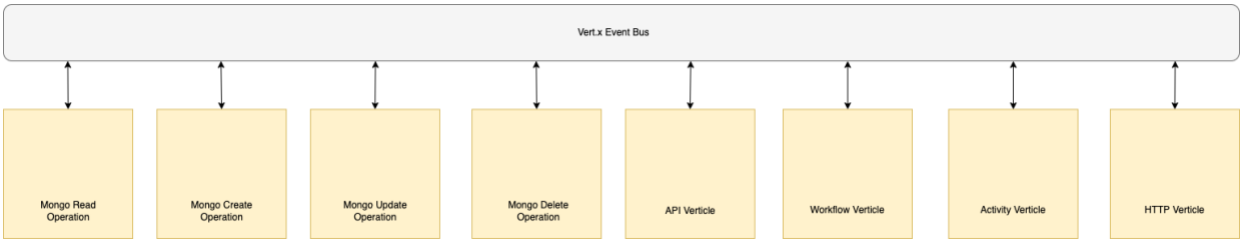


Figura 15 - Comunicação entre os Verticles

Fonte: Criado pelo autor

6.7.3 MongoDB

O *MongoDB* é a base de dados não relacional usada para persistir toda a informação relativa a fluxos e execuções.

Inicialmente, é criada uma base de dados e duas *collections*. Uma *collection* para a definição de fluxos e outra para a sua execução. As *collections* são semelhantes a tabelas em base de dados relacionais e são usadas para armazenar os documentos. Os documentos são os dados/informação que pretendemos armazenar, normalmente em formato *BSON Documents* [12].

6.8 Secured by Design

Foi implementada uma abordagem de *Secured by Design* que consiste em tornar os sistemas livres de vulnerabilidades e protegê-los contra eventuais ataques. Por questões de confidencialidade a Altice Labs não permite a partilha das técnicas e abordagens implementadas.

6.9 Testes Unitários e Integração

Os testes unitários e de integração são bastante importantes pois conseguimos garantir ou reduzir falhas que podem ocorrer no sistema. Caso algum teste falhe este precisa de ser revisto.

Os testes unitários são usados para testar pequenas partes/funcionalidades do sistema, principalmente funções. São testes responsáveis por testar a menor unidade do sistema.

Os testes de integração são utilizados para testar os diferentes módulos do sistema, ou seja, validar se os módulos interagem corretamente entre si.

Os testes unitários foram realizados apenas no final do estágio. A decisão tomada deve-se às alterações constantes do modelo de dados na fase inicial do projeto e verificou-se que, durante esta fase embrionária, não se justificava manter os testes pois estes iriam estar em constantes alterações.

Os testes de integração não foram implementados porque havia a necessidade de fazer o *deploy* da aplicação em *kubernetes* que também não foi realizado. A vantagem de correr os testes de integração num cluster *kubernetes* é a facilidade de validar todo o sistema facilmente, rápido e em *cloud native*. *Cloud native* é uma abordagem usada no desenvolvimento de software que permite criar e executar aplicações em nuvem. Toda a computação é feita em nuvem, por exemplo numa *cloud* publica, o que permite escalar a aplicação.

6.10 POC

Na fase final do estágio foi criado um *POC* para visualizar o comportamento da aplicação e apresentar a toda a equipa, equipa da solução/customização e chefias.

Este *POC* consiste na criação e execução de um pequeno fluxo constituído por todas as atividades suportadas até ao momento.

Na Figura 16 estão representados dois fluxos. O fluxo superior é o principal e o inferior é o fluxo "filho" usado na execução de uma atividade do tipo *SUB_WORKFLOW*.

O fluxo principal inicia com a atividade do tipo *START* e de seguida passa para uma atividade do tipo *DECISION*. Esta atividade de decisão vai validar alguns parâmetros que se encontram definidos no input do fluxo. Se a validação falhar é executada a atividade

de erro (*ERROR*) e o fluxo é terminado. Caso contrário, é executada a atividade seguinte que é do tipo *FORK*.

O *fork* tem como objetivo ordenar a execução de atividades em paralelo e neste exemplo é composto por dois ramos.

O ramo superior começa por executar uma atividade *http* (*HTTP*) que realiza um pedido *GET* a uma *api* pública. De seguida é executada uma atividade de decisão (*DECISION*) que valida o output da atividade anterior (*HTTP*). Neste exemplo, está a ser validado o "*status code*" da resposta do pedido *http*. Se o "*status code*" não pertencer à família de códigos de sucesso (entre 200 e 299) é executada a atividade de erro (*ERROR*) e o fluxo é terminado. Caso contrário, é executado a atividade seguinte que é do tipo *JOIN*.

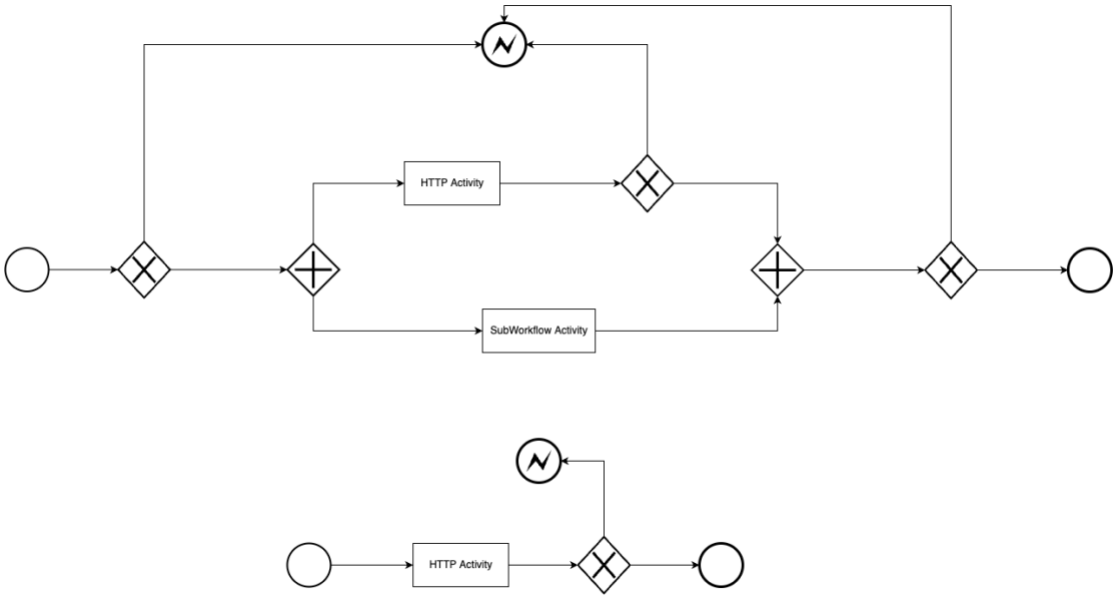


Figura 16 - Exemplo do fluxo utilizado no POC

Fonte: Criado pelo autor

No ramo inferior é executado a atividade do tipo *SUB-WORKFLOW*. Esta atividade executa um fluxo diferente, representado também na Figura 16 (fluxo inferior). Posteriormente, é executada a atividade do tipo JOIN.

Para a atividade do tipo JOIN ser executada é necessário que todos os ramos tenham terminado a sua execução.

Após a execução da atividade *join* é executada uma atividade de decisão (*DECISION*). Esta atividade vai fazer um conjunto de validações baseadas no output de atividades que já foram executadas. Se a validação falhar é executada a atividade de error (*ERROR*) caso contrário é executada a atividade seguinte que é do tipo *END* e corresponde a uma atividade final e é a última atividade a ser executada num fluxo.

O fluxo inferior inicia com uma atividade do tipo *START* e de seguida é executada uma atividade do tipo *HTTP*. Esta atividade faz um pedido *http* a uma *api* pública. Posteriormente, é executada uma atividade do tipo *DECISION* para realizar um conjunto de validações (o *status code* e a resposta devolvida no pedido). Esta atividade tem duas saídas diferentes e ambas correspondem a atividades finais, ou seja, uma atividade do tipo *ERROR* e outra do tipo *END*.

É importante relembrar que uma atividade só pode ser executada depois da atividade anterior terminar a sua execução.

O anexo D representa o pedido para a criação do fluxo principal e do fluxo "filho" (*sub-workflow*). O anexo E representa o pedido para executar o fluxo principal. O payload encontra-se em *json*.

7 Conclusão e Trabalho Futuro

Os principais objetivos do estágio foram concluídos com sucesso. Como o motor, *SWE*, utilizado à data tinha muitas limitações e problemas recorrentes foi necessário desenvolver um novo motor de execução de fluxos. Após a implementação deste novo motor é possível executar outras atividades que não eram suportadas pelo anterior motor de execução como por exemplo atividades relacionadas com *Sub-Workflow*, *Fork* e *Join*.

Não consideramos que o orquestrador ora desenvolvido já esteja maduro o suficiente para ser utilizado em produção, mas encontra-se funcional e com uma base sólida de implementação com as principais atividades primárias já desenvolvidas. Após mais alguns testes e aperfeiçoamentos poderá ir para produção em clientes selecionados e após a consideração de eventuais feedbacks recebidos nesta fase poder-se-á obter uma versão inicial totalmente funcional e pronta para uma utilização plena.

No subcapítulo 7.1, são detalhadas as principais limitações e dificuldades encontradas durante o estágio como também a forma como foram ultrapassadas. No subcapítulo 7.2, é apresentado o trabalho futuro a ser realizado para enriquecer o novo motor de execução de fluxos.

Considerando os objetivos iniciais previstos para este estágio, pode-se referir que:

- **Estado da arte:** Realizou-se um estudo comparativo baseado em dois motores de execução de fluxos *open-source*: o *Netflix Conductor* e o *Uber Cadence*. Consideraram-se estes dois motores para este estudo, pois são dos mais utilizados e eram os que a priori pareciam ser os mais adequados ao cumprimento das necessidades que a Altice Labs tinha. Chegou-se à conclusão de que o *Netflix Conductor* era o motor mais interessante e versátil para a Altice Labs, mas mesmo assim tinha algumas limitações, como referido em 2.6. A análise e as conclusões deste estudo estão plasmadas no relatório desenvolvido para análise por parte da Altice Labs e que se apresenta no Anexo B.
- **Definição do modelo de dados:** Foi criado um modelo de dados consentâneo com as necessidades decorrentes do desenvolvimento do novo motor de execução de fluxos. O seu desenvolvimento teve origem numa consideração inicial de um modelo de dados base o qual foi depois e ao longo do estágio completado de acordo com considerações e novas funcionalidades que foram tidas em conta ao longo do desenvolvimento do estágio. Este modelo de dados foi definido no *swagger* conforme já atrás referido na secção 5.4.1.
- **Definição das *APIs* para a manipulação dos dados considerados no modelo de dados desenvolvido e para a execução de fluxos:** O cumprimento deste objetivo foi realizado em duas fases: a primeira consistiu na definição das *APIs* para as operações mais básicas, como por exemplo criação/alteração/eliminação/execução de fluxos, utilizando-se para tal o *swagger* e o seu consequente desenvolvimento. Numa segunda fase, foram desenvolvidas outras *APIs* como por exemplo as necessárias para as operações de pausar, retomar e cancelar a execução de fluxos. Foi também desenvolvida a *API* para o *skip* de atividades em fluxos que estejam em execução.

- **Implementação das atividades:** Nesta fase utilizaram-se diversas tecnologias tais como *Quarkus*, *Vert.x* e *Mutiny* para a criação de atividades necessárias para o funcionamento do motor de execução. A par da implementação do próprio motor de execução, esta fase foi uma das de maior duração e em simultâneo uma das mais desafiantes e motivantes.
- **Desenvolvimento do motor de execução de atividades:** Como atrás se referiu, esta etapa foi realizada em simultâneo com a da implementação das atividades e teve como objetivo desenvolver o motor de execução de fluxos e das atividades com ele relacionadas.

Deste modo e cumpridos todos estes objetivos inicialmente definidos, concluímos que o motor que foi desenvolvido vai conseguir corresponder às necessidades da Altice Labs tendo em conta as novas capacidades e funcionalidades implementadas, ao contrário do que não acontecia com o anterior motor *SWE*. Por fim, vai ser possível aos utilizadores deste novo motor criarem novos e diferentes fluxos e de uma maneira mais flexível e otimizada facilitando assim o seu trabalho.

7.1 Limitações e Dificuldades

Existiram algumas dificuldades durante o estágio, principalmente na fase inicial do desenvolvimento do motor, nomeadamente a utilização da *framework Quarkus*, *Vert.x* e *Mutiny*. Estas limitações deveram-se ao desconhecimento destas ferramentas que são muito recentes no mercado. Para colmatar estas dificuldades foram realizados cursos online e leitura de documentação das respetivas tecnologias. O estagiário realizou um curso online sobre *Quarkus* que permitiu uma aprendizagem mais uniforme e acelerada. Para consolidar os conhecimentos que o estagiário já tinha sobre *MongoDB*, foi também executado um estudo complementar sobre esta temática.

Para aperfeiçoar e melhorar os conhecimentos já adquiridos durante a sua formação académica sobre a linguagem de programação Java, o estagiário também realizou estudos com esse objetivo. Para isso, houve o apoio de toda a equipa que esteve sempre disponível para o ajudar.

7.2 Trabalho Futuro

Apesar da base do motor já estar desenvolvida existe algum trabalho que é necessário realizar. Por exemplo, é necessário proceder a uma revisão cuidada do código que foi criado de modo a enriquecê-lo, torná-lo mais estável e otimizado. Com esse objetivo e já com a possibilidade de fazer o *deployment* da aplicação em *Kubernetes*, perspetiva-se a criação dos testes de integração.

Também deverão ser implementadas funcionalidades que possibilitem proceder a realização de notificações. Estas funcionalidade ficaria associada às atividades de notificação. Consideramos que esta implementação será importante para notificar sistemas externos.

Perspetivamos também novas atividades para aumentar as capacidades inicialmente previstas para o motor de execução de fluxos. As atividades do tipo *Event*, *Code* e *Manual* deveriam ser futuramente também implementadas pois parece-nos que as mesmas enriqueceriam substancialmente o motor ora desenvolvido. O mesmo se

considera para a implementação de *timeouts*, *retries* e *rollbacks* nas atividades o que poderia tornar o motor mais funcional e autónomo.

Referências Online

[1] Wikipédia, “Altice Portugal,” [Online]. Available: https://pt.wikipedia.org/wiki/Altice_Portugal. [Acedido em 4 08 2022].

[2] ISEC, “Factos e Números,” [Online]. Available: https://www.isec.pt/pt/instituto/#lnkFactosNumeros. [Acedido em 4 08 2022].

[3] Inova-Ria, “A Associação,” [Online]. Available: https://www.inova-ria.pt/associacao. [Acedido em 9 08 2022].

[4] “Notação BPMN, a mais usada para modelar processos,” [Online]. Available: https://www.heflo.com/pt-br/bpm/notacao-bpmn/. [Acedido em 07 03 2022].

[5] [Online]. Available: https://www.infoq.com/br/articles/getting-started-with-quarkus/. [Acedido em 26 03 2022].

[6] Vert.x, "Vert.x Core Manual," [Online]. Available: https://vertx.io/docs/vertx-core/java/. [Accessed 10 08 2022].

[7] Wikipédia, "Vert.x," [Online]. Available: https://en.wikipedia.org/wiki/Vert.x. [Accessed 10 08 2022].

[8] R. Fraga, “Quarkus: Entendendo a relação entre o Mutiny e o Vert.x,” [Online]. Available: https://dev.to/renatasfraga/quarkus-entendendo-a-relacao-entre-o-mutiny-e-o-vert-x-1e3o. [Acedido em 10 08 2022].

[9] P. H. ESCOBAR, “Egestor,” 17 01 2021. [Online]. Available: https://blog.egestor.com.br/kanban/. [Acedido em 19 03 2023].

[10] G. Santos, “Automação Industrial,” 02 03 2022. [Online]. Available: https://www.automacaoindustrial.info/kanban/. [Acedido em 19 03 2023].

[11] Github, "Understanding GitHub Actions," [Online]. Available: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. [Accessed 11 08 2022].

[12] MongoDB, "Databases and Collections," [Online]. Available: https://www.mongodb.com/docs/manual/core/databases-and-collections/ . [Accessed 2022 Agosto 3].

[13] ISEC, “Mensagem do Presidente da República,” [Online]. Available: https://www.isec.pt/pt/Destaque.aspx?ID=3773. [Acedido em 4 08 2022].

Anexo A – Proposta de Estágio

PROPOSTA DE ESTÁGIO

Ano Letivo de 2020/2021

Mestrado em Engenharia Informática

(Engenharia de Software)

TEMA

Service Workflow Engine - Cloud Evolution

SUMÁRIO

Um ou dois parágrafos com uma breve descrição do estágio proposto.

1. Âmbito

A AlticeLabs possui uma gama de produtos na área de Fulfillment que assumem um papel preponderante na orquestração de processos de provisão e diagnóstico de clientes dos Operadores de Telecomunicações. O Service Workflow Engine é um dos componentes que suporta esses produtos.

Responsável pela orquestração dos processos de execução, é, possivelmente a peça mais importante na execução de ordens de provisão e diagnóstico. Contudo, as suas funções podem ser alargadas ao âmbito de execuções de fluxos de configuração de elementos de rede. Actualmente encontra-se em fase de reformulação e *cloudificação*.

2. Objetivos

O presente estágio pretende atingir os seguintes objetivos genéricos:

- Avaliar os orquestradores de fluxos existentes na comunidade *open source*;
- Enumerar vantagens e desvantagens dos orquestradores existentes;
- Implementar uma nova ferramenta de orquestração de fluxos considerando os seguintes aspectos:
  1. Recorrer à *framework Quarkus* como suporte ao desenvolvimento;
  2. Seguir metodologia *Secure by design;*
  3. Seguir a abordagem *Cloud native;*

3. Programa de trabalhos

O estágio consistirá nas seguintes atividades e respetivas tarefas:

- ***T1*** – *Estado da Arte* – Avaliação das ferramentas existentes na comunidade *open source*.
- ***T2*** – *Modelo Base* – Definição do modelo de dados de suporte aos workflows e do modelo operacional.
- ***T3*** – *APIs* - Definição das APIs de CRUD e de execução dos workflows, seguindo a metodologia *Secure by design*.
- ***T4*** - *Atividades* - Criação das atividades que a ferramenta oferece como unidades base de execução.
- ***T5*** - *Motor de Execução* - Desenvolvimento do motor de execução de actividades.
- ***T6*** - *Testes de Integração* - Montar cenário de testes de integração integrado na cloud.

4. Calendarização DAS TAREFAS

As Tarefas acima descritas, incluindo os testes de validação de cada módulo, serão executadas de acordo com a seguinte calendarização:

O plano de escalonamento dos trabalhos é apresentado em seguida:

	Meses										
Tarefas	1	2	3	4	5	6	7	8	9	10	11
T1											
T2											
T3											
T4											
T5											
T6											
Metas	INI		M1		M2	M3	M4			M5	M6

INI Início dos trabalhos
M1 Tarefa T1 terminada
M2 Tarefa T2 terminada
M3 Tarefa T3 terminada
M4 Tarefa T4 terminada
M5 Tarefa T5 terminada
M6 Tarefa T6 terminada

5. Resultados

Os resultados do estágio serão consubstanciados num conjunto de documentos, a elaborar pelo estagiário de acordo com o seguinte plano:

- Documentação das APIs, em formato *swagger*.

- Modelos de dados em formato UML.
- Documentação de configuração da ferramenta, em formato MD.
- Tese final.

6. Local DE TRABALHO

Regime misto.

7. Metodologia

A metodologia de trabalho segue as orientações Agile.

8. Orientação

Entidade de Acolhimento: Altice Labs

Carlos Manuel Rodrigues (carlos-m-rodrigues@alticelabs.com)
Licenciado em Engenharia de Sistemas e Informática
Gilberto Pereira Valente de Matos (gilberto-p-matos@alticelabs.com)
Licenciado em Engenharia de Computadores e Telemática

ISEC:

Doutor Jorge Barbosa (jorbar@isec.pt)
Categoria Professor Coordenador

9. Caracterização e Remuneração

- Data de início: 1/09/2021
- Data de fim: 31/07/2022
- Horário: Full-time
- Outras condições:

Anexo B – Comparação entre motores de execução de fluxos

Workflow engine state of the art

Introdução

Um motor de execução de fluxos é uma aplicação de software que gere processos de negócio. Estes motores gerem e monitorizam o estado das tarefas /atividades num workflow. As tarefas/atividades podem desempenhar qualquer ação. Um motor de fluxos facilita o fluxo de informação, tarefas/atividades e eventos.

O Netflix *Conductor* e o Uber *Cadence* são motores de execução de fluxos e a principal diferença está na definição dos *workflows* e das tarefas /atividades. No *Conductor* os fluxos e tarefas são definidos através de JSON enquanto no *Cadence* os fluxos e atividades são definidos em código (Java ou Go).

Nas secções apresentadas a seguir é feita uma análise mais profunda sobre cada uma das tecnologias, a sua comparação, um exemplo prático da sua utilização e por fim uma conclusão.

Netflix Conductor

O Netflix Conductor é um orquestrador de microserviços usado e desenvolvido pela Netlfix. Este motor executa um conjunto de processos de negócio que são acompanhados por uma orquestração assíncrona de tarefas executadas em microserviços. Alguns destes processos poderão ser de longa duração e durar vários dias.

Os workflows são definidos usando JSON e são constituidos por tarefas que representam o que se quer executar. Existem duas categorias de tarefas diferentes: as *system task* e as *worker tasks*.

As *system tasks* são executadas pelo Conductor e podem ser de diversos tipos:

- *SWITCH*, semelhante a um switch-case;
- *DECISION*, semelhante a uma condição *if*;
- *EVENT*, com capacidade para publicar eventos no Conductor ou num sistema de eventos externos (exemplo SQS);
- *HTTP*, usado para comunicar com outros microserviços através de HTTP;
- SUB WORKFLOW, usado para criar um workflow “filho” dentro do workflow principal;
- *FORK_JOIN*, usado para executar tarefas em simultâneo/paralelo;
- *JOIN*, usado para esperar que uma ou mais tarefas terminem a execução. Estas tarefas têm de estar num *fork*;
- *WAIT*, é uma tarefa que fica com status “IN_PROGRESS” até que seja atribuída com status “*COMPLETED*” ou “*FAILED*” por um *trigger* externo;
- *Terminate*, é uma tarefa que termina a execução do fluxo. O fluxo é terminado com um status e com um *output*. Estes parâmetros são definidos pelo cliente.
- *DO_WHILE*, é usado para executar um conjunto de tarefas enquanto que a condição é verdadeira;
- *SET_VARIABLE*, é uma tarefa usada para alterar o valor de variáveis no *workflow*.

As *worker tasks* são implementadas pela aplicação e executadas num ambiente em separado do *Conductor*. As *worker tasks* podem ser implementadas em qualquer linguagem e comunicam com o servidor *Conductor* através de *REST/gRPC* para obter as tarefas a executar e retornar o estado atualizado do fluxo. Estas tarefas são do tipo “*SIMPLE*”.

Conductor Client

As tarefas que são executadas por *workers* comunicam através de *endpoints HTTP* para fazer o *poll* das tarefas e atualizar o estado da execução.

O Conductor fornece bibliotecas em Java que são usadas para interagir com as APIs podendo ser usadas para registar/atualizar definições de tarefas e workflows, iniciar execução de fluxos e, como foi dito em cima, para fazer o poll de tarefas e atualizar o resultado da mesma.

Uber Cadence

O Uber Cadence também é um orquestrador de microserviços usado e desenvolvido pela Uber.

A definição dos workflows e atividades é feita através de código, para tal é preciso usar a framework da Uber Cadence para realizar esta operação. Atualmente a *framework* é apenas compatível em *Java* e *Go*.

Para definir um *workflow* é necessário criar uma *interface* em que os seus métodos necessitam de conter uma das seguintes anotações:

- *WorkflowMethod*, indica o método do ponto de entrada de um *workflow*. É constituído por parâmetros como *timeouts*;
- *SignalMethod*, indica o método que reage a sinais externos;
- *QueryMetod*, indica o método que reage a pedidos de *query* síncronos. Usado para retornar valores de variáveis do *workflow*.

Para definir atividades é necessário criar uma interface em que os metodos contenham a anotação “*ActivityMethod*”. Cada um destes métodos define uma atividade e cada *workflow* pode usar mais que uma interface de atividades.

Comparação entre os Motores

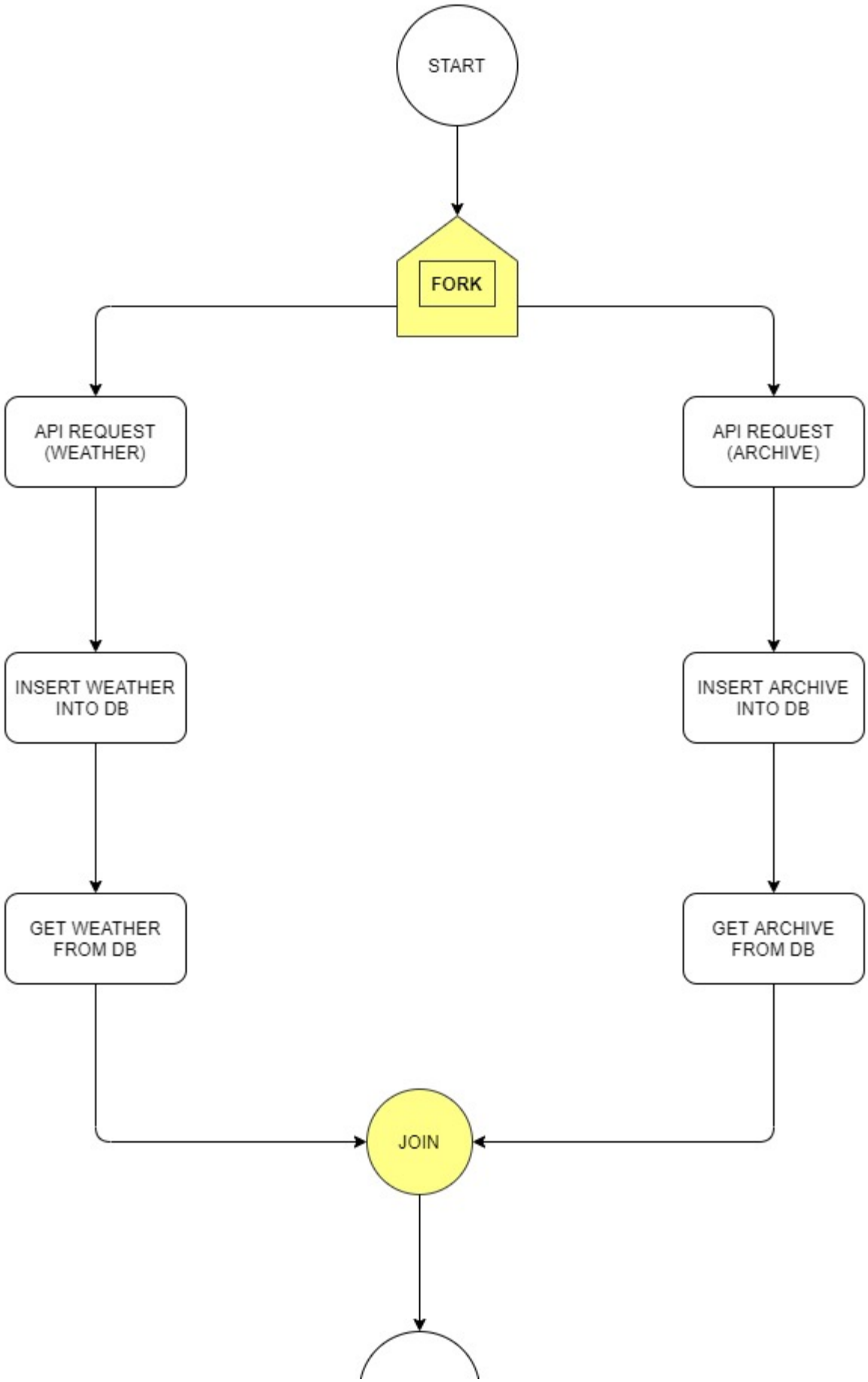
Na tabela seguinte é apresentada uma comparação entre os dois motores, sintetizando os pontos fortes e os pontos fracos de cada uma.

	Netflix Conductor	Uber Cadence
Identificadores do Workflow	Os workflows são constituídos por um identificador único. Quando um fluxo inicia a execução o Conductor gera id's para cada uma das tarefas. Estes id's das tarefas são usados quando se pretende reexecutar um workflow a partir de uma certa tarefa ou fazer o *skip* "dela".	Os *workflows* são constituídos por dois identificadores: *workflow id* e *run id*. O *workflow id* é o identificador do *workflow* e só pode existir um *workflow* com o mesmo id em execução ao mesmo tempo. Se houver uma intenção de executar um *workflow* com um id igual a um *workflow* que esteja em execução, o que estiver a ser executado é terminado e o "novo" é executado. O run id é o identificador de execução. Quando existe um *workflow* que terminou a execução e de seguida pretende-se reiniciá-lo, o novo *workflow* irá ter o mesmo *workflow id,* mas é gerado um novo *run id*, ou seja, pode-se executar um *workflow* já executado (com o mesmo id) mas o *run id* vai ser diferente.
Sub Workflow	Quando um *sub workflow* é executado, o *workflow* "pai" fica à espera que o filho termine. É um fluxo sequencial.	Quando um *sub workflow* é executado, o *workflow* "pai" pode continuar a executar outras atividades.
Cancelar /Terminar Workflow	Quando o motor recebe instrução para cancelar a execução do fluxo, ele é terminado de imediato. Caso exista uma tarefa em execução quando for realizado um pedido de cancelamento, o fluxo é dado como terminado e a resposta da tarefa é "descartada" pelo motor.	Quando o motor recebe instrução para cancelar a execução do fluxo este vai fazer um pedido de cancelamento da atividade que está a ser executada. Como o fluxo só termina quando a execução da atividade terminar, a atividade tem de usar um mecanismo de *heartbeat* para verificar o estado do fluxo, ou seja, tem de usar este mecanismo de x em x tempo e verificar quando é "gerada" uma exceção para terminar a atividade. *Heartbeat* – é um pedido que é feito ao fluxo para avisar que a atividade ainda está em execução e também para guardar o progresso da atividade. Neste caso este mecanismo foi usado para detetar falhas com um erro específico indicando que a atividade foi cancelada. Se não existir nenhuma atividade em execução, o fluxo é terminado imediatamente.
Paralelismo	Possibilidade ter duas ou mais atividades em execução ao mesmo tempo e com a possibilidade de continuar a executar atividades sequencialmente. Possibilidade de esperar que as atividades em paralelo terminem.	
Reexecutar Workflow	Existem três mecanismos para reiniciar um fluxo: • Reiniciar um fluxo que já esteja terminado. Necessário indicar id do *workflow*; • Reiniciar fluxo a partir da última tarefa que falhou. Este mecanismo utiliza os mesmos valores de entrada do fluxo e da tarefa que falhou. Necessário indicar id do *workflow;* • Reiniciar fluxo a partir da tarefa que pretendemos. É possível introduzir dados de entrada do fluxo e da tarefa onde vai começar. É necessário indicar o id do *workflow* e o id da tarefa. Não é possível reiniciar um fluxo (com um certo id) se este ainda estiver a ser executado. É possível reiniciar um *workflow* (com um certo id) se este estiver terminado. A nova execução irá ter o mesmo *workflow* id.	Existem duas possibilidades de reiniciar o fluxo: • Reiniciar a partir do início do fluxo usando os mesmos parâmetros de entrada. Necessário indicar id do *workflow*; • Reiniciar a partir da última atividade que falhou usando os mesmos parâmetros de entrada do fluxo e da atividade. Necessário indicar id do *workflow*. Não há a possibilidade de reiniciar o fluxo com dados de entrada diferentes dos definidos inicialmente. Se uma atividade tem o parâmetro "número máximo de *retries"* definido e a atividade ultrapassa esse número de tentativas, o fluxo é imediatamente terminado com erro. De seguida, se reiniciarmos o fluxo a partir da última atividade que falhou vai gerar novamente o mesmo erro. É possível concluir que o contador do número de vezes que a atividade foi executada não fez *reset* e, por isso, sempre que for pretendido reiniciar o fluxo a partir da última atividade vai dar o mesmo erro. Se for reiniciado um fluxo (com certo id) e este ainda estiver a ser executado, o fluxo atual é terminado e o novo será iniciado.
Pausar /Continuar Workflow	Existência de dois comandos para pausar e retomar a execução de um *workflow*.	A única possibilidade para pausar/retomar *workflow* é através do envio de sinais. Por exemplo, definir um booleano para controlar o fluxo.
Wait	Existe uma tarefa do tipo Wait que fica com status IN_PROGRESS à espera que seja terminada por um evento externo.	Utilizando o método await da biblioteca do Cadence é possível esperar até que o parâmetro seja "verdadeiro" (vá ao encontro do requerido). Para esse parâmetro sofrer alteração e o fluxo continuar a execução são usados sinais.
***Skip* da Tarefa /Atividade**	Existência de comando que permite "saltar" a tarefa, ou seja, a tarefa não é executada. Se a tarefa for "descartada" (*skipped*) enquanto a tarefa estiver em execução, o fluxo salta para a próxima tarefa e "descarta" os resultados retornados pela tarefa anterior. Necessário indicar o nome da referência da tarefa que se pretende saltar.	Possibilidade de usar um sinal para não executar uma certa atividade. Não é "prático" porque teria de existir um parâmetro para cada atividade. E, caso seja enviado um sinal para saltar uma determinada atividade e essa atividade estiver em execução esta ação não terá efeito. Ou seja, o fluxo conclui que a atividade foi executada.

Falha no Worker	Quando um *worker* falha enquanto executa uma tarefa/atividade o fluxo pode mandar executar novamente essa tarefa/atividade. Para isso, a tarefa/atividade tem de ter definido o parâmetro “número máximo de *retries*” (maior que zero) e o tempo máximo de execução de uma tarefa/atividade.	
Eventos/Sinais	Capacidade de criar evento num sistema externo ou interno (Conductor). Os eventos suportados são: • começar um workflow; • terminar uma tarefa com status “Completed”; • terminar uma tarefa com status “Failed”; Mecanismo de comunicação usado é o HTTP.	Possibilidade de “enviar” sinais para um workflow com o objetivo de alterar valores de parâmetros ou terminar o fluxo. Mecanismo de comunicação usado é o HTTP.
Modificar variável do Workflow	Existência de uma tarefa do tipo “SET_VARIABLE” que permite criar, inicializar e modificar variáveis no *workflow*. Esta tarefa tem de ser definida no fluxo. Durante a execução do fluxo não é possível criar /alterar variáveis por sistemas externos.	Através de sinais é possível, apenas, modificar variáveis do *workflow*. As variáveis são alteradas durante a execução do fluxo através do envio de sinais.
Timeouts	Possibilidade de definir vários *timeouts*, que são: • tempo máximo para a execução de um fluxo (antes de ser marcado como “*TIMED_OUT*”); • tempo máximo para a execução de uma tarefa/atividade fluxo (antes de ser marcada como “*TIMED_OUT*”); • número máximo de *retries*; • tempo de espera antes do novo *retry*.	

Exemplo Prático

Nesta secção é apresentado um exemplo prático da implementação de tarefas/atividades e *workflow*. A figura seguinte representa um fluxo constituído por diversas tarefas/atividades.


START
FORK
API REQUEST (WEATHER)
API REQUEST (ARCHIVE)
INSERT WEATHER INTO DB
INSERT ARCHIVE INTO DB
GET WEATHER FROM DB
GET ARCHIVE FROM DB
JOIN

Este fluxo tem como objetico executar tarefas/atividades em paralelo. Começando por realizar em simultâneo dois *API request* para obter dados sobre previsões meteorológicas e de um ficheiro. De seguida essa informação é colocada numa base de dados e por fim esses mesmos dados são recuperados da bd. A execução das tarefas sobre as previsões meteorológicas e sobre o ficheiro é realizado em paralelo mas a execução das tarefas de cada uma das partes é feito de forma sequencial, ou seja, primeiro é executado o *API request*, depois é inserido os dados na base de dados e por fim os dados são recuperados. O *JOIN* fica à espera que as tarefas *GET WEATHER FROM DB* e *GET ARCHIVE FROM DB* terminem.

De seguida vai ser apresentado um exemplo de como definir tarefas/atividades e *workflows* no *Conductor* e no *Cadence*. Vai ser usado, como exemplo, o "desenho" do fluxo apresentado em cima.

Exemplo usando Netflix Conductor

Nesta secção vai ser apresentado a definição de tarefas e *workflows* usando o motor da Netflix.

Definição de Tarefas

No exemplo seguinte é apresentado um conjunto de três tarefas responsáveis por fazer um *API request,* colocar e recuperar dados numa base de dados. Cada tarefa é constituida por vários parâmetros que são usados para definir o nome da tarefa, *timeouts*, parâmetros de entrada e de saída, número de *retr ies* entre outros.

Tasks


```
[
   {
      "name":"api_task",
      "retryCount":1,
      "timeoutSeconds":1200,
      "pollTimeoutSeconds":3600,
      "inputKeys":[
         "url"
      ],
      "outputKeys":[
         "result"
      ],
      "timeoutPolicy":"TIME_OUT_WF",
      "retryLogic":"FIXED",
      "retryDelaySeconds":600,
      "responseTimeoutSeconds":1100,
      "concurrentExecLimit":100,
      "rateLimitFrequencyInSeconds":60,
      "rateLimitPerFrequency":50,
      "ownerEmail":"example@example.com"
   },
   {
      "name":"insert_into_db_task",
      "retryCount":1,
      "timeoutSeconds":1200,
      "pollTimeoutSeconds":3600,
      "inputKeys":[
         "collection",
         "data"
      ],
      "outputKeys":[
         "result"
      ],
      "timeoutPolicy":"TIME_OUT_WF",
      "retryLogic":"FIXED",
      "retryDelaySeconds":600,
      "responseTimeoutSeconds":1100,
      "concurrentExecLimit":100,
      "rateLimitFrequencyInSeconds":60,
      "rateLimitPerFrequency":50,
      "ownerEmail":"example@example.com"
   },
   {
      "name":"get_from_db_task",
      "retryCount":1,
      "timeoutSeconds":1200,
      "pollTimeoutSeconds":3600,
      "inputKeys":[
         "collection"
      ],
      "outputKeys":[
         "result"
      ],
      "timeoutPolicy":"TIME_OUT_WF",
      "retryLogic":"FIXED",
      "retryDelaySeconds":600,
      "responseTimeoutSeconds":1100,
      "concurrentExecLimit":100,
      "rateLimitFrequencyInSeconds":60,
      "rateLimitPerFrequency":50,
      "ownerEmail":"example@example.com"
   }
]
```


Definição de Workflow

O exemplo seguinte representa uma definição de um *workflow* que é constituido pelas tarefas apresentadas acima. Este *workflow* é constituido por tarefas do tipo *SIMPLE* e também por *system tasks*, FORK_JOIN e *JOIN*. Cada workflow é constituido por vários parâmetros que são usados para definir o nome do fluxo, a sua descrição, a versão e o conjunto das tarefas. Para introduzir as tarefas no fluxo é necessário definir alguns parâmetros como o nome da tarefa (em tarefas do tipo *SIMPLE* o nome tem de ser o mesmo que o usado na definição da tarefa), o nome da referência da tarefa que é um "identificador" único e é usado quando é pretendido referirmos a essa tarefa (pode ser usado para obter o output de uma tarefa). Também é necessário introduzir o tipo da tarefa e os valores de *input* da tarefa.

Workflow

```
{
   "name":"AsyncActivities",
   "description":"handle weather and archive",
   "version":1,
   "ownerEmail":"example@example.com",
   "tasks":[
      {
         "name":"fork_join",
         "taskReferenceName":"forkx",
         "type":"FORK_JOIN",
         "forkTasks":[
            [
               {
                  "name":"api_task",
                  "taskReferenceName":"weather",
                  "inputParameters":{
                     "url":"https://www.7timer.info/bin/astro.php?lon=113.2&lat=23.
1&ac=0&unit=metric&output=json&tzshift=0"
                  },
                  "type":"SIMPLE"
               },
               {
                  "name":"insert_into_db_task",
                  "taskReferenceName":"insert_weather_into_db",
                  "inputParameters":{
                     "collection":"weather",
                     "data":"${weather.output.result}"
                  },
                  "type":"SIMPLE"
               },
               {
                  "name":"get_from_db_task",
                  "taskReferenceName":"get_weather_from_db",
                  "inputParameters":{
                     "collection":"weather"
                  },
                  "type":"SIMPLE"
               }
            ],
            [
               {
                  "name":"api_task",
                  "taskReferenceName":"archive",
                  "inputParameters":{
                     "url":"https://archive.org/advancedsearch.php?q=subject:google+sheets&output=json"
                  },
                  "type":"SIMPLE"
               },
               {
                  "name":"insert_into_db_task",
                  "taskReferenceName":"insert_archive_into_db",
                  "inputParameters":{
                     "collection":"archive",
                     "data":"${archive.output.result}"
                  },
                  "type":"SIMPLE"
               },
               {
                  "name":"get_from_db_task",
                  "taskReferenceName":"get_archive_from_db",
                  "inputParameters":{
```

```
                  "collection":"archive"
               },
               "type":"SIMPLE"
            }
         ]
      ]
   },
   {
      "name":"join",
      "taskReferenceName":"join2",
      "type":"JOIN",
      "joinOn":[
         "get_weather_from_db",
         "get_archive_from_db"
      ]
   },
   {
      "name":"weather_switch_task",
      "taskReferenceName":"is_weather_inserted",
      "inputParameters":{
         "case_value_param":"${insert_weather_into_db.output.result}"
      },
      "type":"DECISION",
      "caseValueParam":"case_value_param",
      "decisionCases":{
         "true":[
            {
               "name":"archive_switch_task",
               "taskReferenceName":"is_archive_inserted",
               "inputParameters":{
                  "case_value_param":"${insert_archive_into_db.output.result}"
               }
            }
         ]
      }
   }
]
}
```

Implementação de Worker

Para fazer o *poll* das tarefas a executar é necessário criar *workers* para executar as tarefas que se encontram no *workflow*. Para tal, é preciso criar uma classe que implemente a *interface Worker*. O método *execute(Task task)* é invocado quando existe uma tarefa que tem de ser executada por esse *worker* e retorna um *TaskResult* com a resposta da execução. Aqui pode-se fazer uso do parâmetro *taskReferenceName* do *workflow* para comparar o tipo de ação que é pretendido executar com base no nome da referência da tarefa.

Para invocar um worker é necessário ter um método *main* que é executado em *background* e está "à escuta" de tarefas. É possível ter múltiplas instâncias de um *worker* para responder a diferentes tarefas.

```
public class StarterWorker implements Worker {
    private final String taskDefName;

    public StarterWorker(String taskDefName) {
        this.taskDefName = taskDefName;
    }

    @Override
    public String getTaskDefName() {
        return taskDefName;
    }

    @Override
    public TaskResult execute(Task task) {
        String taskName = task.getTaskDefName();
        Activities activity = new ActivityImpl();
        TaskResult result = new TaskResult(task);
        result.setStatus(TaskResult.Status.COMPLETED);

        String data;
        switch (taskName) {
            case "api_task":
                data = activity.getDataFromAPI(task.getInputData().get("url").toString());
                result.getOutputData().put("result", data);
                break;
            case "insert_into_db_task":
                Boolean isInserted = activity.insertDataToDB(task.getInputData().get("collection").toString(),
task.getInputData().get("data").toString());
                result.getOutputData().put("result", isInserted.toString());
                break;
            case "get_from_db_task":
                data = activity.getDataFromDB(task.getInputData().get("collection").toString());
                result.getOutputData().put("result", data);
                break;
            default:
                result.setStatus(TaskResult.Status.FAILED);
                break;
        }
        return result;
    }
}

public class Main {
    public static void main(String[] args) {
        TaskClient taskClient = new TaskClient();
        taskClient.setRootURI("http://localhost:8080/api/");        //Point this to the server API

        int threadCount = 6;            //number of threads used to execute workers.  To avoid starvation,
should be same or more than number of workers

        Worker worker1 = new StarterWorker("api_task");
        Worker worker2 = new StarterWorker("insert_into_db_task");
        Worker worker3 = new StarterWorker("get_from_db_task");

        // Create TaskRunnerConfigurer
        TaskRunnerConfigurer configurer = new TaskRunnerConfigurer.Builder(taskClient, Arrays.asList(worker1,
worker2, worker3))
                .withThreadCount(threadCount)
                .build();

        // Start the polling and execution of tasks
        configurer.init();
    }
}
```

Exemplo usando Uber Cadence

Nesta secção vai ser apresentado a definição de atividades e *workflows* usando o motor da Uber.

Definição do Workflow

Inicialmente é necessário criar um *worker* para fazer o *poll* das atividades. De seguida são registadas no *worker* as classes do *workflow* e das atividades.

Main

```
public class Main{
    static final String TASK_LIST = "HelloActivity";

    public static void main(String[] args) {
        WorkflowClient workflowClient =
                WorkflowClient.newInstance(
                        new WorkflowServiceTChannel(ClientOptions.defaultInstance()),
                        WorkflowClientOptions.newBuilder().setDomain(DOMAIN).build());
        // Get worker to poll the task list.
        WorkerFactory factory = WorkerFactory.newInstance(workflowClient);
        Worker worker = factory.newWorker(TASK_LIST);


        // Workflows are stateful. So you need a type to create instances.
        worker.registerWorkflowImplementationTypes(WorkflowWithAsyncActivitiesImpl.class);
        // Activities are stateless and thread safe. So a shared instance is used.
        worker.registerActivitiesImplementations(new ActivityImpl());
        // Start listening to the workflow and activity task lists.
        factory.start();
    }
}
```

Depois é definido o *workflow* como é apresentado no código seguinte. Na interface do workflow é possível introduzir o *timeout* para a duração máxima, em segundos, do fluxo através do parâmetro *executionStartToCloseTimeoutSeconds* na anotação *WorkflowMethod*.

Workflow

```
public interface IWorkflow {
    @WorkflowMethod(executionStartToCloseTimeoutSeconds = 240, taskList = TASK_LIST)
    String startWorkflow();

}


public class WorkflowWithAsyncActivitiesImpl implements IWorkflow{
    private final Activities activities = Workflow.newActivityStub(Activities.class);

    private final String weatherURL = "https://www.7timer.info/bin/astro.php?lon=113.2&lat=23.
1&ac=0&unit=metric&output=json&tzshift=0";
    private final String archiveURL = "https://archive.org/advancedsearch.php?q=subject:
google+sheets&output=json";

    @Override
    public String startWorkflow() {
        Promise<String> weatherResult = Async.function(
                () -> {
                    String weatherData = activities.getDataFromAPI(weatherURL);
                    boolean inserted = activities.insertDataToDB("weather", weatherData);
                    return activities.getDataFromDB("weather");
                });

        Promise<String> archiveResult = Async.function(
                () -> {
                    String archiveData = activities.getDataFromAPI(archiveURL);
                    boolean inserted = activities.insertDataToDB("archive", archiveData);
                    return activities.getDataFromDB("archive");
                });

        String weather = weatherResult.get();
        String archive = archiveResult.get();

        return "Finished!";
    }
}
```

Definição das Atividades

Nesta secção é apresentado um conjunto de três atividades responsáveis por fazer um *API request,* colocar e recuperar dados numa base de dados. Na interface da atividade é possível introduzir os *timeouts* para a duração máxima de cada atividade, o número de retries e o intervalo entre a falha de uma atividade e da nova tentativa.

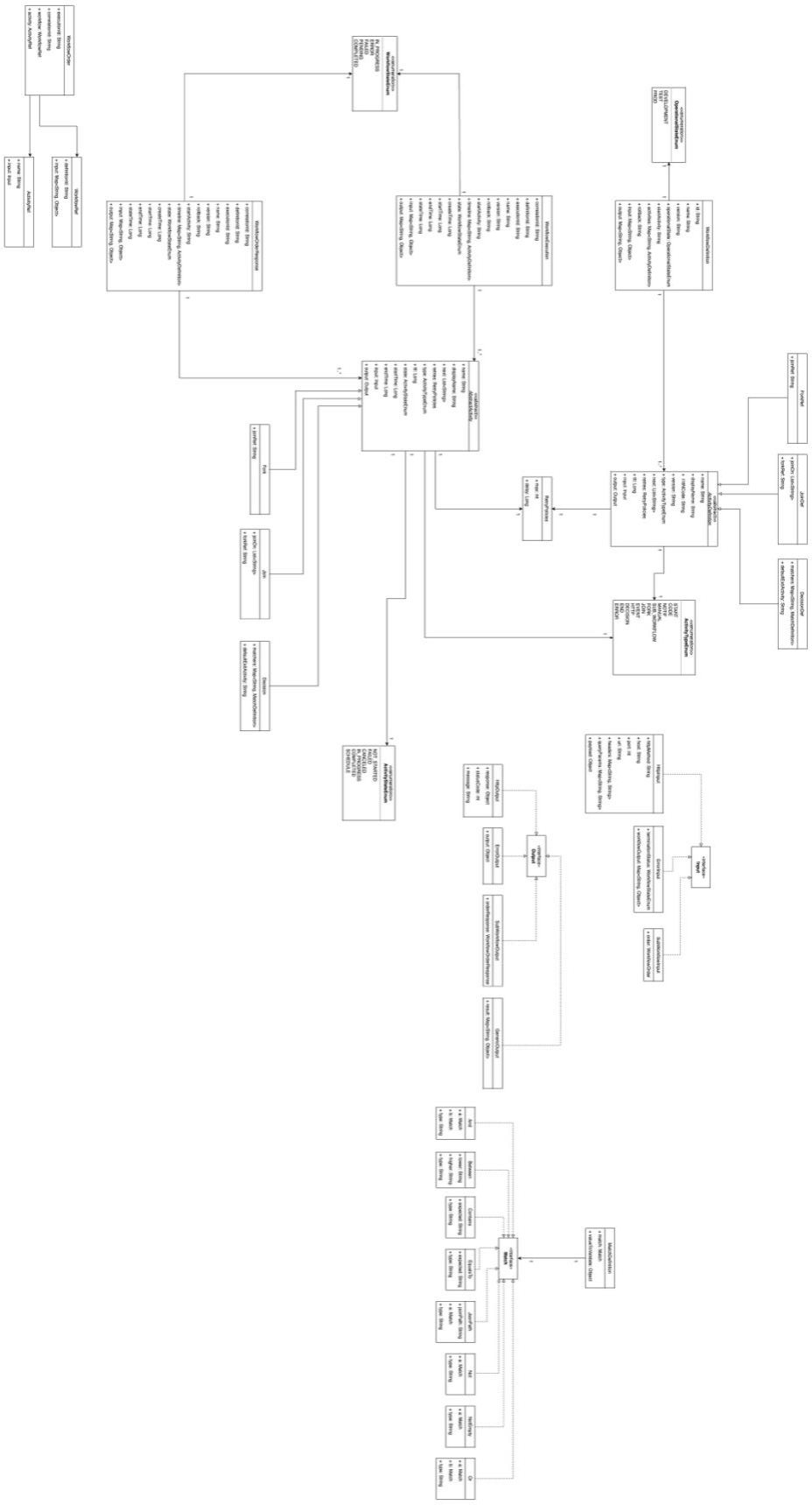
Activities

```
public interface Activities {
    @ActivityMethod(scheduleToCloseTimeoutSeconds = 60)
         @MethodRetry(backoffCoefficient = 1, maximumAttempts = 2, initialIntervalSeconds = 5)
    String getDataFromAPI(String url);

    @ActivityMethod(scheduleToCloseTimeoutSeconds = 60)
         @MethodRetry(backoffCoefficient = 1, maximumAttempts = 2, initialIntervalSeconds = 5)
    Boolean insertDataToDB(String collection, String data);

    @ActivityMethod(scheduleToCloseTimeoutSeconds = 60)
         @MethodRetry(backoffCoefficient = 1, maximumAttempts = 2, initialIntervalSeconds = 5)
    String getDataFromDB (String collection);
}


public class ActivityImpl implements Activities{

    @Override
    public String getDataFromAPI(String url) {

        try {
            HttpClient client = HttpClient.newHttpClient();
            HttpRequest request = HttpRequest.newBuilder()
                    .GET()
                    .header("accept", "application/json")
                    .uri(URI.create(url))
                    .build();

            HttpResponse<String> response = null;
            try {
                response = client.send(request, HttpResponse.BodyHandlers.ofString());
                System.out.println("MESMO ANTES DE RETURN");
                return response.body();
            } catch (InterruptedException e) {
                e.printStackTrace();
                return null;
            }

        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }

    public Boolean insertDataToDB(String collectionName, String data) {
        MongoClient mongoClient = MongoConnection.getInstance().getMongoClient();
        MongoDatabase db = mongoClient.getDatabase("test");
        Document document = Document.parse(data);
        db.createCollection(collectionName);
        db.getCollection(collectionName).insertOne(document);
        return true;
    }

    public String getDataFromDB(String collectionName) {
        MongoClient mongoClient = MongoConnection.getInstance().getMongoClient();
        MongoDatabase db = mongoClient.getDatabase("test");
        MongoCollection collection = db.getCollection(collectionName);
        Document document = (Document) collection.find().first();
        System.out.println(document.toJson());
        return document.toJson();
    }
}
```

Conclusão

Depois da análise realizada entre as duas tecnologias verificou-se que existem alguns defeitos entre elas. Esta análise e comparação entre os motores é importante para perceber o comportamento e como estão implementadas cada uma das suas funcionalidades. Serve também para recolher ideias para o desenvolvimento de um novo motor.

Anexo C – Diagrama de classes completo



Anexo D – Exemplos de payload para criação de fluxos

Fluxo Principal


```
{
	"name": "Workflow Test :: Start -> First_Decision -> Fork -> ( HTTP -> DecisionAfterHTTP | Second_Decision ) -> Join -> Third_Decision -> End",
	"type": "string",
	"version": "1.0.0",
	"operationalState": "DEVELOPMENT",
	"startActivity": "Start",
	"rollback": {
		"id": "6eb7d58e-f06f-4c3b-90cd-7c5c3e4bb6d2"
	},
	"activities": {
		"Start": {
			"name": "Start",
			"version": "1.0.0",
			"type": "START",
			"next": [
				"FirstDecisionAtivity"
			],
			"retries": {
				"max": 3,
				"delay": 100
			},
			"ttl": 150,
			"input": {
				"name": {
					"type": "vString",
					"value": "NONE"
				}
			}
		},
		"FirstDecisionAtivity": {
			"name": "FirstDecisionAtivity",
			"version": "1.0.0",
			"type": "DECISION",
			"next": [
				"Fork"
			],
			"retries": {
				"max": null,
				"delay": 100
			},
			"ttl": 150,
```

```
                "input": {
                        "defaultExitActivity": "Fork",
                        "matchDecision": {
                                "Error": [
                                        {
                                                "valueToValidate": {
                                                        "type": "jsonPathObject",
                                                        "path": "$"
                                                },
                                                "match": {
                                                        "type": "jsonpath",
                                                        "jsonpath":
"$.input.first_decision_input",
                                                        "a": {
                                                                "type": "equalsto",
                                                                "expected": "Hello World"
                                                        }
                                                }
                                        }
                                ]
                        }
                }
        },
        "Error": {
                "name": "Error",
                "version": "1.0.0",
                "type": "ERROR",
                "next": null,
                "retries": {
                        "max": 3,
                        "delay": 100
                },
                "ttl": 150,
                "input": {
                        "terminationStatus": "FAILED",
                        "workflowOutput": {
                                "random": {
                                        "type": "vString",
                                        "value": "random"
                                },
                                "randomCity": {
                                        "type": "vString",
                                        "value": "random city"
                                }
                        }
                }
        },
        "Fork": {
                "name": "Fork",
                "version": "1.0.0",
```

```
            "type": "FORK",
            "next": [
                "HTTP",
                "SubWorkflowActivity"
            ],
            "retries": {
                "max": 3,
                "delay": 100
            },
            "ttl": 150,
            "joinRef": "Join",
            "input": {
                "name": {
                    "type": "vString",
                    "value": "NONE"
                }
            }
        },
        "HTTP": {
            "type": "HTTP",
            "name": "HTTP",
            "version": "1.0.0",
            "next": [
                "DecisionAfterHTTPAtivity"
            ],
            "retries": {
                "max": 3,
                "delay": 100
            },
            "ttl": 150,
            "input": {
                "host": {
                    "type": "vString",
                    "value": "api.agify.io"
                },
                "uri": {
                    "type": "vString",
                    "value": "/?name=James"
                },
                "httpMethod": {
                    "type": "vString",
                    "value": "GET"
                },
                "headers": {
                    "content-type": "application/x-www-form-urlencoded"
                }
            }
        },
        "SubWorkflowActivity": {
            "name": "SubWorkflowActivity",
```

```
                "version": "1.0.0",
                "type": "SUB_WORKFLOW",
                "next": [
                        "Join"
                ],
                "retries": {
                        "max": null,
                        "delay": 100
                },
                "ttl": 150,
                "input": {
                        "order": {
                                "workflowRef": {
                                        "id": "58b01e57-a548-4d1b-9730-
ed75430c0ec4"
                                },
                                "activityRef": {
                                        "name": "Start",
                                        "type": "START",
                                        "input": {
                                                "name": {
                                                        "type": "vString",
                                                        "value": "NONE"
                                                }
                                        }
                                }
                        }
                }
        },
        "DecisionAfterHTTPAtivity": {
                "name": "DecisionAfterHTTPAtivity",
                "version": "1.0.0",
                "type": "DECISION",
                "next": [
                        "Join"
                ],
                "retries": {
                        "max": null,
                        "delay": 100
                },
                "ttl": 150,
                "input": {
                        "defaultExitActivity": "Join",
                        "matchDecision": {
                                "Error": [
                                        {
                                                "valueToValidate": {
                                                        "type": "jsonPathObject",
                                                        "path": "$"
                                                },
```

```
                                        "match": {
                                                "type": "jsonpath",
                                                "jsonpath":
"$.timeline.HTTP.output.statusCode",
                                                "a": {
                                                        "type": "equalsto",
                                                        "expected": "200"
                                                }
                                        }
                                }
                        ]
                }
        }
    },
    "Join": {
        "name": "Join",
        "version": "1.0.0",
        "type": "JOIN",
        "next": [
            "ThirdDecisionAtivity"
        ],
        "retries": {
            "max": 3,
            "delay": 100
        },
        "ttl": 150,
        "joinOn": [
            "DecisionAfterHTTPAtivity",
            "SubWorkflowActivity"
        ],
        "forkRef": "Fork",
        "input": {
            "name": {
                "type": "vString",
                "value": "NONE"
            }
        }
    },
    "ThirdDecisionAtivity": {
        "name": "ThirdDecisionAtivity",
        "version": "1.0.0",
        "type": "DECISION",
        "next": [
            "End"
        ],
        "retries": {
            "max": null,
            "delay": 100
        },
        "ttl": 150,
```

```
                "input": {
                        "defaultExitActivity": "End",
                        "matchDecision": {
                                "Error": [
                                        {
                                                "valueToValidate": {
                                                        "type": "jsonPathObject",
                                                        "path":
"$.timeline.HTTP.output.response"
                                                },
                                                "match": {
                                                        "type": "and",
                                                        "a": {
                                                                "type": "and",
                                                                "a": {
                                                                        "type": "jsonpath",
                                                                        "jsonpath": "$.name",
                                                                        "a": {
                                                                                "type": "equalsto",
                                                                                "expected": "James"
                                                                        }
                                                                },
                                                                "b": {
                                                                        "type": "jsonpath",
                                                                        "jsonpath": "$.age",
                                                                        "a": {
                                                                                "type": "equalsto",
                                                                                "expected": "59"
                                                                        }
                                                                }
                                                        },
                                                        "b": {
                                                                "type": "jsonpath",
                                                                "jsonpath": "$.count",
                                                                "a": {
                                                                        "type": "equalsto",
                                                                        "expected": "111683"
                                                                }
                                                        }
                                                }
                                        }
                                ]
                        }
                }
        },
        "End": {
                "name": "End",
                "version": "1.0.0",
                "type": "END",
                "next": null,
```

```
                "retries": {
                        "max": 3,
                        "delay": 100
                },
                "ttl": 150,
                "input": {
                        "name": {
                                "type": "vString",
                                "value": "NONE"
                        }
                }
        }
    },
    "output": {
        "httpResponse": {
                "type": "jsonPathField",
                "path": "$.timeline.HTTP.output.response"
        }
    },
    "input": {
        "first_decision_input": "Hello World",
        "http-configuration1": {
                "port": 8080,
                "server": "altice"
        }
    }
}
```

Fluxo do SubWorkflow


```
{
	"name": "Sub-Workflow Test :: Start -> First_Decision -> End",
	"type": "string",
	"version": "1.0.0",
	"operationalState": "DEVELOPMENT",
	"startActivity": "Start",
	"rollback": {
		"id": "6eb7d58e-f06f-4c3b-90cd-7c5c3e4bb6d2"
	},
	"activities": {
		"Start": {
			"name": "Start",
			"version": "1.0.0",
			"type": "START",
			"next": [
				"HTTP"
			],
			"retries": {
				"max": 3,
				"delay": 100
			},
			"ttl": 150,
			"input": {
				"name": {
					"type": "vString",
					"value": "NONE"
				}
			}
		},
		"HTTP": {
			"type": "HTTP",
			"name": "HTTP",
			"version": "1.0.0",
			"next": [
				"FirstDecisionAtivity"
			],
			"retries": {
				"max": 3,
				"delay": 100
			},
			"ttl": 150,
			"input": {
				"host": {
					"type": "vString",
					"value": "api.agify.io"
				},
```

```
                        "uri": {
                                "type": "vString",
                                "value": "/?name=Antonio"
                        },
                        "httpMethod": {
                                "type": "vString",
                                "value": "GET"
                        },
                        "headers": {
                                "content-type": "application/x-www-form-urlencoded"
                        }
                }
        },
        "FirstDecisionAtivity": {
                "name": "FirstDecisionAtivity",
                "version": "1.0.0",
                "type": "DECISION",
                "next": [
                        "End"
                ],
                "retries": {
                        "max": null,
                        "delay": 100
                },
                "ttl": 150,
                "input": {
                        "defaultExitActivity": "End",
                        "matchDecision": {
                                "Error": [
                                        {
                                                "valueToValidate": {
                                                        "type": "jsonPathObject",
                                                        "path": "$"
                                                },
                                                "match": {
                                                        "type": "jsonpath",
                                                        "jsonpath":
"$.timeline.HTTP.output.statusCode",
                                                        "a": {
                                                                "type": "equalsto",
                                                                "expected": "200"
                                                        }
                                                }
                                        }
                                ],
                                "PreRerun": [
                                        {
                                                "valueToValidate": {
                                                        "type": "jsonPathObject",
                                                        "path": "$"
```

```
                                        },
                                        "match": {
                                                "type": "jsonpath",
                                                "jsonpath":
"$.timeline.HTTP.output.response.name",
                                                "a": {
                                                        "type": "equalsto",
                                                        "expected": "James"
                                                }
                                        }
                                }
                        ]
                }
        }
},
"PreRerun": {
        "name": "PreRerun",
        "version": "1.0.0",
        "type": "PRE_RERUN",
        "next": [
                "HTTP"
        ],
        "retries": {
                "max": 3,
                "delay": 100
        },
        "ttl": 150,
        "input": {
                "type": "HTTP",
                "executionId": {
                        "type": "jsonPathField",
                        "path": "$.id"
                },
                "activityInput": {
                        "host": {
                                "type": "vString",
                                "value": "api.agify.io"
                        },
                        "uri": {
                                "type": "vString",
                                "value": "/?name=James"
                        },
                        "httpMethod": {
                                "type": "vString",
                                "value": "GET"
                        },
                        "headers": {
                                "content-type": "application/x-www-form-
urlencoded"
                        }
```

```
                }
            }
        },
        "End": {
            "name": "End",
            "version": "1.0.0",
            "type": "END",
            "next": null,
            "retries": {
                "max": 3,
                "delay": 100
            },
            "ttl": 150,
            "input": {
                "name": {
                    "type": "vString",
                    "value": "NONE"
                }
            }
        },
        "Error": {
            "name": "Error",
            "version": "1.0.0",
            "type": "ERROR",
            "next": null,
            "retries": {
                "max": 3,
                "delay": 100
            },
            "ttl": 150,
            "input": {
                "terminationStatus": "FAILED",
                "workflowOutput": {
                    "random": {
                        "type": "vString",
                        "value": "random"
                    },
                    "randomCity": {
                        "type": "vString",
                        "value": "random city"
                    }
                }
            }
        }
    },
    "output": {
        "httpResponse": {
            "type": "vString",
            "value": "Sub-Workflow terminated"
        }
```

```
        },
        "input": {
                "first_decision_input": "Hello SubWorkflow",
                "http-configuration1": {
                        "port": 8080,
                        "server": "altice"
                }
        }
}
```

Anexo E – Exemplo de payload para realizar pedido de execução de um fluxo

```
{
	"workflowRef": {
		"id": "56bebfda-777d-471d-b034-5ce305672d28",
		"input": {
			"first_decision_input": "Hello World",
			"http-configuration1": {
				"port": 8080,
				"server": "altice"
			}
		}
	},
	"activityRef": {
		"name": "Start",
		"type": "START",
		"input": {
			"name": {
				"type": "vString",
				"value": "NONE"
			}
		}
	}
}
```

Instituto Superior
de Engenharia
Politécnico de Coimbra