SERVIDOR ORQUESTADOR DE PROPÓSITO GENERAL

GENERAL-PURPOSE ORCHESTRATOR SERVER

TRABAJO FIN DE MÁSTER
CURSO 2020-2021

AUTOR
SERGIO LUIS PARA

DIRECTOR
ADRIÁN RIESCO RODRÍGUEZ

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

SERVIDOR ORQUESTADOR DE PROPÓSITO GENERAL

GENERAL-PURPOSE ORCHESTRATOR SERVER

TRABAJO DE FIN DE MÁSTER EN INGENIERÍA INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

AUTOR
SERGIO LUIS PARA

DIRECTOR
ADRIÁN RIESCO RODRÍGUEZ

CONVOCATORIA: SEPTIEMBRE 2021
CALIFICACIÓN: 9

MÁSTER EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

4 DE SEPTIEMBRE DE 2021

Se abrió un claro entre las nubes,
hemos vuelto a ver el Sol.

Como dos presos comunes
en el tejao' de una prisión.

- Robe Iniesta (*Pepe Botika*)

AGRADECIMIENTOS

A María. Sin su empujón, nunca me hubiese embarcado en esta aventura.

A mi madre, a mi padre. Los motivos son incontables y no dejan nunca de crecer.

A Rubén, a Pablo. Por ser mentores primero, compañeros y amigos después, delante y lejos del teclado.

A la gata Pixi. Cuando comencé este máster no te conocía, y ahora que lo termino te he perdido. Gracias por este tiempo a tu lado, compañerita.

A la vida, aunque suene a tópico. Ahora, más que nunca, rebosa infinitas posibilidades. Cada día me siento más pequeño y humilde ante lo que se presenta como un camino sinuoso e incierto, y que se ramifica en cada pequeña decisión que tomamos – saludar una persona, asomarte debajo de un arbusto, agotar la noche antes de volver a casa. Es increíble lo distinta que sería mi vida sin esas pequeñas casualidades, y es que cada mañana me despierto pensando qué puede suceder hoy que pueda cambiar radicalmente al Sergio de dentro de un año. Nunca lo sabré, no obstante. Los hechos solo se pueden conectar hacia atrás, nunca hacia adelante. En resumen, poseemos sobre nuestras vidas un escaso control, y, en el mejor de los casos, no somos más que copilotos. Lo más cuerdo que podemos hacer la mayoría de las veces es confiar en que esa mezcla de preparación y oportunidad llamada "suerte" esté de nuestra parte. Si hubiese no obstante una verdad indiscutible, diría que es que este camino merece más la pena cuando se recorre acompañado.

RESUMEN

Servidor Orquestador de Propósito General

En este Trabajo de Fin de Máster se ha implementado un marco de trabajo para orquestar tareas de una manera sencilla y extensible.

Para ello se ha desarrollado un software principal, el orquestador, que ofrece toda la infraestructura necesaria para ejecutar tareas que terceras personas pueden implementar como módulos reutilizables, y que son cargados por el orquestador de manera dinámica al estilo de extensiones.

Adicionalmente se ha desarrollado la base de un servicio accesorio que permite el uso desde el orquestador de otras máquinas (para procesos que puedan ser necesarios en las tareas, como pueda ser compilación y pruebas) mediante una interfaz unificada, independiente del Sistema Operativo.

Para la comunicación entre el servidor orquestador y el servicio accesorio se ha desarrollado un protocolo binario de llamada a procedimiento remoto.

El marco de trabajo formado por el servidor orquestador y el servicio accesorio puede dar soporte a operaciones de desarrollo y pruebas comunes a muchos proyectos software, pero se ha diseñado desde una perspectiva agnóstica a su uso, pudiendo ser de utilidad en cualquier tarea que requiera ejecutar una serie de pasos de manera ordenada.

Palabras clave

Orquestación, automatización, integración continua, entrega continua, desarrollo ágil.

ABSTRACT

General-Purpose Orchestrator Server

This Master Thesis lays down a framework that allows orchestrating tasks.

The main piece of software that implements this framework is the orchestrator server. It provides the necessary infrastructure to run tasks. A third-party developer can implement these tasks as reusable modules. The orchestrator loads these modules as extensions.

The daemon is the secondary piece of software that implements this framework. It allows to run tasks such as builds or tests on separate machines from the orchestrator. The orchestrator accesses the daemon through a unified, Operating System independent interface.

The orchestrator and the daemon communicate to each other using a binary protocol. This protocol supports Remote Procedure Calling, and it is extensible as well.

The developed framework can provide support to development and testing operations. Regardless of that, the framework is completely agnostic to its final purpose. It can be of use in any task that requires executing a series of steps in order.

Keywords

Orchestration, automation, continuous integration, continuous delivery, agile development.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

Capítulo 1 - Introducción

En términos informáticos, se habla de orquestación para referirse a las tareas de configurar, coordinar y, en general, operar sistemas hardware y software de una manera automatizada. Incluye la definición de un flujo de trabajo para la consecución de unos objetivos concretos, y está íntimamente ligado con otros conceptos como control de versiones (donde es deseable que la definición de estos flujos de trabajo viva y evolucione), aprovisionamiento de máquinas, pruebas automatizadas o entrega.

Con la explosión de la "*arquitectura como servicio*" (provista por servicios de computación en la nube) los servicios de "*infraestructura como código*" (como *Terraform*[1] o *CloudFormation*[2]) están en auge. Adicionalmente, desde hace más de diez años, existen servicios de orquestación, como *Jenkins*[3] o *TeamCity*[4], enfocados a la integración y entrega continua (CI/CD, del inglés *Continuous Integration*[5]/*Continuous Delivery*[6]).

La orquestación puede dirigirse (entre otros propósitos) a operar estos servicios (que son independientes entre sí) de manera conjunta, orquestando su uso con la consecución de un objetivo más complejo que el que cada una de estas piezas puede conseguir por separado.

---

[1] https://www.terraform.io

[2] https://aws.amazon.com/cloudformation/

[3] https://www.jenkins.io

[4] https://www.jetbrains.com/teamcity/

[5] La práctica, en control de versiones, de tener integrado lo antes posible (varias veces al día de manera deseable) el trabajo del equipo en la línea principal.

[6] La práctica de producir nuevas versiones de un software en ciclos de trabajo cortos, asegurando que siempre exista una versión reciente que pueda ser liberada, y que dicha liberación se realice de manera automática.

1.1 Motivación

Independientemente del producto software que desarrolle un equipo o individuo, con casi total seguridad se podrán distinguir unas necesidades comunes, siendo la más importante de ellas hacer llegar a sus usuarios versiones verificadas de su producto.

Los pasos para generar estas versiones verificadas pueden ser, de un producto a otro, similares o radicalmente distintos. Seguramente, casi todos obtengan primero una copia del código fuente desde un repositorio de código, pero puede que no todos necesiten compilarlo, no todos ejecuten pruebas de la misma naturaleza, y no todos entreguen el producto a sus clientes usando el mismo canal o plataforma.

No hay una forma única de desarrollar software. Lo que va a ser común a todas ellas es que, tarde o temprano, va a ser necesario automatizar una parte o todo el proceso. Las causas pueden ser múltiples: aumento en la complejidad del proceso, en el tiempo que requiere, en las veces que se debe ejecutar al día…

En un primer momento la automatización recaerá, probablemente por su sencillez, en *scripts*. Si el equipo es disciplinado, estos *scripts* se ejecutarán siempre en la misma máquina, y se mantendrán bajo control de versiones. Sin embargo, con el tiempo, estos *scripts* se pueden empezar a multiplicar y a mutar. Además, puede ser que deban contemplar escenarios más complejos que requieran paralelismo, sincronización, gestión de errores… los *scripts* iniciales difícilmente sobrevivirán a este aumento de la complejidad – no fueron diseñados para ello, al fin y al cabo.

Llegados a ese punto, el equipo buscará en el mercado una solución que se adapte a sus necesidades, y tendrá que reescribir gran parte de la lógica ya implementada para adaptarse a la nueva plataforma. Si existiese desde el principio un marco de trabajo que hubiese permitido al equipo sentar sus automatizaciones sobre unas bases sólidas y extensibles, no se llegaría al punto de ruptura en el que una solución no es lo suficientemente buena y es necesario realizar una migración.

El objetivo de este Trabajo de Fin de Máster es definir e implementar dicho marco de trabajo.

1.2 Objetivos

Se desarrollan a continuación los objetivos principales y secundarios que han guiado la implementación de este Trabajo de Fin de Máster.

1.2.1 Objetivos principales

El principal objetivo de este Trabajo de Fin de Máster es el diseño e implementación de un **programa orquestador multiusuario extensible**, que servirá de marco de trabajo para que cualquier equipo (o individuo) puedan construir sobre ello la automatización de sus tareas de una forma accesible.

Este servidor orquestador (en adelante, simplemente "*el orquestador*") se encargará de ejecutar, bajo demanda de sus usuarios, ***trabajos***, que serán el conjunto de pasos necesarios para la consecución de un objetivo (por ejemplo, generar una nueva versión de un producto software).

Estos trabajos estarán definidos por el ***fichero de definición de trabajo***, un fichero en texto plano JSON que explicitará las variables de entrada para el trabajo, y el conjunto de pasos ordenados que el trabajo ha de llevar a cabo.

Las ***fases*** serán la implementación concreta de cada uno de los pasos del trabajo. Las fases serán lo que los desarrolladores implementen mediante código, y pueden ser tan específicas o generales como se quiera (este grado de especificidad dependerá pues del criterio del usuario y de cómo de reutilizables se desea que sean estas fases, nunca de una restricción impuesta por el software).

Para la implementación de estas fases el desarrollador contará con una biblioteca, a la que puede hacer referencia, que proporcionará la infraestructura básica (en términos de código, los atributos, clases y métodos comunes, o "*API público del orquestador*"). Una vez el desarrollador haya implementado sus propias fases en una biblioteca, puede cargarla como extensión en el orquestador, definir el trabajo a realizar

en un fichero de definición de trabajos, y ordenar su ejecución a través de un API REST[7] que el orquestador expone para su manejo.

1.2.2 Objetivos secundarios

Al tratarse de un marco de trabajo para terceros, no se puede prever de antemano cuáles van a ser las necesidades de estos usuarios, que son a fin de cuentas quienes deberán extender las capacidades del orquestador mediante bibliotecas que implementen nuevas fases.

Para evitar cualquier problema futuro provocado por las dependencias, se marca como objetivo secundario, pero de gran importancia, que el proyecto sea lo más auto contenido posible. Como medida del éxito en la consecución de este objetivo se escoge la cantidad de software que cualquier persona necesita instalar en una máquina para, o bien continuar con el desarrollo del proyecto, o bien ponerlo en ejecución.

Adicionalmente se marcan como objetivos secundarios el desarrollo de las siguientes piezas, consideradas accesorias al propio orquestador.

1.2.2.1 Aplicación cliente para el orquestador

Primero, una aplicación de terminal que permita el manejo remoto del orquestador a través del API REST que este expone para su manejo.

Al tratarse de un API REST accedido por HTTP(S), estas acciones se pueden llevar a cabo desde cualquier herramienta que soporte el protocolo. Sin embargo, se decidió implementar esta funcionalidad en una aplicación empaquetada junto al orquestador para evitar al usuario la complejidad extra inicial.

1.2.2.2 Servicio para el acceso y manejo básico de máquinas remotas

[7] API que se ajusta al conjunto de restricciones conocidas como REST (del inglés, *Representational State Transfer*).

Segundo, un servicio o *daemon*[8] que se instalará en cualquier posible máquina accesoria al trabajo del orquestador. El propósito de su existencia es dotar al usuario de una interfaz unificada para la realización de acciones tales como subir o bajar ficheros, o ejecutar procesos, que sea independiente del Sistema Operativo subyacente y de su modo de acceso. De este modo, no importa si un trabajo tiene que subir ficheros a una máquina remota y después disparar el proceso de compilación. Con el *daemon*, la forma de implementarlo (desde el punto de vista del orquestador), será idéntica tanto para sistemas tipo UNIX como para Windows (y, además, no se dependerá de software adicional). Sin el servicio, la implementación para cada Sistema Operativo variaría según el protocolo de conexión y aplicaciones disponibles.

1.2.2.3 Protocolo ligero de llamada a procedimiento remoto

Tercero, un protocolo RPC[9] sencillo (basado directamente en el *socket*) con un uso de red mínimo gracias a la serialización de datos completamente personalizada que comunique al orquestador con el servicio. En vez de utilizar soluciones existentes para la implementación de RPC tales como RabbitMQ[10], o para la implementación de la seralización tales como Google Protocol Buffers[11], se desea que el proyecto sea completamente auto contenido y que tenga unas dependencias mínimas. La implementación de un protocolo binario directamente sobre el *socket* en lugar de utilizar otros métodos más *verbosos* como JSON RPC[12] viene determinada por la necesidad de rendimiento a la hora de transferir ficheros.

1.2.2.4 Aplicación cliente del servicio de acceso unificado

---

[8] Un tipo de programa que se ejecuta en segundo plano y que no requiere de intervención directa por parte del usuario.

[9] Del inglés, *Remote Procedure Calling*, o Llamada a Procedimiento Remoto.

[10] https://www.rabbitmq.com

[11] https://developers.google.com/protocol-buffers

[12] https://www.jsonrpc.org

Cuarto y último, una aplicación de terminal mínima que permita la interacción con el servicio directamente por el usuario, con el fin de facilitar el probar manualmente las acciones que posteriormente se desean ejecutar desde el una fase del orquestador, prescindiendo para dichas pruebas de todos los pasos adicionales que requeriría la ejecución en el orquestador.

1.3 Plan de trabajo

Para la consecución de los objetivos marcados, en el plan de trabajo se pueden distinguir las fases que se describen a continuación.

Debiendo ser el proyecto desarrollado en su totalidad por una sola persona, no se impuso una metodología de desarrollo estricta. Se siguieron principios generales de metodología ágil y se fomentó una iteración rápida una vez los requisitos generales quedaron definidos, con el objetivo de tener piezas entregables funcionales en fases tempranas ("*Working software is the primary measure of progress*"[13]).

1.3.1 Fase 0 – propuesta inicial

Al tratarse el proyecto de una propuesta cerrada desde el principio, no existió una fase de investigación y documentación previa de la que surgiesen posteriormente posibles casos de uso y requisitos. Todos los requisitos sobre la funcionalidad de la aplicación habían quedado cerrados en la propuesta inicial.

1.3.2 Fase 1 – núcleo del orquestador y protocolo RPC

A la hora de encarar el desarrollo se priorizaron aquellas partes sobre cuya consecución exitosa había una mayor incertidumbre, con el fin de poder identificar rápidamente posibles bloqueantes en el camino y decidir temprano, en caso de ser necesario, un nuevo rumbo.

Por lo tanto, en una primera fase se comenzó el desarrollo del protocolo RPC (que afecta a servicio, a su aplicación cliente y al orquestador) y del motor de activación de

[13] De los principios detrás del Manifiesto Ágil (http://agilemanifesto.org/principles.html)

fases (el conjunto de piezas de código que permiten, por un lado, instanciar las clases concretas que implementarán las fases, y por el otro, ejecutar sobre estas instancias los métodos correctos con los argumentos adecuados).

1.3.3 Fase 2 – API REST para el manejo del servidor, y refinamientos

Una vez se determinó que una primera versión básica del protocolo RPC estaba listo para soportar los principales casos de uso y que era estable, y que la implementación inicial del motor de activación era capaz de convertir un fichero JSON en una secuencia de ejecuciones, se pasó a implementar las capas superiores (en lo que a abstracción se refiere) del orquestador, dejando el servicio (cuyo funcionamiento es más sencillo) para el final.

En esta segunda fase pues se implementaron los distintos métodos REST que permiten el manejo del orquestador a través de HTTP, el reporte del progreso de la ejecución, la gestión de usuarios y secretos, y se continuó refinando el funcionamiento del motor de activación y del protocolo RPC.

1.3.4 Fase 3 – Comunicación entre capas y carga dinámica de extensiones

Una vez que la gestión de usuarios, secretos y trabajos ya están implementadas (se pueden crear usuarios, los usuarios se pueden identificar y renovar sus credenciales de acceso, los usuarios pueden crear, modificar y eliminar secretos de usuario, y pueden pedir el lanzamiento de trabajos y consultar su progreso -aunque realmente no suceda nada cuando lo hacen-), se considera que el proyecto entra en una tercera fase dedicada a la integración entre capas.

El API REST tiene que ser capaz de lanzar realmente trabajos. Los trabajos lanzados han de ser capaces de "suplantar" al usuario para descifrar los secretos necesarios. Las fases de los trabajos han de ser capaces de reportar su progreso a las fases raíz, y estas a su vez a los usuarios, y los trabajos se deben poder detener en caso de ser necesario sin poner en riesgo la estabilidad de la aplicación. Es, por lo tanto, el momento de integrar el motor de activación, implementado en la primera fase, con las capas superiores implementadas en la segunda fase. Además, una vez que las responsabilidades finales de cada componente software están claras, se realiza la

división definitiva en bibliotecas de las piezas implementadas (necesaria para que terceros puedan desarrollar sus fases y cargarlas en el orquestador), y se implementa la carga dinámica de bibliotecas junto al descubrimiento automático de fases.

1.3.5 Fase 4 – Servicio y memoria de trabajo

Una vez el orquestador es capaz de ejecutar correctamente los trabajos pedidos por los usuarios, es el momento de extender sus capacidades a otras máquinas. Para ello, y con el protocolo RPC como base, se comienza el desarrollo de un conjunto limitado de funcionalidades en el servicio que sirvan, a la vez, para ofrecer una funcionalidad básica y para demostrar mediante ejemplos cuál es la forma de extender estas funcionalidades.

1.4 Estructura de la memoria

En el *Capítulo 2 - Estado de la cuestión*, se repasan brevemente algunas de las alternativas de software que existen en el mercado destinadas a la orquestación y a la automatización.

En el *Capítulo 3 - Lenguaje de programación, entorno de desarrollo y dependencias de terceros*, se introducen las herramientas y el entorno utilizados para el desarrollo, así como se motiva el lenguaje de programación escogido y se listan y justifican las dependencias de terceros utilizadas.

En el *Capítulo 4 - Estructura del proyecto, ejecución de las pruebas y ejecución de las aplicaciones* se introduce la estructura del proyecto y se relatan los pasos necesarios para navegar y compilar los componentes que se detallarán a continuación.

En el *Capítulo 5 - El orquestador* se introduce la pieza principal de software implementada en este Trabajo de Fin de Máster y se dan algunos detalles relevantes sobre su implementación. De manera análoga el *Capítulo 6 - El servicio* detalla el servicio, un complemento opcional del orquestador.

Finalmente, se incluyen las conclusiones de este trabajo y posibles líneas de trabajo futuro en el *Capítulo 7 - Conclusiones y trabajo futuro*.

El código fuente se encuentra disponible en GitHub, accesible a través de la dirección Web https://github.com/SergioLuis/orchestrator. Se distribuye bajo una licencia MIT (https://github.com/SergioLuis/orchestrator/blob/main/LICENSE).

Capítulo 2 - Estado de la cuestión

Cuando se habla de software orquestador, inevitablemente aparecerá otro término ligado: automatización. Si bien son conceptos íntimamente relacionados, es sencillo confundirlos. La automatización se refiere a conseguir que una tarea, cuyos pasos antes eran manuales, pase a ejecutarse por sí misma. Entre los ejemplos de tareas que se pueden automatizar encontramos crear una máquina en un servicio de virtualización, realizar una copia de seguridad, o compilar una nueva versión de una aplicación, y otros más cotidianos como rellenar un formulario en una página Web con la información personal del usuario, o encender las luces de la casa cuando anochece.

Orquestación es, sin embargo, ordenar y componer la manera en la que dichas automatizaciones se ejecutan. Esto amplía las posibilidades de qué es y qué no es un orquestador: en la práctica, hablaremos de orquestador para referirnos a un software que permita ejecutar una o más automatizaciones de manera ordenada, fácilmente modificable y reproducible.

Una forma primitiva de orquestación son los *scripts*. En un *script*, un usuario ordena una serie de invocaciones a comandos, que representarían cada una de las etapas de la automatización. Devolviendo los comandos tanto texto como códigos de error, o incluso objetos (como es el caso en PowerShell[14]), las salidas de unos comandos pueden utilizarse para alimentar las entradas de los posteriores. De hecho, incluso el propio *script* puede ser parametrizado, variando su comportamiento según los parámetros que utilice el usuario durante su ejecución. Sin embargo, los *scripts* que se ejecutan en línea de comandos no están pensados para la orquestación en escenarios complejos donde entran en juego múltiples máquinas y servicios, existe paralelismo, hay que mantener un estado, proporcionar visibilidad de la ejecución fuera de la propia máquina, es necesaria una política de reintentos y de recuperación de errores... Aunque pueden

[14] Al contrario de las *shells* que aceptan y devuelven texto, PowerShell acepta y devuelve objetos .NET.

formar parte de orquestaciones complejas, no son suficiente para proporcionar la infraestructura necesaria por sí mismos.

Sin embargo, estas características sí las posee el software de integración y entrega continua, que puede verse como una amplia familia (no la única) de software orquestador, de la que este Trabajo de Fin de Master toma inspiración, y de la que se pueden obtener múltiples ejemplos: entre ellos, se pueden encontrar tanto orquestadores de código libre como propietario, tanto gratuitas como de pago, y tanto diseñadas para ser ejecutados en las máquinas de los usuarios como para ser accedidos a través de servicios Cloud. Entre otros muchos nombres que destacan en cuanto a relevancia en los motores de búsqueda, están Jenkins[15], CircleCI[16] y TeamCity[17].

Todos ellos tienen unas características que se podrían considerar comunes. No obstante, cada solución las implementa de una forma u otra dependiendo de la filosofía del producto, utilizando distinta terminología e interfaces de usuario, e introduciendo pequeñas diferencias que puedan hacer preferible una solución concreta frente a la competencia en un determinado escenario.

Por ello, en vez de repasar en profundidad cada una de las opciones comerciales (tarea, por otro lado, imposible), se pasa a enumerar las características comunes en términos generales.

El software orquestador destinado a la integración y a la entrega continua:

- Tiene la capacidad de definir uno o más planes de ejecución. En Jenkins, "*pipelines*", en CircleCI, "*workflows*", y en TeamCity, "*build chain*".
- Cada uno de estos planes de ejecución estará formado por una serie de etapas reconfigurables que se definen en una representación que no es el propio código fuente del orquestador. En Jenkins, el *pipeline* se define en

[15] https://www.jenkins.io

[16] https://circleci.com

[17] https://www.jetbrains.com/teamcity/

un fichero con nombre `Jenkinsfile`. En CircleCI, mediante el fichero `config.yml`, y en TeamCity, mediante un fichero estilo Maven.

- Cada una de las etapas de los planes de ejecución se especializan en una tarea concreta, y son bloques reutilizables. Algunos de estos bloques son tan genéricos como ejecutar un script, mientras que otros son mucho más especializados, diseñados para integrarse con un servicio o API concretos.
- Las capacidades del orquestador pueden ampliarse mediante la instalación de nuevos bloques o etapas, bien mediante la configuración de una etapa predefinida, que se podrá pasar a usar desde múltiples planes de ejecución, o bien mediante la instalación de extensiones.
- Cada uno de estos planes de ejecución pueden definir políticas de expiración, de reintentos, y de recuperación en caso de fallo.
- Adicionalmente, estos planes de ejecución se pueden parametrizar antes de comenzar. Por ejemplo, mediante el paso de parámetros, se puede indicar de qué punto de un repositorio de código se deben obtener los ficheros para la compilación y pruebas, o a qué canal de distribución se deben subir los artefactos resultantes.
- Mientras un plan de ejecución se encuentra en marcha, es posible consultar su estado. Bien mediante un log de ejecución, una representación gráfica en un panel web, o a través de un API.
- Los planes de ejecución se pueden encadenar y ramificar según el resultado. Por ejemplo, si el plan de construcción ha tenido éxito, continuar con el de pruebas. En caso contrario, continuar con el de reporte. Si el de pruebas ha tenido éxito, continuar con el de despliegue, etc.
- La máquina que aloja a los orquestadores no es necesariamente la única que ofrece servicio. Cuando el objetivo es compilar y probar software, suele ser un requisito hacerlo en distintas plataformas. Para ello, el orquestador principal se puede valer de máquinas accesorias, utilizadas mediante agentes instalados en estas máquinas, accedidos a través de la red.

En este contexto, este Trabajo de Fin de Máster implementa un orquestador y un servicio agente inspirados en soluciones ya existentes en el mercado. Sin embargo,

mientras que las soluciones ya existentes están, por filosofía del producto, fuertemente enfocadas al CI/CD, el servidor orquestador ha sido diseñado de manera agnóstica al uso que se le quiera dar: el orquestador puede servir tanto para sustituir una instalación Jenkins para generar nuevas versiones de una aplicación como para automatizar y orquestar cualquier otra parte de un negocio. Por ejemplo, se podría implementar en fases del orquestador el generar entradas en PDF para un evento deportivo y enviarlas a los destinatarios de correo encontrados en un fichero CSV. Y, si en el futuro, los destinatarios se obtienen de una base de datos, únicamente sería necesario cambiar una de las fases. El orquestador ofrece pues una alternativa tan potente como un orquestador de CI/CD comercial y tan sencilla como un *script*. Será el desarrollador el que decida cuál va a ser el grado de complejidad del sistema que se implemente utilizándolo, y el orquestador será capaz de soportar desde los escenarios más sencillos a los más enrevesados.

Adicionalmente, mientras que las soluciones de orquestación ya existentes tienen una curva de aprendizaje elevada, pues es necesario manejar un amplio abanico de conceptos, el orquestador presenta un marco de trabajo mínimo y sencillo de aprender. De igual forma sucede para los recursos necesarios para su ejecución. Mientras que, productos como Jenkins requieren al menos 10GiB[18] de espacio libre en disco, el orquestador en una instalación limpia requiere menos de 110MiB[19] (tanto como ocupan los binarios de las distintas aplicaciones junto con el *runtime* de .NET necesario para su ejecución, y las dependencias de terceros). Adicionalmente, no necesita para funcionar ningún otro software como pueda ser un gestor de bases de datos, un servidor PHP o una caché distribuida. Todo se encuentra empaquetado en una única aplicación.

[18] Gibibytes. 1GiB = 2^{30} bytes.

[19] Mebibytes. 1MiB = 2^{20} bytes.

Capítulo 3 - Lenguaje de programación, entorno de desarrollo y dependencias de terceros

Se relatan a continuación las decisiones tomadas en cuanto al lenguaje de programación en el que se desarrolla la totalidad del proyecto, y qué entornos de desarrollo integrados se pueden utilizar para trabajar con ello. Finalmente se justifican las dependencias de terceros que se han introducido.

3.1 Lenguaje de programación

El lenguaje de programación escogido ha sido C#. En concreto la revisión 9, implementada por .NET (leído "*punto-net*", o "*dot-net*" en inglés) en su versión 5. Los motivos de escoger la tecnología .NET son los siguientes:

3.1.1 Tecnología multiplataforma

.NET es una tecnología multiplataforma. Mientras que .NET Framework 4.x (el sufijo "*Framework*" desaparece del nombre en la versión 5) estaba disponible de manera oficial únicamente para el Sistema Operativo Microsoft Windows (aunque en SO *UNIX-like* se podía ejecutar sobre el *Mono Runtime*, desarrollado por Xamarin[20]), empezando en .NET Core (la nomenclatura "*Core*" también desaparece en la versión 5 – Microsoft ha estado transformando y unificando el ecosistema .NET durante los últimos años) el soporte para los tres sistemas operativos principales (Windows, macOS y GNU/Linux) y multiarquitectura (no solo x86 y x64, sino también ARM, ARM64, y recientemente Apple Silicon) es oficial. De hecho, este Trabajo de Fin de Máster ha sido probado con éxito en las siguientes configuraciones software/hardware:

- Windows 10 sobre procesador Intel de 64 bits.
- Ubuntu Server 20.04 sobre una Raspberry Pi 4 con procesador ARM64.

[20] Xamarin fue finalmente adquirida por Microsoft para servir como base a las versiones multiplataforma de .NET.

- macOS Big Sur sobre un MacBook Pro con procesador Apple Silicon (nomenclatura comercial M1).

3.1.2 Lenguaje de programación con soporte a la reflexión

C# tiene APIs específicas de reflexión – la capacidad de un programa para examinar y modificar su propia estructura u comportamiento. Esto permite cargar código externo en tiempo de ejecución, descubrir e instanciar nuevos tipos y descubrir y ejecutar nuevos métodos que no existían en el momento en el que la aplicación fue compilada.

Esta característica es clave a la hora de permitir que terceras personas desarrollen para el orquestador sus propias fases y las carguen como extensiones, sin tener que pasar por el proceso de compilar el propio orquestador para añadirlas.

3.1.3 Tecnología Open Source

Aunque .NET Framework no lo era, .NET Core (2.x, 3.x) y .NET (5 en adelante) es código libre, gestionado por la *.NET Foundation*[21], una entidad independiente de Microsoft y sin ánimo de lucro a la que cualquier individuo puede unirse e influir en las decisiones que se toman respecto al rumbo de .NET. Esta licencia libre además no incluye únicamente al lenguaje – este es solo una parte pequeña (aunque fundamental) del ecosistema. De la *.NET Foundation* dependen además las bibliotecas base, el *runtime* de ejecución, los compiladores, otros lenguajes de programación como F#, algunos *frameworks* ofrecidos por Microsoft (como ASP.NET) *frameworks* que antes dependían exclusivamente de terceros (como NUnit), etc.

Esto significa que, al contrario que en otros lenguajes como Java, no es necesario el pago de regalías al propietario original del lenguaje para distribuir el software, independientemente de la licencia y del precio escogidos para este.

[21] https://dotnetfoundation.org

3.1.4 Nativo para la Web desde el primer momento

.NET tiene incluido en las bibliotecas base y en el *runtime* todo lo necesario para trabajar con Web desde el primer momento, sin necesitar *frameworks* de terceros. Gracias a ASP.NET[22] sobre Kestrel[23] se puede añadir a cualquier aplicación .NET 5 la posibilidad de levantar un pequeño (pero muy eficiente) servidor HTTP, algo que era necesario si se quería permitir el acceso al orquestador a través de un API REST.

3.1.5 Modelo de programación asíncrona sencillo

C# abstrae un modelo de programación asíncrona sencillo a través del propio lenguaje y del compilador. Mediante el uso del API asíncrono de .NET (que incluye no solo clases como *Task*, sino palabras reservadas del propio lenguaje como *async* y *await*, que el compilador más tarde transforma en máquinas de estados, ejecuciones en hilos secundarios y llamadas de vuelta) se pueden desarrollar aplicaciones multi-hilo capaces de atender un alto volumen de peticiones I/O de una manera eficiente, mediante la abstracción de en qué hilo se ejecuta qué código (aunque, como en cualquier otro lenguaje de programación, sea conveniente conocer qué sucede debajo de la abstracción para asegurar la corrección del programa).

Sobre este API asíncrono se basará el modelo de fases que implementa el orquestador.

3.1.6 Foco en la reducción de errores en el desarrollo

C#, mediante el compilador y el *linter*[24], incluyen comprobaciones en tiempo de compilación de los errores más comunes derivados de un mal uso de las referencias nulas. Esta funcionalidad hay que activarla explícitamente en los ficheros de proyecto, y causa que tanto desde el entorno de desarrollo como durante el proceso de compilación se emitan avisos en caso de que una referencia pueda ser nula y no se esté

---

[22] https://dotnet.microsoft.com/apps/aspnet

[23] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel

[24] Herramienta de análisis estático de código fuente.

tratando correctamente, reduciendo así la posibilidad de errores derivados de un mal manejo de los nulos.

Además, gracias a que *Roslyn*[25] (el compilador, código abierto) expone APIs para ejecutar consultas sobre piezas de código, la calidad de los analizadores estáticos ha aumentado enormemente en los últimos años, siendo estos capaces de detectar en los patrones de código posibles causas de fallo mucho más difusas que un nulo – como el uso de más parámetros de los esperados en una cadena de formateo:

```
string.Format("{0}{1}", "argumento 1", "argumento 2", "argumento 3");
```

3.1.7 Conjunto de herramientas completo

Las propias herramientas de la plataforma incluyen un extenso soporte para una gran variedad de tareas cotidianas, como añadir y ejecutar pruebas o hacer "andamiaje" (más común es el término anglosajón "*scaffolding*"[26]) de nuevos proyectos, lo que reduce las dependencias de herramientas de terceros para facilitar el desarrollo. La herramienta de línea de comandos por defecto (`dotnet`) incluye incluso soporte para generar, de una manera sencilla, certificados de desarrollo, evitando al desarrollador tener que lidiar con las herramientas para la labor de cada Sistema Operativo concreto (normalmente complicadas).

3.2 Entorno de desarrollo

El entorno de desarrollo principal escogido ha sido *Visual Studio Code*[27], por ser un editor multi-plataforma (disponible para los tres Sistemas Operativos principales), con soporte nativo para C# gracias a *OmniSharp*[28], y gratuito. No obstante, el código desarrollado puede ser igualmente ejecutado, depurado y probado en otros entornos

[25] https://github.com/dotnet/roslyn

[26] Técnica referida a la generación automática de los ficheros y estructuras de directorios necesarios en un proyecto.

[27] https://code.visualstudio.com

[28] https://github.com/OmniSharp/csharp-language-server-protocol

de desarrollo que soporten .NET, tales como *Visual Studio*[29] (disponible para Windows y macOS), o *JetBrains Rider*[30], o incluso sin un entorno de desarrollo integrado – con las herramientas de terminal incluidas en el SDK de .NET 5 y editores de texto plano.

3.3 Dependencias de terceros

En la sección *Objetivos secundarios* se marcó como objetivo secundario, pero de gran peso, reducir al mínimo posible la dependencia del proyecto de software de terceros.

Con ello se pretende conseguir un triple beneficio: por un lado, reducir la complejidad del software. Por otro, reducir los requisitos software para el desarrollo, reduciendo la barrera de entrada a nuevos contribuidores. Y, por último, reducir los requisitos software en el momento de ejecución, facilitando la adopción.

El razonamiento que hay detrás de todas las veces en las que se decidió no usar una biblioteca o tecnología accesoria es sencillo: siempre es posible incrementar la complejidad de una aplicación, pero una vez introducida esta complejidad, difícilmente se va a poder prescindir de ella en el futuro. Por lo tanto, en vez de añadir al proyecto tecnología por el mero hecho de utilizarla, se retrasaron estas decisiones hasta el momento en el que añadir algún nuevo componente fuese imprescindible.

3.3.1 Reducir la complejidad del software

Reducir las dependencias de terceros puede ayudar a reducir la complejidad del software. Una vez se decide utilizar una biblioteca concreta para un caso de uso específico, usar de nuevo esa biblioteca para casos que pueden ser parecidos no tiene un coste adicional, y no obliga al desarrollador a razonar sobre las necesidades reales de llevar a cabo una tarea de una determinada manera. A lo mejor no es necesario una biblioteca para hacer inyección de dependencias, solo se han de pasar las referencias hacia abajo. O a lo mejor no es necesario una biblioteca para propagar

[29] https://visualstudio.microsoft.com

[30] https://www.jetbrains.com/rider/

eventos en la aplicación, se pueden utilizar patrones de diseño software bien conocidos para ello, como el observador.

Adentrándonos en una decisión concreta, no se planteó la necesidad de hacer uso de una base de datos porque el orquestador no haría un uso intensivo de ella, ni requiere las operaciones a las que típicamente una base de datos da soporte. Por lo tanto, se pueden utilizar ficheros de texto (manejados, eso sí, de manera correcta, para prevenir corrupciones y pérdidas) para el mismo propósito.

3.3.2 Reducir los requisitos de desarrollo

Empezar a colaborar en cualquier proyecto software suele tener una gran barrera de entrada. Esta barrera suele intentar suavizarse mediante documentación, guías de inicio, o marcando tareas como "aptas para principiantes" (las llamadas "*low-hanging fruit*") pero cuanto menor sea la barrera de entrada, más posibilidades hay de atraer a posibles colaboradores al proyecto. Lo que, a largo plazo, se puede traducir en unas mayores posibilidades de supervivencia y adopción, aunque sea entre un público de nicho.

Reducir las dependencias software en el entorno de desarrollo disminuye una de las barreras de entrada iniciales para esos deseados colaboradores.

El software nace, cambia y muere rápidamente. Aparecen nuevas tecnologías que prometen hacer lo mismo, pero mejor. Se introducen cambios que rompen la compatibilidad hacia atrás. Se descubren fallos de seguridad que obligan a retirar versiones afectadas. Las empresas y los equipos detrás del mantenimiento desaparecen o son comprados. En definitiva, y a no ser que se esté haciendo un trabajo continuo de mantenimiento y actualización de dependencias, la configuración software necesaria para colaborar activamente en el proyecto pasa a ser tan difícil de conseguir que expulsa a la gente del proyecto incluso antes de haber entrado.

3.3.3 Facilitar la adopción

Reducir las dependencias facilita la implantación del marco de trabajo. A la hora de implantar una nueva herramienta en una organización, pueden existir barreras tanto legales como técnicas. Por lo tanto, cuantas más sean las dependencias del marco de

trabajo, mayores posibilidades habrá de encontrarse con alguna de estas barreras a la hora de la implantación.

La funcionalidad principal del orquestador es ejecutar código de terceros. Esto significa, desde un punto de vista pragmático, que el *cómo* funcione el marco de trabajo (si requiere una u otra base de datos, una u otra caché) es irrelevante para el usuario – lo que el usuario desea es programar sus fases y ejecutarlas en sus trabajos. Si el marco de trabajo tiene dependencias que entran en conflicto con las dependencias del usuario, y no se pueden cambiar, se obligará a este último a buscar una alternativa, rompiendo la promesa de que lo que importa de verdad es su código, pues tendrá que trabajar alrededor de una incompatibilidad que no debería haber estado ahí en un primer momento.

3.3.4 Biblioteca de terceros utilizadas

Las bibliotecas de terceros utilizadas, su justificación y licencia son las siguientes:

3.3.4.1 NUnit

NUnit es un *framework* de pruebas compatible con todos los lenguajes .NET. Forma parte de la Fundación .NET, es *Open Source* y se distribuye bajo una licencia MIT (https://nunit.org/nuget/nunit3-license.txt). Se utiliza para implementar las pruebas unitarias de las aplicaciones desarrolladas.

3.3.4.2 Moq

Moq es un *framework* que permite implementar pruebas dirigidas por comportamiento (*behaviour-driven testing*[31]) compatible con C#. Es *Open Source* y se

[31] *Behaviour Driven Testing*, o BDT, es una técnica de pruebas en la que cada componente se aísla y se prueba de manera individual, reemplazando sus dependencias por "*mocks*" o falsas implementaciones con un comportamiento determinado definido en el propio test. De esta forma, el componente que se está probando se puede considerar correcto bajo los comportamientos definidos para sus dependencias, que, de otra forma, podrían ser difíciles o imposibles de probar.

distribuye bajo una licencia BSD-3 (https://github.com/moq/moq4/blob/main/License.txt). Se utiliza en conjunto con NUnit para la implementación de las pruebas de unidad.

3.3.4.3 NLog

NLog es un *framework* de rastreo o *logging* flexible, gratuito y Open Source. Se distribuye bajo una licencia BSD (https://github.com/NLog/NLog/blob/master/LICENSE.txt). Se utiliza para dejar un rastro de la ejecución de la aplicación en puntos estratégicos. Estos mensajes deberían permitir, en caso de error, realizar un análisis *post-mortem* de este. En algunos casos los mensajes de traza pueden servir también con fines de auditoría para conocer qué usuario ejecutó qué acción. Sin embargo, los mensajes dejados no tienen ningún tipo de protección ante falsificaciones.

3.3.4.4 Spectre.Console

Spectre.Console es un *framework* de dibujado de interfaz de usuario en línea de comandos. Es *Open Source* y se distribuye bajo una licencia MIT (https://github.com/spectreconsole/spectre.console/blob/main/LICENSE.md). Se utiliza para el dibujado en consola de progresos, tablas y otras interfaces en las aplicaciones de terminal implementadas.

3.3.4.5 System.CommandLine

System.CommandLine es un *framework* de interpretación de parámetros de línea de comandos. Es parte de los proyectos de la fundación .NET, es *Open Source* y se distribuye bajo una licencia MIT (https://github.com/dotnet/command-line-api/blob/main/LICENSE.md). Se utiliza para interpretar, en todas las aplicaciones, los parámetros pasados por la línea de comandos y ejecutar, en cada caso, el código adecuado.

3.3.4.6 Newtonsoft.Json

Newtonsoft.Json es un framework de serialización y deserialización de JSON. Es Open Source y se distribuye bajo una licencia MIT

(https://github.com/JamesNK/Newtonsoft.Json/blob/master/LICENSE.md). Se utiliza para serializar (en la aplicación cliente) y deserializar (en el orquestador) el contenido del fichero de definición de trabajos, así como para serializar y deserializar el cuerpo de todas las peticiones HTTP que se realizan al API REST a través del cuál se maneja el orquestador.

3.3.4.7 System.IdentityModel.Tokens.Jwt

System.IdentityModel.Tokens.Jwt es una biblioteca que permite, de una manera sencilla, manejar *Json Web Tokens*. Forma parte del marco de trabajo de ASP.NET, pero se distribuye por separado para evitar el aumento de tamaño de las aplicaciones que no usen determinadas funcionalidades. Por tanto, es uno de los proyectos de la Fundación .NET, es *Open Source*, y se distribuye bajo una licencia MIT (https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/dev/LICENSE.txt). Sobre el uso de este tipo de *tokens* se profundiza en la sección *Tokens de acceso, de refresco y de trabajos*.

Capítulo 4 - Estructura del proyecto, ejecución de las pruebas y ejecución de las aplicaciones

Para compilar y ejecutar la aplicación, y compilar y ejecutar las pruebas, son necesarios los siguientes pasos. Se presupone que en el momento de seguirlos se dispone de una copia del repositorio.

4.1 Estructura del proyecto

La totalidad del código fuente se encuentra debajo del directorio `/src`. Ahí se encuentra "`orchestrator.sln`", el fichero más importante a la hora de trabajar con el proyecto. Es un fichero de solución (en jerga .NET, "*solution*", denotado por la extensión `*.sln`) que tiene enlazados la totalidad de los proyectos (en jerga .NET, "*projects*", denotados por la extensión `*.csproj`) y es, por tanto, el principal punto de entrada tanto para compilar todos los artefactos como para ejecutar las pruebas.

Debajo de `/src` se encuentran los siguientes directorios (en orden alfabético). Para cada directorio se señalan también los subdirectorios de especial interés:

- `client`: contiene el código fuente de la aplicación de línea de comandos con la que se puede operar el orquestador a través de su API REST.
- `commandlinecommon`: contiene el código fuente de la biblioteca con las dependencias comunes a todas las aplicaciones que requieren hacer uso de la línea de comandos para cosas tales como interpretar argumentos o escribir texto con formato.
- `daemon`: contiene el código fuente del servicio que se instala en máquinas accesorias al propio orquestador, así como sus dependencias y cliente de línea de comandos.
    - `daemonclient`: contiene el código fuente de la aplicación de línea de comandos con la que se puede interactuar con el servicio.

    - `daemoncommon`: contiene el código fuente de la biblioteca con código común tanto al cliente de línea de comandos del servicio (`daemonclient`) como al servicio en sí mismo (`daemonserver`). Este código común incluye toda la capa de red (la implementación de las llamadas remotas), por lo que esta biblioteca `daemoncommon` referencia, a su vez, a la biblioteca que se encuentra en `/src/rpc`.
    - `daemonserver`: contiene el código fuente del servicio.
- `jobscommon`: contiene el código fuente de la biblioteca con las dependencias comunes a todos los trabajos. Esta es la biblioteca a la que terceras partes tienen que hacer referencia para implementar sus propias fases y cargarlas en el orquestador.
- `rpc`: contiene el código fuente de la biblioteca que implementa el protocolo de llamada a procedimiento remoto.
    - `client`: contiene el código fuente que implementa la parte cliente del protocolo RPC.
    - `server`: contiene el código fuente que implementa la parte servidor del protocolo RPC.
    - `shared`: contiene clases que tanto la parte cliente como la parte servidor del protocolo RPC necesitan para funcionar.
- `server`: contiene el código fuente de la aplicación servidor orquestador.
    - `jobs`: contiene el código fuente necesario para la gestión de trabajos, incluyendo el motor de activación de fases.
    - `settings`: contiene los modelos necesarios para cargar y escribir ficheros de configuración, que ayudan a ajustar como sea necesario el comportamiento de la aplicación.
    - `users`: contiene el código fuente necesario para la gestión de usuarios y sus secretos, incluyendo la autenticación.

    - `web`: contiene el código fuente necesario para el funcionamiento de API REST que sirve para el manejo del orquestador.
- `tests`: contiene el código fuente de las pruebas.
    - `nunitclient`: contiene las pruebas unitarias de la aplicación de cliente.
    - `nunitdaemon`: contiene las pruebas unitarias del servicio accesorio.
    - `nunitrpc`: contiene las pruebas unitarias del protocolo de llamada a procedimiento remoto.
    - `nunitserver`: contiene las pruebas unitarias del servidor orquestador.

4.2 Instalación del SDK de .NET 5

Como se ha señalado en secciones anteriores, la aplicación se ejecuta sobre .NET 5. No se distribuyen binarios, pero en el futuro sería interesante hacerlo (se profundiza en la sección *Distribuir los binarios*). Para ello hay distintas alternativas, que se repasan en la sección *Compilación de las aplicaciones*.

El Kit de Desarrollo de Software (SDK en adelante, por sus siglas en inglés) de .NET 5 está disponible para las versiones más recientes de los Sistemas Operativos principales (Windows, macOS, distribuciones GNU/Linux). Para las tres plataformas existen tanto métodos de instalación tradicional (MSI para Windows, instalador en formato PKG para macOS, y distribución a través de repositorios para GNU/Linux), como binarios que el usuario puede descargar en su máquina y utilizar directamente.

Para obtener el método adecuado de instalación para la plataforma escogida, es necesario navegar a la página Web oficial de Microsoft[32]. Se garantiza que todo el software y las pruebas implementadas funcionan correctamente con, al menos, la versión de .NET 5.0.302, publicada originalmente el 13 de julio de 2021.

[32] https://dotnet.microsoft.com/download/dotnet/5.0

Una vez seguidos los pasos oportunos para la plataforma (que quedan fuera del alcance de este trabajo, pues pueden variar en el tiempo), será necesario comprobar que las herramientas están disponible a través de la línea de comandos. Para ello, se abrirá una terminal y se ejecutará el comando "`dotnet --info`". La salida estándar debería ser muy similar a la que se señala (obviando información específica de la instalación, como el *commit* donde se generaron los binarios, o los directorios que contienen los SDK):

```
ubuntu@onyx:~/dotnet$ ./dotnet --info
.NET SDK (reflecting any global.json):
 Version:   5.0.100
 Commit:    5044b93829

Runtime Environment:
 OS Name:     ubuntu
 OS Version:  20.04
 OS Platform: Linux
 RID:         ubuntu.20.04-arm64
 Base Path:   /home/ubuntu/dotnet/sdk/5.0.100/

Host (useful for support):
  Version: 5.0.0
  Commit:  cf258a14b7

.NET SDKs installed:
  5.0.100 [/home/ubuntu/dotnet/sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 5.0.0
[/home/ubuntu/dotnet/shared/Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 5.0.0 [/home/ubuntu/dotnet/shared/Microsoft.NETCore.App]

To install additional .NET runtimes or SDKs:
  https://aka.ms/dotnet-download
```

4.3 Ejecución de las pruebas

En el momento de redactar esta sección de la memoria se han implementado un total de 172 pruebas unitarias.

Para ejecutarlas, es necesario, de nuevo desde el directorio `/src`, ejecutar el comando "`dotnet test`". Por defecto, dicho comando busca un fichero de solución de .NET en el directorio actual. Como en `/src` únicamente se encuentra `orchestrator.sln`, ese será el escogido. Si hubiese más de uno, sería necesario pasarlo como argumento de manera explícita mediante "`dotnet test orchestrator.sln`".

4.4 Compilación y ejecución de las aplicaciones

Se describe a continuación cómo compilar y ejecutar las distintas aplicaciones que conforman el proyecto.

4.4.1 Compilación de las aplicaciones

Gracias a la versatilidad de .NET 5, existen diversas formas de compilar el orquestador y las aplicaciones asociadas. Dependiendo de lo que se quiera conseguir, la forma de hacerlo variará. Entre otras, las posibilidades más utilizadas suelen ser:

- Ejecutar la aplicación directamente.
- Compilar la aplicación para su posterior ejecución en la propia máquina, con símbolos de depuración.
- Empaquetar la aplicación para ejecutarla en otra máquina con el *runtime* de .NET 5 instalado, con o sin símbolos de depuración (tanto misma arquitectura como "*compilación cruzada*").
- Empaquetar la aplicación y todas sus dependencias en un único fichero ejecutable autoextraíble.

Los dos comandos fundamentales de `dotnet` que se utilizarán son los siguientes:

- `build`: compilará la aplicación para utilizarla en la misma máquina. Es el comando principal a la hora de probar cambios durante el desarrollo.
- `publish`: publicará la aplicación, lista para su transferencia a una máquina distinta. Es el comando principal a la hora de construir los distintos artefactos antes de hacer un despliegue o entregar una nueva versión a los usuarios.

Típicamente no se utilizará ningún argumento adicional con el comando `build`. Este genera los binarios correspondientes, así como las dependencias que estén marcadas para su copia en la salida, en un determinado directorio de la máquina. Una vez compilada la aplicación, tendremos que conocer su estructura para identificar el binario que contiene el punto de entrada.

Cuando se compila con el comando `build`, por defecto se hará utilizando la configuración de DEBUG. Se puede ejecutar `build` también en `RELEASE`, que es la otra configuración que se crea por defecto, o en cualquiera de las configuraciones personalizadas que hayamos podido añadir a los ficheros de solución y proyecto. Sin embargo, hacerlo en `RELEASE` no es lo común, debido a los motivos que se señalan a continuación.

El compilar con la configuración de `DEBUG` activa causa que el compilador evite ciertas optimizaciones (*inlining* de métodos[33], desenrollado de bucles[34], etc.) y genera como artefacto adicional los ficheros de símbolos. Estos ficheros, con extensión `*.PDB` (*Program Data Base*), son fundamentales a la hora de depurar la aplicación, tanto con un depurador interactivo conectado como a la hora de obtener trazas mediante *logs* y otros métodos de monitorización cuando la aplicación falla. Los ficheros PDB contienen las relaciones entre los símbolos originales, en el fichero de código fuente, con los símbolos en la(s) DLL(s) que contienen el IL[35] ejecutable. Estos PDB son los que permiten hacer cosas tales como ejecutar la aplicación paso a paso, cambiar el contador de programa a una determinada línea e inspeccionar el contenido de las variables directamente relacionándolo con el código fuente en el entorno de desarrollo, entre otras funcionalidades típicas de un depurador interactivo.

Igual que sucedía con el comando `test`, el comando `build` toma por defecto el fichero de solución que exista en el directorio actual. Para compilar todos los artefactos de la solución es necesario pues, desde `/src`, ejecutar "`dotnet build`". La salida estándar (recortada aquí) indicará dónde se encuentran los artefactos generados.

---

[33] Reemplazar la llamada a una función por el cuerpo de dicha función, para evitar el trabajo añadido de realizar la invocación, acelerando la ejecución.

[34] Multiplicar el cuerpo del bucle tantas veces sea necesario para evitar el trabajo extra de mantener un contador, realizar una comparación con el contador al final del bucle, y realizar un salto condicional a otra parte del código para continuar o finalizar la ejecución del bucle.

[35] *Intermediate Language*, el código de bajo nivel independiente de la arquitectura que el *runtime* transforma a código específico para el procesador.

```
$ dotnet build
Microsoft (R) Build Engine version 16.8.3+39993bd9d for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  jobscommon -> (...)/jobscommon/bin/Debug/net5.0/jobscommon.dll
  commandlinecommon -> (...)/commandlinecommon/bin/Debug/net5.0/commandlinecommon.dll
  rpc -> (...)/rpc/bin/Debug/net5.0/rpc.dll
  client -> (...)/client/bin/Debug/net5.0/client.dll
  server -> (...)/server/bin/Debug/net5.0/server.dll
  nunitrpc -> (...)/tests/nunitrpc/bin/Debug/net5.0/nunitrpc.dll
  daemoncommon -> (...)/daemon/daemoncommon/bin/Debug/net5.0/daemoncommon.dll
  nunitdaemon -> (...)/tests/nunitdaemon/bin/Debug/net5.0/nunitdaemon.dll
  nunitclient -> (...)/tests/nunitclient/bin/Debug/net5.0/nunitclient.dll
  daemonserver -> (...)/daemon/daemonserver/bin/Debug/net5.0/daemonserver.dll
  nunitserver -> (...)/tests/nunitserver/bin/Debug/net5.0/nunitserver.dll
  daemonclient -> (...)/daemon/daemonclient/bin/Debug/net5.0/daemonclient.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:14.97
```

El comando `publish` sirve para empaquetar la aplicación de forma que esta se pueda llevar a otra máquina, y ejecutarla, independientemente de si la máquina tiene o no instalado el *runtime* de .NET 5. Esto se consigue empaquetando junto a la aplicación el propio *runtime* (que es lo suficientemente pequeño como para hacerlo asumible) y la cantidad mínima de bibliotecas estándar necesarias (típicamente, aquellas bajo el espacio de nombres *System* o *Microsoft*) para el funcionamiento de la aplicación. Se muestra a continuación un ejemplo de la salida estándar al ejecutar el comando `publish`, cortada debido a su longitud.

```
$ dotnet publish -r osx-x64 -o /Users/sluisp/Desktop/orchestrator -c Release

Microsoft (R) Build Engine version 16.8.3+39993bd9d for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  Restored (...)/commandlinecommon/commandlinecommon.csproj (in 862 ms).
  (...)

  commandlinecommon -> (...)/commandlinecommon/bin/Release/net5.0/osx-
x64/commandlinecommon.dll
  (...)

  daemonclient -> /Users/sluisp/Desktop/orchestrator/
```

Sin embargo, el tamaño de las aplicaciones empaquetadas mediante este método es grande en proporción al tamaño que aportan las bibliotecas que realmente son de interés para el desarrollador (es decir, aquellas donde se encuentra el código de la aplicación principal).

Las ventajas de empaquetar las aplicaciones junto a todas sus dependencias y al propio *runtime* son claras: el usuario no necesita nada más que los binarios proporcionados para ejecutar la aplicación. La barrera de entrada en este frente es mínima comparada con otro software que pueda requerir la instalación de controladores o motores gráficos que, bien por tamaño, bien por motivos de licencia, no se pueden distribuir junto a la propia aplicación.

Las desventajas son también claras: la imposibilidad de reutilizar bibliotecas que el usuario pueda tener ya en su sistema, y el aumento del tamaño del paquete a distribuir (que puede incurrir en costes económicos si la aplicación se distribuye mediante un sistema como *Google Cloud* o *Amazon Web Services*, que factura por tráfico emitido).

Para publicar una aplicación para su empaquetamiento y distribución mediante el comando `publish` se utilizarán típicamente los argumentos `--configuration`, `--runtime` y `--output`.

- `--configuration`: indica la configuración con la que se compilará la aplicación. Como se ha visto anteriormente, típicamente será `DEBUG` o `RELEASE`, pero se pueden incluir distintas configuraciones personalizadas (e incluso borrar las predefinidas) mediante los ficheros de solución y de proyecto. En distintas configuraciones se puede, por ejemplo, definir distintos símbolos de compilación condicional que activen o desactiven partes del código, que permitan o prohíban la aparición de bloques de código *`unsafe`*, o que activen y desactiven distintas optimizaciones. Un ejemplo típico donde estas configuraciones resultan útiles es en aplicaciones multiplataforma: mediante símbolos de compilación condicional, en puntos concretos del código se puede llamar a un API o a otro dependiendo del Sistema Operativo o del hardware subyacente. Mediante configuraciones distintas se podrán activar o desactivar estos símbolos en el momento de compilación.

- `--runtime`: indica el *runtime* para el que se debe compilar la aplicación. .NET dispone de un catálogo[36] de RIDs (del inglés, *Runtime Identifiers*), que permiten indicar tanto el Sistema Operativo como la arquitectura del procesador. Por ejemplo, el RID win-x64 compila la aplicación para ejecutarse sobre Windows en procesadores AMD64, mientras que linux-arm64 lo hace para GNU/Linux en procesadores ARM de 64 bits. Además, los RIDs pueden ser generales (linux-x64 servirá típicamente para cualquier distribución GNU/Linux sobre AMD64) o específicos (rhel-x64 servirá para Red Hat Enterprise Linux). Estas diferencias se deben a cambios de bajo nivel entre versiones del Sistema Operativo que en algunas ocasiones pueden afectar a la aplicación, dependiendo de las APIs que utilicen. En este proyecto no se da esa particularidad.
- `--output`: no indica más que el directorio de salida de los binarios. En el caso de que no se especifique, por defecto los binarios se colocan en subdirectorios nombrados de acuerdo con el RID utilizado, de manera que no (suele) existir posibilidad de mezclar binarios para Sistemas Operativos y arquitecturas distintas. En caso de que esta mezcla se produzca por error, en el mejor de los casos la aplicación no se podrá ejecutar, y en el peor se podrá ejecutar, pero será inestable (pudiendo provocar pérdida de datos o corrupción de memoria, entre otros problemas).

Así pues, al igual que los comandos `test` y `build`, `publish` también tomará por defecto el fichero de solución existente en el directorio actual.

Actualmente se están explorando diversas técnicas, mediante el análisis estático de dependencias, para reducir el tamaño de las bibliotecas adjuntadas[37]. Sin embargo, una funcionalidad núcleo del propio lenguaje (y utilizada de manera extensiva en este proyecto) hace que eliminar código ejecutable de las bibliotecas basándose únicamente en un análisis estático sea peligroso: la reflexión, o la capacidad de

[36] https://docs.microsoft.com/en-us/dotnet/core/rid-catalog

[37] https://github.com/dotnet/core/blob/main/samples/linker-instructions.md

introspección del propio lenguaje. Si un tipo concreto es cargado y examinado mediante reflexión, el analizador de dependencias no lo encontrará. Lo marcará pues como no utilizado, y lo eliminará del binario resultante, rompiendo por completo la aplicación.

En cualquier caso, este enlazado temprano es configurable, y se pueden marcar tipos de manera manual para que no sean excluidos. Es una opción interesante si, además, se van a usar técnicas como el compilado antes de tiempo (AOT, *Ahead of Time*), que permite generar binarios nativos para la plataforma destino. No obstante, estas opciones no se han explorado en el presente Trabajo de Fin de Máster debido a, como ya se ha señalado antes, el extensivo uso que se hace de la reflexión.

4.4.2 Ejecución de las aplicaciones

Puede resultar extraño, si se examina la salida estándar adjuntada anteriormente, que todos los binarios generados (y no únicamente aquellos que son bibliotecas diseñadas para ser reutilizadas, como `rpc`) tengan extensión DLL. Cuando se compila mediante el comando `build`, la salida está preparada para ser ejecutada por el *runtime* de .NET (que ahora es una aplicación más y no una parte embebida en el propio Sistema Operativo, como sucedía en el caso de Windows y las aplicaciones .EXE).

Así pues, se generan DLLs independientemente del tipo de binario (biblioteca, aplicación de terminal, aplicación con GUI para Windows…). El *runtime* de .NET cargará dicha biblioteca, buscará su punto de entrada (típicamente, un método público y estático llamado `Main` que tenga como único argumento un *`array`* de *`string`*), y lo ejecutará.

Esto se hace mediante la aplicación `dotnet`, sin ningún subcomando:

```
$ dotnet ./server/bin/Debug/net5.0/server.dll
```

También se pueden pasar argumentos a la aplicación ejecutada especificándolo después de la DLL:

```
$ dotnet server.dll argumento_1 argumento_2
```

Si la aplicación por el contrario se ha compilado con el comando `publish`, en el directorio de salida, además de la DLL principal correspondiente, se encontrará un

ejecutable del mismo nombre (con extensión .EXE en Windows, sin extensión en sistemas UNIX-like).

Este ejecutable es un pequeño cargador. Cargará el *runtime*, con él la biblioteca principal, y comenzará su ejecución.

```
$ ls -l server*
-rwxr-xr-x  1 sluisp  staff   95244 Jul 25 17:36 server
-rw-r--r--  1 sluisp  staff   62429 Jul 25 17:36 server.deps.json
-rw-r--r--  1 sluisp  staff  123904 Jul 25 17:36 server.dll
-rw-r--r--  1 sluisp  staff   60464 Jul 25 17:36 server.pdb
-rw-r--r--  1 sluisp  staff     256 Jul 25 17:36 server.runtimeconfig.json

$ file server
server: Mach-O 64-bit executable x86_64

$ file server.dll
server.dll: PE32+ executable (console) x86-64 Mono/.Net assembly, for MS Windows

$ ./server argumento_1 argumento_2
```

Capítulo 5 - El orquestador

En los siguientes capítulos se detalla la pieza software principal implementada en este Trabajo de Fin de Máster: el servidor orquestador.

Para dar contexto sobre su funcionamiento, se comenzará reflejando un ejemplo de uso de principio a fin. A continuación, se listará la terminología que se usará en el resto de las secciones. Seguidamente se revisará la configuración necesaria para el funcionamiento del orquestador. Más adelante se introducirá el fichero de definición de trabajos para, a continuación, dar una explicación de cómo se deben implementar las fases vistas en este fichero de definición.

Una vez cubierta la parte externa al orquestador, se introducirán las partes más interesantes de su implementación, comenzando por la parte central (el motor de activación de fases), la gestión de usuarios y de sus secretos, y terminando con el API REST, que permite el manejo remoto del orquestador.

5.1 Sesión de ejemplo

El orquestador tiene dos modos de ejecución. El modo "servidor" (`runserver`) y el modo "automático" (`runstandalone`).

En el modo servidor el orquestador tiene habilitadas todas las funcionalidades. El API REST para operar el orquestador está disponible, así como la gestión de usuarios y de secretos. El API se introduce en la sección *El API REST*, y los usuarios y los secretos de usuario en la sección *Gestión de usuarios*.

Sin embargo, el modo automático existe para poder probar de manera rápida las fases. Al orquestador se le indica mediante línea de comandos las bibliotecas que ha de cargar y el fichero de definición de trabajos que tiene que ejecutar. Sobre la extensibilidad del orquestador mediante bibliotecas se profundiza en la sección *El motor de activación*, mientras que el fichero que define qué ha de ejecutarse se introduce en *El fichero de definición de trabajos*.

En el modo automático el progreso se reporta directamente a través de la línea de comandos:

```
> ./server runstandalone `
    --assembly ./externalphases.dll `
    --jobdefinition ./example.json

                   Build (Running) ──────────────────────────────────────   0% 00:00:07
      Deploy AWS machines (Running) ──────────────────────────────────────   0% 00:00:07
   Deploy Windows machine (Running) ──────────────────────────────────────   0% 00:00:07
 Deploying 156.119.187.74 (Running) ──────────────────────────────────────  64% 00:00:07
     Deploy Linux machine (Running) ──────────────────────────────────────   0% 00:00:07
Deploying 139.117.221.205 (Running) ──────────────────────────────────────  57% 00:00:07
```

En el modo servidor es necesario especificar un fichero de configuración, tal y como se explica en la sección *El fichero de configuración*. Entre otras cosas, este fichero define qué bibliotecas tiene que cargar el orquestador y el puerto donde el servidor HTTP escuchará las peticiones al API REST. Una vez inicializada, la aplicación escribe por salida estándar un log detallado de las operaciones que ejecuta:

```
> ./server runserver --appsettings ./appsettings.json

00:13:45 Program - Running with arguments: ['runserver', '--appsettings', './appsettings.json']
00:13:48 RunWebServer - Loading phases from assembly 'self'
00:13:48 PhaseCatalogue - Registering phases in assembly /Users/sluisp/Desktop/orchestrator/server.dll
00:13:48 PhaseCatalogue - Found phase compositePhase on type
Orchestrator.Server.Jobs.Phases.Predefined.CompositePhase
00:13:48 PhaseCatalogue - Registered entry point compositePhase.launchSequential on method LaunchSequential.
00:13:48 PhaseCatalogue - Registered entry point compositePhase.launchParallel on method LaunchParallel.
00:13:48 RunWebServer - Loading phases from assembly '/Users/sluisp/gitrepos/ucm-orchestrator-
tfm/src/externalphases/bin/Debug/net5.0/externalphases.dll'
00:13:48 PhaseCatalogue - Registering phases in assembly /Users/sluisp/gitrepos/ucm-orchestrator-
tfm/src/externalphases/bin/Debug/net5.0/externalphases.dll
00:13:48 PhaseCatalogue - Found phase awsDeployPhase on type ExternalPhases.AwsDeployPhase
00:13:48 PhaseCatalogue - Registered entry point awsDeployPhase.deployFromTemplate on method DeployFromTemplate.
00:13:48 PhaseCatalogue - Found phase awsShutdownPhase on type ExternalPhases.AwsShutdownPhase
00:13:48 PhaseCatalogue - Registered entry point awsShutdownPhase.shutdownFromIpAddress on method
ShutdownFromIpAddress.
00:13:49 RunWebServer - Binding to URLs [http://*:9999]
00:13:55 RunWebServer - The application does not have a SSL certificate configured. Trying to listen under HTTPS
will fail.
Hosting environment: Production
Content root path: /Users/sluisp/Desktop/orchestrator/
Now listening on: http://[::]:9999
Press ENTER to exit
```

El log puede servir para depurar que durante el arranque se encuentren y registran todas las fases necesarias (más sobre la búsqueda de fases en la sección *Carga de bibliotecas y descubrimiento de fases*).

Una vez el orquestador escribe "`Now listening on` http://[::]:9999" (o el protocolo y puertos que correspondan según la configuración), está listo para aceptar peticiones.

Para ello se puede usar tanto la aplicación cliente desarrollada como parte de este Trabajo de Fin de Máster como cualquier otra herramienta capaz de manejar HTTP.

La aplicación cliente está autodocumentada, así que se puede ejecutar sin comandos (o con la opción --help) para que muestre la ayuda de cada acción disponible:

```
> ./client --help

client

Usage:
  client [options] [command]

Options:
  --verbose                 Show verbose output
  --config <config>         Sets configuration file
  --skip-certificate-check  Skips orchestrator certificate checks
  --version                 Show version information
  -?, -h, --help            Show help and usage information

Commands:
  u, user  Manages users related data: users, profiles, and secrets
  j, job   Manages remote jobs
```

El primer paso para usar el orquestador consiste en crear una cuenta de usuario. Para ello se indica la dirección Web del orquestador, un nombre de usuario, una contraseña y el nombre del perfil bajo el que se quieren guardar los datos recibidos por el orquestador. Estos serán los *tokens* de acceso y de refresco, y la parte pública de la clave de cifrado que se usará para la gestión de secretos. Los *tokens* se introducen en la sección *Tokens de acceso, de refresco y de trabajos.*

```
> ./client user create `
    --server http://localhost:9999 `
    --user sergio --password "Th1s_P4ssw0rd!" `
    --profile demoprofile

- ⚠️ This is your public key.
- Losing this key will render your account useless!
- This key is already saved in your profile.
- Still, it is a good idea to copy it and store it safely.

─── BEGIN Public Key ──────────────────────────────────────────────────────
MIIBCgKCAQEA4bBBfFSv/pnL8tU6uz88IDZAb1M/KtvHkdhNiG0vQuIUdRh/DEIXbLRlwVQlwYQ7TK
BUe8T2VlZTVMHO2DACgC8pCEZoaQ6aS+UUpQi2Q2/AWSiDSvopFqePUJPUqT1B/m7TvHOn2JPoFBbT
YWk72jrr8pdVmUfsxku2YkSZDMG9T6j1Nr0ECf3rN3obRWbKOAKzuJb1WDNzre8bBtwmEC+qSNJoqc
Sf8ORjwlVBrkMxLrkCOwunVLotDQ82qDV3sMEPCUgkVr1isMcRamrQoAP8qKy4ZBa7OJu0c2sWYi22
bsSUlFNSiN2s1e6ie9lNZ7X8sz5ePqRlFgPrARiI8wIDAQAB
─── END Public Key ────────────────────────────────────────────────────────

Correctly created user and associated profile!
```

Dependiendo del Sistema Operativo utilizado, el fichero de perfil se guardará en un directorio u otro. En las siguientes operaciones contra el orquestador, únicamente es necesario especificar el nombre de perfil para que la aplicación cliente utilice los datos correctos para el usuario deseado.

Los secretos de usuario también se pueden manejar desde la aplicación cliente para crearlos…

```
> ./client user secrets write `
    --name "aws_secret_key" --value "k2E1buQaHV" `
    --profile demoprofile

Correctly wrote secret 'aws_secret_key'!
```

…listarlos…

```
> ./client user secrets list  --profile demoprofile

  Secret name
 ═══════════════
  aws_secret_key
```

…leerlos…

```
> ./client user secrets read  --profile demoprofile

  Secret name     │ Secret value
 ═════════════════╪══════════════
  aws_secret_key  │ k2E1buQaHV
```

…y borrarlos…

```
> ./client user secrets delete --name "aws_secret_key"  --profile demoprofile

Correctly deleted secret 'aws_secret_key'!
```

Los secretos, una vez escritos, pueden referenciarse desde el fichero de definición de trabajos para usar su valor como argumento.

Los trabajos se pueden igualmente lanzar desde la línea de comandos.

```
> ./client job run --jobfile ./example.json --profile demoprofile

Job launched! Id: '081371d2-f5a1-44ab-ab24-a58c65a78702'
```

El identificador de trabajo devuelto se puede obtener también listando los trabajos en ejecución.

```
> ./client job list --profile demoprofile
```

```
  Job name  | Job ID                                | Start Date UTC       | Status
============|=======================================|======================|=========
  Demo      | 081371d2-f5a1-44ab-ab24-a58c65a78702  | 09/01/2021 00:02:11  | running
```

El estado de un trabajo se puede consultar utilizando dicho identificador. El anidamiento en los progresos indica que una fase se engloba dentro de la superior.

```
> ./client job status --jobid 081371d2-f5a1-44ab-ab24-a58c65a78702 --profile demoprofile --verbose

Job ID: 081371d2-f5a1-44ab-ab24-a58c65a78702
Job Name: Demo
Start date UTC: 09/01/2021 00:02:11
Status: running
Progress:
 -- Build --
  Started at: 09/01/2021 00:02:11
  Status: running
  Progress: 0% (indeterminate: no)

  -- Deploy AWS machines --
   Started at: 09/01/2021 00:02:11
   Status: running
   Progress: 0% (indeterminate: no)

   -- Deploy Windows machine --
    Started at: 09/01/2021 00:02:11
    Status: running
    Progress: 0% (indeterminate: no)

    -- Deploying 127.131.104.76 --
     Started at: 09/01/2021 00:02:11
     Status: running
     Progress: 92% (indeterminate: no)

   -- Deploy Linux machine --
    Started at: 09/01/2021 00:02:11
    Status: running
    Progress: 0% (indeterminate: no)

    -- Deploying 67.181.176.2 --
     Started at: 09/01/2021 00:02:11
     Status: running
     Progress: 92% (indeterminate: no)
```

La aplicación cliente también es capaz de parar trabajos en marcha.

5.2 Terminología

En todo proyecto software surge una terminología propia, ligada a la lógica de negocio y a la implementación de la propia aplicación, que permite referirse a sus diversas partes sin necesidad de entrar en profundidad.

El idioma escogido para el desarrollo (nombres de clases, métodos y variables, pero también comentarios, documentación generada y mensajes de *commit* en el Control de Versiones) ha sido el inglés. Por lo tanto, parte de esta terminología ha surgido

en dicho idioma. Sin embargo, por consistencia, en la presente memoria se introducirá el nombre original y, a partir de ese momento, se utilizará la traducción al castellano.

- *Job* (trabajo): un conjunto de pasos ordenados, definidos por el usuario, que el orquestador ejecutará como si de una unidad indivisible se tratase. Deseablemente estos pasos llevarán a la consecución de un objetivo concreto, como por ejemplo generar una nueva versión de un producto software. Sin embargo, desde el punto de vista del orquestador, dicha utilidad no es relevante.
- *Job definition* (definición de trabajo): la representación de los pasos ordenados y de los datos de entrada que se requieren para la ejecución de un trabajo. En la implementación actual, esta definición de trabajo se representa mediante un fichero estructurado JSON. Sin embargo, dicho fichero no es más que interfaz de usuario – en el futuro podrían añadirse nuevas formas de introducir en el orquestador una definición de trabajo (se discute en la sección *Añadir una interfaz Web para el manejo del orquestador*).
- *Job phase* (fase): La implementación concreta de cada uno de los pasos de un trabajo. Estas fases son la unidad mínima que un desarrollador puede implementar en una extensión y cargar en el orquestador. Se detallan en la sección *Cómo implementar una fase*.
- *Job identifier* (identificador de trabajo): identificador alfanumérico que permite referirse a una ejecución concreta de un trabajo concreto de manera unívoca. Un usuario puede reutilizar un fichero de definición de trabajos tantas veces como desee. Cada una de esas ejecuciones tendrá su propio identificador de trabajo, independientemente de que el fichero de definición se haya utilizado con anterioridad. Este identificador será de utilidad para consultar el estado del trabajo en curso y para, en caso de ser necesario, cancelarlo.
- *Inputs* (entradas): cada uno de los datos de entrada que una fase necesita para ejecutarse. Las entradas permiten la parametrización de las

fases, y, por tanto, la parametrización de los trabajos formados por dichas fases. Existen entradas de primer nivel (aquellas que se definen en el propio fichero de definición de trabajo, que serán constantes, y a los que todas las fases de dicho trabajo tienen acceso) como de segundo nivel (aquellas que son generadas por la propia ejecución de alguna de las fases, y que solo serán accesibles por las fases que se ejecuten a continuación).

- *Outputs* (salidas): cada uno de los datos de salida que un trabajo genera como resultado accesorio de su ejecución. Las salidas de una fase pueden formar parte de las entradas de las fases que se ejecuten a continuación. Las salidas de la última fase más externa del trabajo son las salidas del trabajo completo.
- *User* (usuario): la persona que opera el orquestador. El orquestador únicamente admite la ejecución de trabajos en el contexto de un usuario. Esto podría permitir en el futuro funcionalidades tales como restringir el tipo de recursos a los que tiene acceso un usuario concreto, entre otras opciones que se discuten más adelante en la sección *Restringir la operativa de los usuarios*.
- *User secrets* (secretos de usuario): datos almacenados en el orquestador, cifrados mediante un sistema de clave pública/privada, que se pueden referenciar como entrada de los trabajos y sus fases. Estos secretos de usuario permiten a los usuarios almacenar en el propio orquestador aquella información sensible a la que el resto de los usuarios no deban tener acceso (como contraseñas) sin tener para ello que restringir el acceso a los ficheros de definición de trabajos.

5.3 El fichero de configuración

El servidor orquestador requiere de una determinada configuración para poder funcionar. Ya que el orquestador levanta un servidor HTTP de ASP.NET para su funcionamiento, y la operativa principal del orquestador se dispara a través de dicho

servidor (mediante el API REST), se ha utilizado para la configuración el formato estándar de ASP.NET – un fichero en formato JSON.

El nombre de este fichero JSON es, por defecto, `appsettings.json`, pero se puede indicar mediante parámetros en la línea de comandos, en el momento de ejecutar el orquestador, un fichero con un nombre diferente.

Si alguna configuración no se encuentra presente en el fichero, en el código existen valores por defecto. Por lo tanto, se puede evitar tener que configurar la mayoría de las opciones. Sin embargo, sí sería conveniente cambiar aquellas de las que dependen cosas tales como claves de cifrado.

Se repasan a continuación las claves de configuración junto con su tipo. Se ha de tener en cuenta que las configuraciones pueden estar representadas tanto por tipos primitivos (`int`, `string`, `bool`) como por tipos compuestos. Para los tipos compuestos se denotará primero el nombre de la configuración (*Nombre*) y luego cada una de sus propiedades usando la notación *Nombre.Propiedad*. Se puede encontrar un ejemplo completo del fichero de configuración en el apéndice *Ejemplo del fichero de configuración del orquestador*.

- `Urls` (*string[]*): cada una de las direcciones a través de las cuales el orquestador es accesible, incluyendo protocolo (HTTP o HTTPS), dirección IP y puerto. Si se especifica alguna URL con protocolo HTTPS, es necesario dar valores adecuados a las opciones bajo '`Certificate`' (ver a continuación). No tiene un valor por defecto. En su lugar, se proporcionan los siguientes ejemplos de los que poder extrapolar nuevos valores:
  - http://*:9999 > el orquestador escucha usando el protocolo HTTP en todas las direcciones IP disponibles, en el puerto 9999.
  - https://10.43.5.127:443 > el orquestador escucha usando el protocolo HTTPS en la dirección IP 10.43.5.127, en el puerto 443 (el puerto por defecto para HTTPS).
- `Certificate` (*CertificateSettings*): configuración relativa al certificado SSL necesario para que el orquestador escuche peticiones a través de HTTPS.
- `Certificate.UseCertificate` (*boolean*): indica si se debe o no usar un certificado SSL. Su valor por defecto es `'false'`. Cuando su valor es

`'false'`, la configuración indicada para la localización y contraseñas del certificado será ignorada. Si no se carga ningún certificado, la aparición de URLs con protocolo HTTPS en '`Urls`' evitarán que la aplicación arranque correctamente.

- `Certificate.CertificatePath` (*string*): localización en disco del fichero del certificado. Si el certificado existe, se intentará cargar. Si el certificado no existe, se generará uno autofirmado.
- `Certificate.CertificatePassword` (*string*): contraseña del fichero del certificado. Si el certificado indicado por `Certificate.CertificatePath` existe, se intentará cargar utilizando esta contraseña. Si el certificado no existe, cuando se genere uno autofirmado utilizará esta contraseña.
- `AssembliesToLoad` (*string[]*): una lista con la localización de las bibliotecas DLL a cargar como extensiones. La palabra reservada `'self'` indica que el orquestador debería examinarse a sí mismo en busca de fases, ya que este implementa algunas predefinidas. En vez de forzar por defecto al orquestador a realizar siempre esta búsqueda, se escogió el uso de la palabra reservada `'self'` para que el mecanismo de búsqueda de fases fuese uniforme (y para permitir saltar la búsqueda sobre el propio orquestador, en caso de que un usuario quiera cargar fases con los mismos nombres que las predefinidas).
- `ApplicationPaths` (*ApplicationPathsSettings*): las localizaciones en disco donde el orquestador va a crear las estructuras de directorios y de ficheros necesarias para su funcionamiento. Esto se hace a través de una opción en lugar de forzar a que sea un directorio predefinido para poder cambiar fácilmente los ficheros de los que se nutre el orquestador sin tener que cambiar su localización en disco, permitiendo (mediante dos ficheros de configuración diferentes) levantar dos instancias distintas con diferentes datos utilizando los mismos binarios.
- `ApplicationPaths.BasePath` (*string*): directorio base para todos los datos.
- `UserPolicy` (*UserPolicySettings*): especifica las distintas políticas que se deben cumplir en cuanto a los usuarios del orquestador.

- `UserPolicy.NameRegEx` (*string*): especifica la expresión regular que deben cumplir los nombres de usuario. Mediante una expresión regular se pueden especificar cosas tales como longitud mínima y máxima, y conjunto de caracteres permitido. Por defecto, solo permite caracteres alfanuméricos (a-z, A-Z, 0-9) y una longitud mínima de tres caracteres:

```
^[a-zA-Z0-9]{3,}$
```

- `UserPolicy.PasswordRegEx` (string): especifica la expresión regular que deben cumplir las contraseñas de usuario. Por defecto, obliga a un mínimo de ocho caracteres, conteniendo al menos una letra mayúscula, una minúscula, y un número.

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(.*){8,}$
```

- `SaltedPassword` (*SaltedPasswordSettings*): especifica los parámetros que se usan para almacenar las contraseñas de manera segura (más sobre esto en la sección *Almacenamiento seguro de la contraseña*). El cambio de alguna de estas opciones causará que todas las contraseñas almacenadas en el sistema dejen de ser válidas. Por lo tanto, **no se deberían cambiar una vez que el primer usuario se registre en el sistema**.
- `SaltedPassword.SaltLength` (*int*): la longitud del salto. Por defecto, 16.
- `SaltedPassword.PasswordLength` (*int*): la longitud del resumen (*hash*) de la contraseña. El valor por defecto es 20.
- `SaltedPassword.HashIterations` (*int*): el número de iteraciones que se hacen a la hora de calcular el resumen. El valor por defecto es 10000.
- `Jwt` (*JwtSettings*): especifica los parámetros que se utilizan para generar los *tokens* de acceso al API REST (*Json Web Tokens*) de manera segura. Más sobre esto en la sección *Tokens de acceso, de refresco y de trabajos*. El cambio de alguna de estas opciones causará que todos los *tokens* ya generados pasen a ser inválidos. Si bien no prevendrá el acceso al sistema por parte de los usuarios, el invalidar tanto el token de acceso como el de refresco forzará a que todos los usuarios deban renovar sus *tokens* mediante su contraseña.

- `Jwt.SecretKey` (*string*): la clave de cifrado simétrica con la que se firman los *tokens*. Su valor por defecto debería ser cambiado siempre, por lo que no se especifica aquí.
- `Jwt.AccessTokenLifeTime` (*TimeSpan*): tiempo de vida del token de acceso. Una vez el token de acceso ha caducado, es necesario obtener uno nuevo. El valor por defecto es de 7 días.

```
"accessTokenLifeTime": "07.00:00:00.000"
```

- `Jwt.RefreshTokenLifeTime` (*TimeSpan*): tiempo de vida del token de refresco. El token de refresco se utiliza para obtener un nuevo par de *tokens* (nuevo *token* de acceso y nuevo *token* de refresco). El valor por defecto es de 30 días.

```
"refreshTokenLifeTime": "30.00:00:00.000"
```

- `Jwt.JobTokenLifeTime` (*TimeSpan*): tiempo de vida del token de trabajo. Este *token* se genera en el momento de ejecutar un trabajo, y permite a las distintas fases que forman dicho trabajo suplantar al usuario para operaciones tales como descifrar secretos. El valor por defecto es de un día, lo que significa que si se ejecuta un trabajo que dure más tiempo, a partir de las 24 horas después del comienzo de ejecución perderá acceso a los secretos del usuario.

```
"jobTokenLifeTime": "01.00:00:00.000"
```

- `Jwt.LifeTimeCheckPrecission` (*TimeSpan*): precisión máxima que se utilizará a la hora de comprobar las fechas de un *token* con la hora del sistema. Más detalles sobre por qué es necesario especificar esta precisión en la sección *Tokens de acceso, de refresco y de trabajos*. El valor por defecto es de un segundo.

```
"lifeTimeCheckPrecission": "00.00:00:01.000"
```

- `Encryption` (*EncryptionSettings*): especifica los parámetros que se utilizan para la gestión de claves asimétricas. De momento las claves asimétricas se utilizan únicamente para la gestión de secretos de usuario.

- `Encryption.KeyLength` (*int*): tamaño del par de claves RSA que se generarán, en bits. Por defecto su valor es de 2048.
- `ApiMessages` (*ApiMessagesSettings*): especifica los mensajes que son devueltos por el API REST en condiciones de fallo. Estos mensajes forman parte de la configuración para poder ser cambiados (y traducidos, llegado el momento) de manera sencilla sin necesitar reconstruir la aplicación.
- `ApiCodes` (*ApiCodesSettings*): especifica los códigos de resultado que son devueltos por el API REST. Estos códigos forman parte de la configuración para poder ser cambiados y accedidos, a modo de catálogo, de manera sencilla (aunque, una vez definidos y documentados, su significado no debería cambiar).

5.4 El fichero de definición de trabajos

Al igual que el fichero de configuración, el fichero de definición de trabajos es un fichero en texto plano, en formato JSON. Sin embargo, al contrario que el fichero de configuración (que viene determinado por las capacidades de ASP.NET) no tiene por qué ir ligado a un formato concreto, y ni tan siquiera ser un fichero. El fichero de definición de trabajos no es pues más que mera interfaz de usuario, escogida en este formato concreto por sencillez y coherencia con el resto de los ficheros.

El nodo principal consta de tres nodos hijos: el nombre del trabajo, las entradas para dicho trabajo, y la fase raíz:

```
{
  "jobName": <string>,
  "inputs": <dictionary>,
  "rootPhase": <phaseDefinition>
}
```

El nombre del trabajo no es más que un nombre con el que se pueda identificar fácilmente el propósito de los trabajos en ejecución. En el futuro se podría completar con información extra, como se detalla en la sección *Añadir información extra a los trabajos*.

```
{
  "jobName": "Build and deploy MyAwesomeApplication for Android / Play Store"
```

```
  ...
  ...
}
```

Las entradas consisten en un diccionario donde las claves son *strings* y los valores son o bien objetos primitivos o bien objetos compuestos. En las entradas de un trabajo se puede hacer referencia a los secretos de usuario mediante la notación `${secret:NAME}`. Adicionalmente, en las entradas de una fase, se puede hacer referencia a las entradas del trabajo o a las salidas de fases previas mediante la notación `${NAME}`.

```
{
  ...
  "inputs": {
    "awsAccessKeyId": "${secret:aws_access_key}",
    "awsSecretKeyId": "${secret:aws_secret_key}",
    "awsRegion": "useast4",
    "deployTimeout": "00.00:00:20.000",
    "linuxTemplatePath": "/opt/templates/linux.yaml",
    "windowsTemplatePath": "/opt/templates/windows.yaml",
    "launchParallelism": 2,
    "shutdownParallelism": 5
  },
  ...
}
```

La fase raíz es aquella que se ejecutará durante el trabajo. Es por lo tanto esperable que, si el trabajo consiste en más de una fase, la fase raíz permita el anidamiento de sub-fases. Para ello hay una fase predefinida, la `compositePhase`, que permiten lanzar sub-fases de manera secuencial (punto de entrada `launchSequential`) o paralela (punto de entrada `launchParallel`). Las entradas que recibe esta fase en sus dos puntos de entrada son, pues, la definición de otras fases, además del *token* de cancelación.

Todas las fases han de tener, no obstante, las siguientes propiedades: un nombre, el identificador de la fase, el identificador del punto de entrada de la fase, y las salidas que exportan a fases siguientes. Esta información está reflejada en el modelo `JobPhaseDefinition`.

```
{
  ...
  ...
  "rootPhase": {
    "name": "Deploy AWS Linux machine",
```

```
    "phaseId": "awsDeployPhase",
    "phaseEntryPointId": "deployFromTemplate",
    "inputs": {
      "accessKey": "${awsAccessKeyId}",
      "secretKey": "${awsSecretKeyId}",
      "region": "${awsRegion}",
      "template": "${linuxTemplatePath}",
      "timeout": "${deployTimeout}"
    },
    "outputs": {
      "publicMachineIp": "linuxMachineIp"
    }
  }
}
```

Las salidas que exporta una fase pueden parecer, a priori, confusas. Es un diccionario donde tanto el valor como la clave son cadenas de caracteres. Esto es debido a que, como ya se ha hecho hincapié, las fases son el mínimo bloque de construcción de trabajos, y son reutilizables. Por lo tanto, el resultado de una fase siempre tendrá el mismo nombre (en este caso, `'publicMachineIp'`), y, por lo tanto, para evitar colisiones de nombres, es necesario realizar un renombrado (en este caso, `'linuxMachineIp'`). Esta situación se ve más claramente con el siguiente ejemplo, considerando que ambas fases están anidadas de tal forma que se ejecuten en paralelo:

```
...
[
  {
    "name": "Deploy AWS Linux machine",
    "phaseId": "awsDeployPhase",
    "phaseEntryPointId": "deployFromTemplate",
    "inputs": {
      "accessKey": "${awsAccessKeyId}",
      "secretKey": "${awsSecretKeyId}",
      "region": "${awsRegion}",
      "template": "${linuxTemplatePath}",
      "timeout": "${deployTimeout}"
    },
    "outputs": {
      "publicMachineIp": "linuxMachineIp"
    }
  },
  {
    "name": "Deploy AWS Windows machine",
    "phaseId": "awsDeployPhase",
    "phaseEntryPointId": "deployFromTemplate",
    "inputs": {
      "accessKey": "${awsAccessKeyId}",
      "secretKey": "${awsSecretKeyId}",
```

```
      "region": "${awsRegion}",
      "template": "${windowsTemplatePath}",
      "timeout": "${deployTimeout}"
    },
    "outputs": {
      "publicMachineIp": "windowsMachineIp"
    }
  }
]
```

La fase y el punto de entrada de fase en ambos casos es exactamente la misma. Sin embargo, mediante parametrización, cada instancia tendrá un propósito diferente: la primera creará una máquina Linux, mientras que la segunda creará una máquina Windows. Como el código que se está ejecutando en ambas es el mismo, el nombre de la salida (determinada pues por el código) será el mismo, '`publicMachineIp`'. Sin embargo, el renombrado permite diferenciar en las entradas de fases posteriores estas salidas:

```
...
[
  {
    "name": "Shutdown AWS Linux machine",
    "phaseId": "awsShutdownPhase",
    "phaseEntryPointId": "shutdownFromIp",
    "inputs": {
      "accessKey": "${awsAccessKeyId}",
      "secretKey": "${awsSecretKeyId}",
      "region": "${awsRegion}",
      "machineIpAddress": "${linuxMachineIp}"
    },
    "outputs": {}
  },
  {
    "name": "Shutdown AWS Windows machine",
    "phaseId": "awsShutdownPhase",
    "phaseEntryPointId": "shutdownFromIp",
    "inputs": {
      "accessKey": "${awsAccessKeyId}",
      "secretKey": "${awsSecretKeyId}",
      "region": "${awsRegion}",
      "template": "${windowsMachineIp}"
    }
  }
]
```

Si el código de una fase no exporta ninguna salida (o no se desea que la exporte, aunque el código sí esté preparado para ello), el diccionario '`outputs`' puede aparecer vacío o incluso no especificarse.

5.5 Cómo implementar una fase

Una vez visto el fichero de definición de trabajos, resulta más sencillo describir cómo implementar a código cada una de las fases que forman un trabajo. La mayoría de los conceptos que se presentan a continuación ya han sido introducidos.

5.5.1 Dependencias de las fases

Las fases no se instancian explícitamente, sino que son instanciadas en el momento de su ejecución (de manera perezosa) por el motor de activación. Más sobre esto en la sección *El motor de activación*.

Por lo tanto, las dependencias que puede tener la instancia de una fase de instancias de otros tipos son limitadas (más allá de las que la propia fase pueda instanciar por sí misma), y viene determinada por las capacidades de inyección de dependencias del motor de activación. Como trabajo futuro se plantea la necesidad de una inyección de dependencias mejor, tal y como se discute en la sección *Inyección de dependencias de terceros*.

Por el momento, una fase puede definir únicamente un constructor, y dicho constructor puede definir únicamente parámetros de los siguientes tipos:

- `IPhaseContext`: es el contexto de ejecución de una fase. Se describirá a continuación.
- `ISystemDate`: es un envoltorio en torno a `DateTime`. Permite falsear la fecha actual del sistema, y es útil para probar que el código se comporta como se espera bajo determinadas circunstancias (cambios de fecha, largos tiempos de ejecución) de manera sencilla.

5.5.2 Los atributos PhaseId y PhaseEntryPointId

Los conceptos de `PhaseId` y `PhaseEntryPointId` ya fueron introducidos en el fichero de definición de trabajos. Utilizados en conjunto sirven para identificar de forma unívoca un método concreto a ejecutar por el orquestador.

En vez de atributos podría haberse utilizado el nombre completamente cualificado del método (`Namespace.ClassName.MethodName`). Sin embargo, esto podría

causar nombres innecesariamente largos, poco explicativos, y ataría los ficheros de definición de trabajos a detalles muy concretos de la implementación. Además, el hecho de utilizar un atributo ayuda, desde el punto de vista del programador, a diferenciar para una fase qué es API pública y qué es API privada sin necesidad de saber cómo están utilizando dicha fase los usuarios – en cualquier momento se podría renombrar el método o las clases que implementan una fase concreta sin romper la compatibilidad con ningún fichero de definición de trabajos que pueda existir.

Ambos atributos se definen en el espacio de nombres `Orchestrator.Server.Jobs.Common`. El atributo `PhaseIdAttribute` puede decorar únicamente clases, mientras que `PhaseEntryPointIdAttribute` puede decorar solo métodos.

Revisitando el ejemplo visto en la sección *El fichero de definición de trabajos*, donde se tenía una fase con `'phaseId'` igual a `'awsDeployPhase'` y un `'phaseEntryPointId'` igual a `'deployFromTemplate'`, esta sería una posible implementación:

```
using Orchestrator.Server.Jobs.Common;

[PhaseId("awsDeployPhase")]
public class AwsReployPhase
{
  public AwsDeployPhase(IPhaseContext ctx, ISystemDate systemDate)
  {
    this._context = ctx;
    this._systemDate = systemDate;
  }

  [PhaseEntryPointId("deployFromTemplate")]
  public async Task<PhaseResult> DeployFromTemplate(
    string accessKey,
    string secretKey,
    string region,
    string template,
    CancellationToken ct)
  {
    // Method code…
  }

  readonly IPhaseContext _context;
  readonly ISystemDate _systemDate;
}
```

En el ejemplo, tanto la clase como el método se llaman igual que el atributo que los decora. Sin embargo, como ya se ha señalado anteriormente, esto no es un requisito.

Además, en el ejemplo se puede ver que el método decorado con el atributo `PhaseEntryPointId` tiene las siguientes particularidades:

- Los nombres de los parámetros son los mismos que los nombres de las claves en el diccionario de entradas del fichero de definición de trabajos:
  - Como se detalla en la sección *Activación de fases*, los nombres de los parámetros en el código se utilizan para el emparejamiento con las entradas.
  - Se propone una alternativa en la sección *Permitir la ofuscación de las fases*.
- El método está marcado como `async`, tiene un parámetro de tipo `CancellationToken` que no aparece en el diccionario de entradas, y devuelve un objeto `Task<PhaseResult>`.
  - Se profundiza sobre este aspecto en la sección *El API asíncrono y el token de cancelación*.

5.5.3 El API asíncrono y el token de cancelación

Como se detalló en la sección *Modelo de programación asíncrona sencillo*, uno de los motivos para escoger C# / .NET como tecnología para desarrollar este proyecto es su API asíncrono. Este se implementa mediante los siguientes tres elementos:

El primer elemento es la palabra reservada `'async'` en la signatura del método, que le indica al compilador que es un método asíncrono. En realidad, el uso de la palabra `'async'` es obligatorio únicamente si dentro del cuerpo de la función se va a ejecutar un `'await'`. Estas dos palabras reservadas en conjunto permiten al compilador y al *runtime* escoger en qué hilo se ejecutarán tanto la llamada asíncrona como el código que se encuentra a continuación.

Adentrarse en la complejidad del funcionamiento interno del API asíncrono requeriría un capítulo en sí mismo y queda fuera del alcance de esta memoria.

Sin embargo, se puede detallar lo siguiente: el mero uso de las palabras clave `async/await` no provocan que el *runtime* asigne la ejecución de un método a un hilo distinto. Si se quiere que la ejecución de un método comience inmediatamente en un hilo distinto a aquel desde el que se está haciendo la llamada, es necesario usar el API `Task.Run`.

```
public async Task RunExample()
{
  await Example_1();
  await Example_2();
}

public async Task Example_1()
{
  Console.WriteLine("This runs on a thread...");
  await Task.Delay(TimeSpan.FromSeconds(1));
  Console.WriteLine("This continuation code runs on a differente thread");
}


public async Task Example_2()
{
  Console.WriteLine("This runs on a thread...");
  await Task.CompletedTask;
  Console.WriteLine("This continuation code runs on exactly the same thread");
}
```

En el método `Example_1`, la tarea disparada por `Task.Delay` no está completada en el momento de ejecutar el `await` (esta tarea consiste en un retraso de un segundo, algo parecido a un `Thread.Sleep`). Por lo tanto, el hilo en el que se ejecuta el código previo al `await` (el primer `Console.WriteLine`) y el código que se ejecuta después del `await` (el segundo `Console.WriteLine`) se ejecutarán en hilos distintos.

Sin embargo, en el método `Example_2`, la tarea ya está completada en el momento de ejecutar el `await`. Por lo tanto, no es necesario ejecutar la continuación en un hilo distinto, por lo que el cuerpo completo del método `Example_2` se ejecutará en el mismo hilo.

En realidad, como se ve en el segundo ejemplo de `await`, el tipo sobre el que `await` funciona es sobre el retorno de una función asíncrona, que será una tarea o `Task<T>`. Eso permite hacer cosas como esta:

```
public async Task RunExample()
{
```

```
  Task<Result> apiResultTask = DoSomeHeavyApiCall();

  // Let's get some other things ready...
  var localResult = SomeLocalWork();
  var someOtherLocalResult = SomeOtherLocalWork();

  // Okay, now we REALLY need the result
  Result apiResult = await apiResultTask;
  ComputeResults(apiResult, localWork, someOtherLocalResult);
}
```

La llamada al API se empieza a computar en un hilo distinto al del método que hace la llamada tan pronto como `DoSomeHeavyApiCall` se encuentre bloqueado en una llamada que implique I/O – típicamente, una operación que involucre o bien al disco, o bien a la red.

En ese momento, continúa el trabajo del método inicial, ejecutando las llamadas a `SomeLocalWork` y `SomeOtherLocalWork`. Cuando llega el momento en el que no se puede retrasar más obtener el resultado de la llamada que se ha quedado esperando por I/O, se hace `await` a la tarea correspondiente, que puede ser instantánea si el resultado ya está disponible, o quedarse bloqueada hasta que lo esté. Después de ese `await`, el resultado ya se puede utilizar como si de una variable local más se tratase.

El segundo elemento es el tipo de retorno, que, como ya hemos visto, permite al código que hace la llamada interactuar con la tarea y con su resultado: `Task<T>`. También existe un tipo no genérico para representar a las tareas, `Task`. Este tipo no genérico debe usarse cuando el método asíncrono no tiene ningún valor de retorno, es decir, aquellos que de no ser asíncronos estarían marcados como `'void'`. Esto es realmente importante – si un método asíncrono con tipo de retorno `void` falla, el código cliente pierde control sobre este error. La llamada asíncrona fallará silenciosamente, y no será posible recuperar una traza. Sin embargo, si el método asíncrono devuelve `Task`, en caso de error, el código cliente lo podrá manejar adecuadamente:

```
public async void BadExample()
{
  // Some code that fails..
  throw new Exception("Something bad happened!");
}

public async Task GoodExample()
{
  // Some code that fails..
```

```
  throw new Exception("Something bad happened!");
}

public static void Main(string[] args)
{
  try
  {
    BadExample();
  }
  catch (Exception ex)
  {
    // This catch block never runs
  }

  try
  {
    await GoodExample();
  }
  catch (Exception ex)
  {
    // Here we can capture the exception!
  }
}
```

El último de los tres elementos es el *token* de cancelación, que, según las convenciones del lenguaje, en caso de aparecer debe ser el último parámetro de la llamada. Dicho *token* permite la parada ordenada de las tareas.

En .NET Core / .NET 5 se eliminó del API de hilos la posibilidad de abortarlos. Era ya considerado antes una mala práctica, pues en el momento de abortar un hilo no se sabe qué código se ha llegado a ejecutar, y, por lo tanto, abortar el hilo puede dejar la aplicación en un estado indeterminado desde el cuál sea imposible la recuperación. Sin embargo, en .NET Core y .NET 5, la posibilidad de abortar hilos directamente no existe. Si se quiere parar una tarea, se ha de hacer mediante el *token* de cancelación.

Utilizar el *token* de cancelación para cancelar una tarea implica una cooperación explícita entre el código cliente (que es quien quiere cancelar la tarea) y la propia tarea a ser cancelada. Cuando el código de la tarea detecta que esta ha de cancelarse (comprobando el estado del *token* de cancelación), tiene dos opciones para terminar su ejecución: salir (mediante un `return` con el valor de retorno que sea menester), o lanzando una excepción del tipo `OperationCanceledException`.

Ambas opciones tienen implicaciones respecto al estado final de la tarea y a cómo debe implementarse el código cliente: si la tarea simplemente retorna, su estado

en la `Task` que lo representa terminará siendo `TaskStatus.RanToCompletion`, lo que indica que la tarea se ejecutó hasta que terminó.

Sin embargo, si se lanza la excepción `OperationCanceledException`, el estado de la `Task` correspondiente pasará a ser `TaskStatus.Canceled`. Además, cancelar la tarea mediante una excepción implica que hacer `await` sobre la `Task` que la representa re-lanzará dicha excepción, y será necesario manejarla (o dejar que flote hasta que acabe con la ejecución de la aplicación, si es lo que se desea). La excepción `OperationCanceledException` puede lanzarse utilizando el propio *token*, llamado a `token.ThrowIfCancellationRequest`.

Dependiendo del modelo que se haya escogido para cancelar tareas, es importante tener cuidado con cómo se usa el *token* de cancelación. **Esto es de especial relevancia para la implementación del orquestador**. Se puede ver por qué con el siguiente ejemplo:

```
public static async Task PrintTimeEachSecondAsync(CancellationToken ct)
{
  while (!ct.IsCancellationRequested)
  {
    Console.WriteLine(DateTime.Now.ToString("HH:mm:ss.fff"));
    await Task.Delay(TimeSpan.FromSeconds(1), ct);
  }
}
```

Si la cancelación del *token* sucede **antes** de la comprobación en el bucle `while`, este bucle terminará, la tarea saldrá de manera normal, y el estado de la `Task` será `TaskStatus.RanToCompletion`. Sin embargo, el *token* que le llega al método está siendo reutilizado para pasárselo a `Task.Delay`. Si durante la espera configurada de un segundo el *token* se cancela, `Task.Delay` **SÍ lanzará una excepción**, que el método no está manejando, y que, por tanto, causará que el estado de la `Task` una vez termine sea `TaskStatus.Canceled`.

En el orquestador **no se desea dicho comportamiento**, pues las capas superiores no esperan que una tarea sea cancelada de esta manera. Por lo tanto, a la hora de implementar una fase del orquestador, si se necesita pasar un *token* de cancelación a tareas cuya implementación no se controla, lo deseable es hacerlo enlazando *tokens*.

Enlazar *tokens* permite además un grano más fino en el control del ciclo de vida de las tareas subyacentes. Si se tiene un *token* A, y se crea un *token* B enlazado a este, cancelar A resulta también en la cancelación de B, pero cancelar B no resulta en la cancelación de A.

De esta manera, podemos cancelar una sub-tarea sin que por ello se vea afectada la tarea al completo:

```
public static async Task PrintTimeEachSecondAsync(CancellationToken ct)
{
  CancellationTokenSource cts =
    CancellationTokenSource.CreateLinkedTokenSource(ct);

  while (!ct.IsCancellationRequested)
  {
    Console.WriteLine(DateTime.Now.ToString("HH:mm:ss.fff"));

    try
    {
      await Task.Delay(TimeSpan.FromSeconds(1), cts.Token);
    }
    catch (TaskCanceledException) { }
  }
}
```

5.5.4 La clase PhaseResult

Todas las fases han de comunicar resultados a las fases superiores (esto es, aquella fase que la haya invocado), y al propio orquestador.

Este retorno de resultados se hace a través de la clase `PhaseResult`. Tiene los siguientes miembros:

- `ExitReason`: es un enumerado que indica cuál es el motivo por el que la fase terminó. Permite a las fases superiores saber con exactitud si debería continuar la ejecución (y de qué forma) sin tener que examinar el estado de la tarea asociada. Tiene tres posibles valores:
  - `Success`: indica éxito. La fase se ejecutó hasta el final de una manera satisfactoria.
  - `Failure`: indica fallo. La fase falló de alguna manera, y ahora corresponde a la fase madre (en caso de haberla) actuar en consecuencia.

    - `Cancellation`: indica cancelación. A lo largo de su ejecución la tarea detectó que el token de cancelación fue activado, y salió correctamente, liberando, en caso de ser necesario, los recursos asociados.

- `Outputs`: es un diccionario (`Dictionary<string, object>`) opcional donde la fase deja sus salidas para que puedan servir de entradas a las fases que se ejecutan a continuación.
    - El valor de las claves del diccionario `Outputs` debe estar correctamente documentado, para que cualquier usuario del sistema sea capaz de reflejar en el fichero de definición de trabajo qué salidas quiere renombrar para exportarlas.

Es necesario remarcar que cuando una tarea retorna como `ExitReason` los valores `Success` o `Failure`, actuar en consecuencia significa sencillamente actuar de acuerdo con las intenciones del programador. Póngase por ejemplo una fase compuesta que activa dos fases hijas. La primera fase se dedica a reservar unos recursos, y la segunda fase a utilizarlos. Si la primera fase devuelve un fallo, significaría que dichos recursos no se han podido reservar, y, por lo tanto, no se puede ejecutar la segunda fase.

Sin embargo, este comportamiento puede no ser siempre el deseado. Si por ejemplo hay dos maneras de reservar dichos recursos (por ejemplo, con credenciales distintas, o desde servicios diferentes), el desarrollador podría querer implementar un mecanismo de recuperación en caso de que la primera fase en ejecutarse falle al reservarlos, activando la segunda fase (que usará esas credenciales distintas, o ese otro servicio) únicamente en ese escenario de fallo de la primera fase.

El diccionario `Outputs` es opcional. Por lo tanto, para evitar al desarrollador verbosidad a la hora de programar sus fases, la clase `PhaseResult` tiene tres miembros estáticos de solo lectura que puede utilizar para retornar un resultado sin salidas:

- `PhaseResult.SimpleSuccess`
- `PhaseResult.SimpleFailure`
- `PhaseResult.SimpleCancellation`

A modo de ejemplo, esta sería una implementación válida del método de entrada de una fase con los elementos vistos hasta ahora. El método está decorado con el atributo `EntryPointId`, se hace uso de la clase `PhaseResult` para reportar el resultado y se comprueba el *token* de cancelación cuando es oportuno:

```
[PhaseEntryPointId("deployFromTemplate")]
public async Task<PhaseResult> DeployAwsMachine(
  string accessKey,
  string secretKey,
  string region,
  string template,
  TimeSpan deployTimeout,
  CancellationToken ct)
{
  AwsApiClient awsApiClient = new(accessKey, secretKey, region);

  if (ct.IsCancellationRequested)
    return PhaseResult.SimpleCancellation;

  CancellationTokenSource cts =
    CancellationTokenSource.CreateLinkedTokenSource(ct);
  cts.CancelAfter(deployTimeout);

  try
  {
    DeployResult result = await awsApiClient.DeployAsync(template, cts.Token);
  }
  catch (TaskCanceledException ex)
  {
    // Log the exception
    return PhaseResult.SimpleCancellation;
  }
  catch (Exception ex)
  {
    // Log the exception
    return new PhaseResult(
      ExitReason.Failure,
      new Dictionary<string, object>
      {
        { "errorMessage", ex.Message }
      });
  }

  return new PhaseResult(
    ExitReason.Success,
    new Dictionary<string, object>
    {
      { "machinePublicIp", result.IpAddress.ToString() }
    });
}
```

5.5.5 Cómo reportar el progreso a las capas superiores

Una parte fundamental del orquestador es la capacidad de las fases para reportar hacia el exterior en qué punto de la ejecución se encuentran. Sin esta opción, el orquestador sería una caja negra que ejecutaría código sin ningún tipo de retroalimentación al usuario.

En la sección *Dependencias de las fases* se introdujo `PhaseContext`, y se describió como "*el contexto de ejecución de una fase*". Una de las partes de este contexto es el progreso. El resto de las partes se describirán en secciones posteriores.

El progreso viene representado por la clase `PhaseProgress`. Tiene las siguientes propiedades públicas:

- `StartDateUtc`: de tipo `DateTime`, indica el momento preciso en el que el activador de fases (más sobre esto en la sección *Activación de fases*) activó dicha fase.
- `Description`: de tipo `string`, es el nombre dado a la fase en el fichero de definición de trabajos.
- `IsIndeterminate`: de tipo `boolean`, indica si el progreso reportado es de tipo indeterminado (es decir, no se puede dar un porcentaje de completitud) o no lo es.
- `IsFinished`: de tipo `boolean`, indica si la tarea representada por el `PhaseProgress` terminó.
- `CurrentStatus`: de tipo `PhaseProgress.Status`, indica con más detalle que `IsFinished` el estado del progreso:
    - `PhaseProgress.Status.Unknown` indica un estado desconocido. Este estado desconocido se da cuando el progreso se creó, pero la actividad relacionada no comenzó todavía. Existe para evitar condiciones de carrera al preguntar el progreso de un trabajo mientras se está activando una fase y enlazando su contexto, para evitar tener que proteger la activación mediante interbloqueos.

    - `PhaseProgress.Status.Running` indica que la actividad asociada al progreso se está ejecutando.
    - `PhaseProgress.Status.FinishedFailed` indica que la actividad asociada al progreso finalizó su ejecución y además falló.
    - `PhaseProgress.Status.FinishedCancelled` indica que la actividad asociada al progreso finalizó su ejecución y además fue por cancelación.
    - `PhaseProgress.Status.FinishedSucceeded` indica que la actividad asociada al progreso finalizó su ejecución y además lo hizo con éxito.
- `ProgressPercent`: de tipo `float`, indica el avance de la tarea asociada al progreso con un valor numérico que va entre `0.0f` y `1.0f`, siendo 0 el valor más bajo (todavía no se ha avanzado nada) y 1 el valor más alto (la tarea ya ha finalizado su ejecución).
- `Children`: de tipo `List<PhaseProgress>`, son los sub-progresos enlazados a un proceso padre.

La clase `PhaseProgress` se estructura en forma de grafo acíclico dirigido. La navegación entre progresos se permite pues únicamente desde el nodo raíz hacia las hojas. El nodo raíz va a representar siempre el progreso del trabajo completo, mientras que el único progreso hijo de este nodo raíz será el progreso de la fase raíz. Este no es más que un detalle de implementación que se puede ignorar, durante el uso normal.

No obstante, lo importante de cara al desarrollador es que un progreso puede tener sub-progresos. Cada vez que una fase activa una sub-fase, se hace a través del contexto de activación (`PhaseContext`) de la fase madre, y el orquestador crea automáticamente un progreso enlazado al de esta para crear el contexto de activación de la fase hija.

Pero el mecanismo que utiliza el motor de activación para crear progresos hijos no tiene su uso restringido al propio orquestador. Cualquier fase puede hacer uso de ello para dar un mayor detalle de qué se está haciendo. Lo único de lo que debe

preocuparse el programador (para no confundir al usuario, en realidad, ya que no es un requisito estricto) es marcar los progresos adicionales creados como completos una vez que la ejecución de la tarea que representan ha terminado.

En el siguiente ejemplo, la fase cuenta desde cero hasta un determinado número, y luego al revés. En cada conteo que hace, actualiza el progreso que engloba a la fase concreta (`mPhaseContext.Progress`). Además, crea dos progresos hijos, uno para la cuenta hacia adelante (`countForwardProgress`) y otro para la cuenta hacia atrás (`countBackwardsProgress`). Actualiza cada uno de ellos en el bucle que corresponde, y al final de dicho bucle, los marca como completados con éxito.

Adicionalmente, en cada iteración de cada bucle, comprueba el estado del *token* de cancelación. Si la tarea ha sido cancelada, marca como finalizados por cancelación los progresos activos en ese momento (el global y `countForwardProgress` en el primer bucle, el global y `countBackwardsProgress` en el segundo bucle) y devuelve el `PhaseResult` adecuado.

```
[PhaseId("countNumbers")]
public class CountNumbersPhase
{
  public CountNumbersPhase(PhaseContext pc, ISystemDate sd)
  {
    mPhaseContext = pc;
    mSystemDate = sd;
  }

  [PhaseEntryPointId("countForwardsAndBackwards")]
  public async Task<PhaseResult> CountForwardsAndBackwards(
    int limit, CancellationToken ct)
  {
    float countProgressStep = 1.0f / limit;
    float totalProgressStep = 1.0f / (2 * limit);

    PhaseProgress countForwardProgress =
      mPhaseContext.Progress.CreateChild(
        mSystemDate.UtcNow(),
        "Counting forward");

    for (int i = 0; i < limit; i++)
    {
      mPhaseContext.Progress.IncrementProgress(totalProgressStep);
      countForwardProgress.IncrementProgress(countProgressStep);

      if (ct.IsCancellationRequested)
      {
        mPhaseContext.Progress.UpdateCurrentStatus(
```

```
          PhaseProgress.Status.FinishedCancelled);

        countForwardProgress.UpdateCurrentStatus(
          PhaseProgress.Status.FinishedCancelled);

        return PhaseResult.SimpleCancellation;
      }
    }
    countForwardProgress.UpdateCurrentStatus(
      PhaseProgress.Status.FinishedSucceeded);

    PhaseProgress countBackwardsProgress =
      mPhaseContext.Progress.CreateChild(
        mSystemDate.UtcNow(),
        "Counting backwards");

    for (int i = limit; i > 0; i--)
    {
      mPhaseContext.Progress.IncrementProgress(totalProgressStep);
      countBackwardsProgress.IncrementProgress(countProgressStep);

      if (ct.IsCancellationRequested)
      {
        mPhaseContext.Progress.UpdateCurrentStatus(
          PhaseProgress.Status.FinishedCancelled);

        countForwardProgress.UpdateCurrentStatus(
          PhaseProgress.Status.FinishedCancelled);

        return PhaseResult.SimpleCancellation;
      }
    }

    countBackwardsProgress.UpdateCurrentStatus(
      PhaseProgress.Status.FinishedSucceeded);
    mPhaseContext.Progress.UpdateCurrentStatus(
      PhaseProgress.Status.FinishedSucceeded);

    return PhaseResult.SimpleSuccess;
  }

  readonly PhaseContext mPhaseContext;
  readonly ISystemDate mSystemDate;
}
```

Recapitulando, lo necesario para implementar una nueva fase es lo siguiente:

- Decorar la clase correspondiente con el atributo `PhaseId`.
- Decorar los métodos que vayan a ser puntos de entrada para la fase con el atributo `PhaseEntryPointId`.
    - Cada uno de estos métodos deberá devolver `Task<PhaseResult>`.

    - Los parámetros de estos métodos se corresponderán con las entradas para la fase en el fichero de definición de trabajos y, por lo tanto, sus nombres no se pueden ofuscar.
    - El último parámetro ha de ser de tipo `CancellationToken`.
        - La fase debería consultar antes de comenzar cualquier trabajo pesado el estado del *token* de cancelación para terminar temprano en caso de que sea necesario.
        - Para controlar que la fase no termina con una excepción indeseada, en lugar de pasar el *token* de cancelación hacia tareas cuya implementación no se controla, se deberían crear `CancellationTokenSource` enlazados al *token* original, y controlar al lanzar tareas desde la fase las posibles `TaskCanceledException` que puedan ocurrir.
- La fase debería tener (aunque no es imprescindible) un parámetro de tipo `PhaseContext` en el constructor.
    - Este objeto representa el contexto de activación de la fase, y le permitirá, entre otras cosas, activar sub-fases y reportar el progreso de la fase a las capas superiores.
- Adicionalmente, si la fase necesita acceder a la hora del sistema, debería tener (aunque, de nuevo, no es imprescindible) un parámetro de tipo `ISystemDate` en el constructor, en lugar de acceder a `System.DateTime` para lo mismo.

5.6 El motor de activación

El motor de activación es la parte más compleja del sistema implementado. Es la que hace posible extender la funcionalidad del orquestador mediante bibliotecas de terceros, cargando, instanciando y ejecutando tipos y métodos mediante reflexión.

Las clases principales del motor de activación se definen en el espacio de nombres `Orchestrator.Server.Jobs.Phases`:

- `PhaseActivator`
- `PhaseCatalogue`

- `PhaseContext`

Sin embargo, una parte fundamental de la activación de fases es la correcta interpretación de las entradas para esta fase especificadas en el fichero de definición de trabajos. Las clases que se encargan de este trabajo se definen en el espacio de nombres `Orchestrator.Server.Jobs.Phases.Inputs`:

- `PhaseInputs`
- `ConvertPhaseInputs`
- Transformadores JSON asociados a tipos concretos: `Int16Converter`, `Int32Converter`, `DateTimeConverter`…

El motor de activación actúa en dos tiempos: la carga de bibliotecas y descubrimiento de fases, y la activación de fases.

5.6.1 Carga de bibliotecas y descubrimiento de fases

La primera etapa del motor de activación es la carga de bibliotecas y descubrimiento de fases. Esta etapa se ejecuta de manera incondicional con el arranque de la aplicación, no se puede ejecutar bajo demanda. Es decir, aunque el código se puede adaptar para soportar este escenario (ver sección *Carga y descarga de bibliotecas en caliente*), no se pueden añadir nuevas bibliotecas en caliente al orquestador. Para ello, es necesario editar el fichero de configuración y reiniciar el programa.

Como ya se vio en la sección *El fichero de configuración*, las bibliotecas que el orquestador ha de cargar se definen mediante la clave `AssembliesToLoad`, cuyo valor será una lista de `strings` con las ubicaciones en disco de las bibliotecas. La palabra reservada `'self'` hará que el orquestador se examine a sí mismo en busca de fases, lo que le hará cargar las que se ofrecen predefinidas:

```
{
  "AssembliesToLoad": [
    "self",
    "/Users/sluisp/orchestrator/assemblies/AwsPhases.dll",
    "/Users/sluisp/orchestrator/assemblies/ApplicationDeployment.dll"
  ]
}
```

Para realizar la carga de bibliotecas se usa un `AssemblyLoadContext`. En el *runtime* de .NET (Core y 5), el `AssemblyLoadContext` se encarga de localizar y cargar dependencias, por lo que todas las aplicaciones lo utilizan, aunque sea de manera implícita. Todas las aplicaciones .NET tienen, por tanto, un *contexto de carga de binarios* por defecto, accesible a través de la propiedad `AssemblyLoadContext.Default`.

Para cargar bibliotecas de manera manual, sin embargo, el `AssemblyLoadContext` tiene que utilizarse de manera explícita. Un contexto de carga puede contener al mismo tiempo una única versión de una biblioteca, lo que plantea el primer problema. Si la biblioteca que se va a cargar (que contendrá las fases desarrolladas por terceros) depende de otras bibliotecas de las que ya dependa a su vez el orquestador, si las versiones difieren (por ejemplo, el orquestador usa la biblioteca "*A*" en su versión 1, y la extensión con las fases usa también la biblioteca "*A*" pero en su versión 2) la carga de estas dependencias no va a ser posible, al menos no en el contexto de carga por defecto.

Por lo tanto, para conseguir cargar tanto la biblioteca como todas sus dependencias con éxito, es necesario utilizar un contexto de carga diferente al de por defecto. Para ello, se le asignará un nombre único durante el tiempo de vida de la aplicación, y un booleano que indicará si la descarga de ese contexto está habilitada.

En caso de que se quisiese soportar en el futuro la carga en caliente de extensiones, también se debería soportar la descarga. Más sobre esto en la sección *Carga y descarga de bibliotecas en caliente*.

Una vez creado el contexto de carga correspondiente, se puede cargar la biblioteca utilizando su localización en disco. Esto, además de cargar la biblioteca, proporcionará como resultado una referencia a un objeto de tipo `Assembly`, a través del cuál se puede ejecutar la reflexión necesaria para el descubrimiento de las fases.

```
public void LoadAssemblyAndRegisterPhases(string assemblyPath)
{
  string dllName = Path.GetFileNameWithoutExtension(assemblyPath);
  string loadContextName =
    $"loadContext.{dllName}.{Guid.NewGuid().ToString()[0..8]}";

  AssemblyLoadContext alc = new(loadContextName, false);
  Assembly assembly = alc.LoadFromAssemblyPath(assemblyPath);
```

```
  RegisterPhasesInAssembly(assembly);
}
```

En cuanto al descubrimiento de fases, como ya se ha señalado anteriormente, se realiza por reflexión.

Siendo la reflexión un mecanismo costoso en tiempo, toda la información descubierta sobre los tipos y métodos necesarios para activar una fase se guardan en memoria, en un diccionario. De esta manera, cuando llegue la parte de la activación, no es necesario pasar de nuevo por el proceso de reflexión, aumentando el tiempo de carga de una biblioteca a cambio de disminuir el tiempo requerido para comenzar a ejecutar un trabajo.

Primero se comienza con el descubrimiento de tipos dentro de la biblioteca. Para ello, el API de reflexión de .NET proporciona un método para obtener de manera sencilla todos los tipos dentro de un ensamblado (traducción de *Assembly*, que representa, en realidad, a cualquier binario .NET – tanto a bibliotecas como aplicaciones):

```
Assembly asm = /* Load the assembly */;
foreach (Type t in asm.GetTypes())
{
  // Examine the type
}
```

Una vez obtenido el tipo, se comprueba si este está decorado con el atributo `PhaseId`. En caso de no estarlo, se puede descartar rápidamente y continuar con el siguiente:

```
PhaseIdAttribute? phaseIdAttr =
  t.GetCustomAttribute<PhaseIdAttribute>();

  if (phaseIdAttr == null)
    continue;
```

Si, por el contrario, el tipo sí está decorado con el atributo `PhaseId`, entonces se comienza a examinar sus métodos:

```
foreach (MethodInfo m in t.GetMethods())
{
  // Examine the method...
}
```

Y, de la misma manera que se hizo para el tipo, ahora es necesario comprobar si el método está decorado con el atributo `PhaseEntryPointId`, para, de nuevo, descartarlo rápidamente de no ser así y poder continuar con el siguiente:

```
PhaseEntryPointIdAttribute? entryPointIdAttr
  = m.GetCustomAttribute<PhaseEntryPointIdAttribute>();

  if (entryPointIdAttr == null)
    continue;
```

Una vez se tiene un método decorado por `PhaseEntryPointId`, dentro de una clase decorada por `PhaseId`, es necesario comprobar si el método cumple con el contrato definido anteriormente: debe tener al menos un parámetro de tipo `CancellationToken`, y debe devolver `Task<PhaseResult>`.

Comprobar el tipo de retorno es sencillo: es la propiedad `ReturnType` del tipo `MethodInfo`. Para comprobar el parámetro hay que iterar sobre todos ellos (en realidad, que el `CancellationToken` sea el último parámetro es una cuestión de estilo – el orquestador solo fuerza su existencia).

```
if (m.ReturnType != typeof(Task<PhaseResult>)
  continue;

bool complies = false;
ParameterInfo[] parameters = m.GetParameters();
foreach (var parameter in parameters)
{
  if (parameter.ParameterType == typeof(CancellationToken))
  {
    complies = true;
    break;
  }
}

if (!complies)
  continue;
```

Si todas las condiciones se cumplen, se guardarán todos los datos necesarios para la activación en dos diccionarios. El primero de ellos indexa el tipo por el valor del atributo `PhaseId`, mientras que el segundo indexa los puntos de entrada de la fase por el tipo que implementa dicha fase. Los valores de este segundo diccionario son, a su vez, diccionarios que indexan la información de los métodos por el nombre de los puntos de entrada de fase que representan esos métodos:

```
readonly Dictionary<string, Type> mPhases = new();
readonly Dictionary<Type, Dictionary<string, MethodInfo>> mEntryPoints = new();
```

Estos dos diccionarios serán utilizados (de manera indirecta, a través del `PhaseCatalogue` o *catálogo de fases*) por el activador.

5.6.2 Activación de fases

La segunda etapa del motor de activación es la instanciación de las fases y la ejecución de su método de entrada. Esta etapa se ejecuta bajo demanda cuando un usuario pide la ejecución de un nuevo trabajo, y posteriormente cada vez que una fase activa una fase anidada.

La clase donde tiene lugar este proceso es `PhaseActivator`. Tiene tres métodos públicos, cuyo propósito es:

- `AddService<T>`: permite añadir nuevas dependencias al mecanismo de inyección implementado por el orquestador.
- `ActivateRootPhase`: activa la fase raíz de un trabajo.
- `ActivateNestedPhase`: activa las fases anidadas.

Como se ha comentado anteriormente, el mecanismo de inyección de dependencias proporcionado por el orquestador es muy simple. Únicamente permite registrar instancias, y solo se permite registrar una instancia por tipo. Para más información sobre el trabajo futuro en la inyección de dependencias, consultar la sección *7.1.2Inyección de dependencias de terceros*.

En cuanto a la activación de la fase raíz y de las fases anidadas, es un proceso muy similar.

Para la fase raíz es necesario crear las dependencias que para las fases anidadas ya existen: el contexto de activación (`PhaseContext`) padre y el *token* de cancelación.

El proceso de activar una fase es el siguiente:

Para activar la fase raíz, primero es necesario disponer del contexto de activación del trabajo completo (`PhaseContext.BuildForJob`). Este contexto de activación se crea a partir de dependencias manejadas directamente por el orquestador:

- El propio activador (de forma que este se va pasando hacia abajo, permitiendo a su vez a fases anidadas activar nuevas fases).
- Un *token* de trabajo.
- Las entradas declaradas en el fichero de definición de trabajos.
- El diccionario con las salidas que se esperan (para realizar los correspondientes renombrados).
- El progreso raíz.

El *token* de trabajo se verá en más profundidad en la sección *Tokens de acceso, de refresco y de trabajos*. De momento basta señalar que sirve para suplantar al usuario a la hora de realizar determinadas acciones.

En cuanto al progreso, aquí referirse a ello como progreso "raíz" no significa que sea el progreso de la fase raíz. Como se vio en la sección *Cómo reportar el progreso a las capas superiores*, la clase `PhaseProgress` se estructura en forma de grafo acíclico dirigido, y la raíz representa al progreso del trabajo completo. Esta raíz tendrá un único progreso hijo, que sí representará el progreso de la fase raíz. Luego, el progreso de cada fase anidada se irá añadiendo al progreso de esta fase raíz (que no al progreso raíz) y a los hijos de esta, dependiendo de cómo estén anidados los trabajos.

En cuanto al *token* de cancelación, como ya se ha visto en secciones anteriores, será lo que permita parar el trabajo desde el API REST en caso de que el usuario lo solicite.

Una vez creado el contexto de activación del trabajo completo, la fase raíz se activa de igual manera a las fases anidadas.

Mientras que en la sección *El fichero de definición de trabajos* se vio cómo es el fichero con el que opera el usuario, para seguir hablando sobre la activación de fases es necesario conocer las clases que modelan el contenido de dicho fichero. Son `JobDefinition` (que modela el fichero completo) y `JobPhaseDefinition` (que modela la fase raíz y cualquier fase anidada):

```
public class JobDefinition
{
  [JsonProperty("jobName")]
  public string? JobName { get; set; }
```

```
  [JsonProperty("inputs")]
  public Dictionary<string, object>? Inputs { get; set; }

  [JsonProperty("rootPhase")]
  public JobPhaseDefinition? RootPhase { get; set; }
}

public class JobPhaseDefinition
{
  [JsonProperty("name")]
  public string? Name { get; set; }

  [JsonProperty("phaseId")]
  public string? PhaseId { get; set; }

  [JsonProperty("phaseEntryPointId")]
  public string? PhaseEntryPointId { get; set; }

  [JsonProperty("inputs")]
  public Dictionary<string, JToken>? Inputs { get; set; }

  [JsonProperty("outputs")]
  public Dictionary<string, string>? Outputs { get; set; }
}
```

Todos los tipos que se usan para modelar las distintas propiedades del fichero son los esperables: los nombres (`JobDefinition.JobName` y `JobPhaseDefinition.Name`) son cadenas de texto, las entradas del trabajo (`JobDefinition.Inputs`) son un diccionario que mapea cadenas de texto a objetos… sin embargo las entradas de las fases (`JobPhaseDefinition.Inputs`) es un diccionario que mapea cadenas de texto a objetos `JToken`, que modelan un *token* JSON. Deserializar una cadena de texto JSON a un objeto de tipo `JToken` permite retrasar en el tiempo el examinarla, ya que cualquier cadena válida JSON es, al fin y al cabo, un *token* JSON. Más adelante se verá la importancia de retrasar esta deserialización.

Una vez visto este detalle se pasa a describir en detalle la activación de una fase. Es un proceso complejo que involucra multitud de pasos, por lo que se recomienda seguir la explicación relacionándola con el código del fichero *PhaseActivator.cs*. Se añaden los fragmentos de código más relevantes aquí, no obstante, como referencia.

Para activar una fase, se comienzan comprobando los datos de entrada básicos. Ni el identificador de la fase (`PhaseId`), ni el identificador del punto de entrada (`PhaseEntryPointId`) ni el nombre (`Name`) pueden ser nulos o cadenas de texto vacías.

Una vez comprobado eso, se recupera del catálogo (`PhaseCatalogue`, donde se realizó la parte de descubrimiento de fases) la información sobre el tipo que implementa la fase y sobre el método que implementa el punto de entrada.

```
if (!mCatalogue.TrySolvePhase(
  definition.PhaseId,
  definition.PhaseEntryPointId,
  out Type? phaseType,
  out MethodInfo? methodInfo,
  out string? errorMessage))
{
  throw new InvalidOperationException(errorMessage);
}
```

Con esta información ya disponible, se crea el `PhaseContext` enlazado. Anteriormente ya se creó un `PhaseContext` para el trabajo completo. Como la fase raíz y las anidadas se instancian de igual forma, el contexto de activación de la fase raíz estará enlazado al contexto de activación del trabajo completo.

```
PhaseContext ctx = PhaseContext.BuildForNestedPhase(
  mSystemDate.UtcNow(),      // DateTime
  this,                      // PhaseActivator
  definition.Name,           // string
  definition.Outputs,        // Dictionary<string, string>
  parentActivationContext);  // PhaseContext
```

Seguidamente, se obtiene el primer constructor definido para el tipo. No se soporta pues sobrecarga de constructores. A partir de la información sobre los parámetros necesarios que este proporciona, se obtienen los argumentos correctos para su instanciación:

```
object[] argumentsForInstantiation =
  GetArgumentsForInstantiation(ctx, phaseType.GetConstructors()[0]);
```

Para ello, primero se recorre cada uno de los parámetros del constructor (`ConstructorInfo.GetParameters()`). Si el tipo del parámetro coincide con alguno de las posibles dependencias (`IPhaseContext`, `ISystemDate`, o el resto de las dependencias añadidas mediante el método `AddService`), se utiliza para la instanciación. Si se encuentra algún parámetro cuyo tipo no se tiene disponible como servicio, se pasará como nulo.

```
object?[] GetArgumentsForInstantiation(
  PhaseContext context, ConstructorInfo constructor)
{
  Type activationContextType = typeof(IPhaseContext);
```

```
  ParameterInfo[] parameters = constructor.GetParameters();
  object?[] arguments = new object[parameters.Length];

  for (int i = 0; i < parameters.Length; i++)
  {
    if (mServices.TryGetValue(
      parameters[i].ParameterType,
      out object? service))
    {
      arguments[i] = service;
      continue;
    }

   if (parameters[i].ParameterType.Equals(activationContextType))
   {
     arguments[i] = context;
     continue;
   }

    arguments[i] = null;
  }

  return arguments;
}
```

Seguidamente, se crea la instancia correspondiente utilizando el API de reflexión de .NET. Este este API el que da nombre al motor de activación, pues en .NET crear una instancia es el proceso de *activar un tipo*:

```
object phase = Activator.CreateInstance(phaseType, argumentsForInstantiation);
```

Una vez creada la instancia, es necesario obtener los argumentos para invocar al método de entrada de la fase. Salvo el *token* de cancelación, todos los argumentos se obtendrán a partir de las entradas de la fase. Anteriormente se señaló que estas entradas son un diccionario cuyos valores son objetos de tipo `JToken`, es decir, cadenas JSON que todavía no han sido deserializadas.

La importancia de retrasar la deserialización de dichos *tokens* radica en que, hasta el momento mismo de la llamada al método de entrada mediante reflexión, no se sabe cuál es el tipo adecuado al que hay que deserializar cada parámetro. Y, lo más importante, puede que el tipo final ni tan siquiera coincida con el tipo que se refleja en el JSON. Esta situación se detalla en los siguientes ejemplos.

Sea esta la signatura del método de entrada de una fase:

```
 public async Task<PhaseResult> Method(string address, CancellationToken ct) { }
```

Y esta una posible definición de trabajo en el fichero:

```
(...)
{
  "name": "Phase name",
  "phaseId": "somePhaseId",
  "phaseName": "somePhaseName",
  "inputs": {
    "address": "42 Wallaby Way, Sydney"
  }
}
(...)
```

Una vez se examina la signatura de `Method`, se sabe que el parámetro `address` tiene que ser deserializado como un `string`. Coincidentemente, en el JSON también está definido como un `string`, por lo que no supone un problema.

Sin embargo, en el siguiente caso los tipos del JSON y de la signatura del método ya no coinciden. Sea esta la signatura:

```
public async Task<PhaseResult> Method(int meaningOfLife, CancellationToken ct) { }
```

Y esta la definición del trabajo en el fichero:

```
{
  (...)
  "inputs": {
    "deepThoughtAnswer": 42
  },
  "rootPhase": {
    "name": "Phase name",
    "phaseId": "somePhaseId",
    "phaseName": "somePhaseName",
    "inputs": {
      "meaningOfLife ": "${deepThoughtAnswer}"
    }
  }
  (...)
}
```

El parámetro `meaningOfLife` debería ser leído como un `int`. Sin embargo, el tipo usado en el fichero JSON no es numérico, sino que es una cadena de texto. Adicionalmente, dicha cadena de texto está utilizando una sintaxis especial que, como ya se vio en la sección *El fichero de definición de trabajos*, está referenciando a una variable – en este caso, definida en una de las entradas del trabajo.

Por lo tanto, la deserialización de los parámetros necesarios para invocar al método no se puede hacer confiando simplemente en que el tipo del parámetro sea

compatible con el tipo JSON utilizado en el fichero de definición: también hay que examinar si se trata de una variable (con la sintaxis `'${<NAME>}'`, que puede referenciar a las entradas del trabajo, o a las salidas de una fase previa) o de un secreto de usuario (con la sintaxis `'${secret:<NAME>}'`).

De dicha deserialización se encargan en conjunto las siguientes clases:

- `SolveUserSecret`: se encarga de descifrar los secretos de usuario. Para ello se vale, entre otras cosas, del *token* de trabajo, que, como ya se ha dicho, se verá en profundidad en la sección *Tokens de acceso, de refresco y de trabajos*. La funcionalidad de `SolveUserSecret` es sencilla: dado el nombre de un secreto, devuelve su valor en texto plano.
- `ConvertPhaseInputs`: se encarga de transformar las entradas del diccionario de `PhaseInputs` a los tipos adecuados.
- Las clases `*Converter` (una por cada tipo que se quiere manejar explícitamente – que son en su mayoría los tipos primitivos del lenguaje): se encargan de deserializar un *token* JSON a un tipo concreto. Para ello, se valen de la clase `ConvertPhaseInputs`. Derivan de la clase base `JsonConverter<T>`, proporcionada por el API de la biblioteca .NET *Newtonsoft.Json*.

Casi todas las clases de tipo `*Converter` se implementan de la misma forma. Es necesario tener una por cada tipo que se quiera manejar (`Int32Converter` derivará de `JsonConverter<Int32>`, etc), pero la deserialización del token JSON es idéntica para casi todos los tipos, menos para las cadenas de texto, por la particularidad de que hay que examinar su contenido para averiguar si hacen referencia a alguna variable o a algún secreto del usuario.

Así pues, la deserialización de todos los tipos (que tarde o temprano recurren a la deserialización de cadenas de texto para resolver las variables de una manera unificada) se realiza de la siguiente forma (se ofrece el ejemplo de `Int32Converter` para facilitar la lectura, pero es extrapolable a cualquier tipo):

```
public override int ReadJson(
  JsonReader reader,
  Type objectType,
```

```
  [AllowNull] int existingValue,
  bool hasExistingValue,
  JsonSerializer serializer)
{
  if (reader.TokenType == JsonToken.Integer)
    return Convert.ToInt32(reader.Value);

  if (reader.TokenType != JsonToken.String || reader.Value == null)
    return hasExistingValue ? existingValue : default;

  if (mConvertInput.TryConvertSingle(
    (string)reader.Value, int.Parse, out int intResult))
  {
    return intResult;
  }

  if (mConvertInput.TryConvertSingle(
    (string)reader.Value, long.Parse, out long longResult))
  {
    return (int)longResult;
  }

  return hasExistingValue ? existingValue : default;
}
```

Primero se comprueba si el tipo del *token* que se quiere deserializar coincide con el tipo al que se quiere deserializar. En ese caso, no es necesario hacer más: se convierte el valor al tipo adecuado y se devuelve.

Seguidamente, se comprueba si el tipo es un `string`. Si no lo es, se devuelve el valor por defecto para el tipo (que, para `Int32`, será 0). La alternativa era lanzar una excepción: se prefirió ejecutar el método de entrada de la fase con un valor por defecto para el parámetro a fallar en la activación por los parámetros.

Si el tipo del *token* es un `string`, se utiliza `ConvertPhaseInputs` para obtener un valor. `ConvertPhaseInputs` es una clase sencilla que solo trabaja en términos de cadenas de texto. Por ello, se pasa una función (`int.Parse`) para convertir el resultado (en .NET las funciones son *ciudadanos de primer nivel*). Si la conversión tiene éxito (los métodos cuyo nombre comienza por *Try* retornan, según el argot de C#, un booleano que indica el éxito de la operación) significa que se ha podido convertir o bien una variable o bien un secreto de usuario a un `Int32`, por lo que se devuelve el resultado.

Si no ha tenido éxito, se devuelve nuevamente el valor por defecto.

En cuanto a `ConvertPhaseInputs`, su funcionamiento es relativamente sencillo. Al igual que los progresos, las entradas, representadas por `IPhaseInput`, también guardan una relación de parentesco. Una fase puede, a través de su contexto de activación, acceder a su instancia de `PhaseInput`, y a las instancias de `PhaseInput` de las fases en las que se anida, hasta la raíz.

Este es el mecanismo que permite a una fase devolver un resultado, y a una fase que se ejecute a continuación acceder a él. Póngase por ejemplo la fase A, que activa como fases anidadas primero a la fase B y luego a la fase C, de manera secuencial. La fase B puede acceder a las entradas de sí misma, y a las entradas de la fase A a través de la relación de parentesco. Cuando la fase B termina, exporta un resultado con un nombre concreto, que la fase que la activó (A) incorpora a sus propias entradas. Cuando se ejecute la fase C, podrá acceder al resultado de la fase B a través de las entradas de la fase A. La exportación de salidas a las entradas de la fase madre se realiza cuando termina la ejecución de la fase, y es un mecanismo que enlaza el orquestador mediante una continuación de tarea (ligado con el API de `Task<T>`) como última parte de la activación, por lo que se verá más adelante.

La clase `ConvertPhaseInput` hace lo que se puede esperar de ella: extraer de las cadenas de texto los marcadores adecuados para reconocer el nombre de variables o de secretos de usuario, y luego hacer los reemplazos correspondientes con los valores encontrados. La búsqueda de los valores a través de las entradas de la fase, recorriendo el parentesco, se puede observar en el método `ConvertPhaseIntpus.GetFromInputs`:

```
object? GetFromInputs(string key)
{
  IPhaseInputs? currentInputs = mPhaseInputs;
  while (currentInputs != null)
  {
    if (currentInputs.TryGet(key, out object? result))
      return result;

    currentInputs = currentInputs.Parent;
  }

  throw new InvalidOperationException(
    $"None of the inputs contains key '{key}'!");
}
```

Al igual que en un lenguaje de programación tradicional, en caso de colisión de nombres, se prima la variable del contexto más local. Así pues, la búsqueda de variables comienza por el contexto más local, y continúa la búsqueda por las entradas de la fase madre únicamente en caso de no haber encontrado la variable adecuada. En última instancia, si no se encuentra la variable, se lanza una excepción.

Una vez se conoce el funcionamiento de deserialización de las variables de entrada del método a ejecutar y la importancia de deserializar el fichero de definición de trabajo de manera perezosa, se puede continuar detallando el proceso de activación.

Para obtener los argumentos de invocación al método de entrada son necesarios los siguientes elementos:

- Un JsonSerializer (que se encargará de la deserialización del diccionario de entradas de la manera que se acaba de ver).
- El diccionario de entradas de la fase (cuyos valores serán deserializados por el JsonSerializer creado).
- El token de cancelación (que será pasado como parámetro en la posición -todavía desconocida- que corresponda).
- La lista de parámetros del método, obtenida del MethodInfo correspondiente.

```
object?[] argumentsForInvokation = GetArgumentsForInvokation(
  BuildJsonSerializer(ctx, mUserAuthentication, mUserSecrets), // JsonSerializer
  methodInfo.GetParameters(),                                  // ParameterInfo[]
  definition.Inputs,                                           // Dictionary
  ct);                                                         // CancellationToken
```

Recorriendo el vector de `ParameterInfo` se crea un vector paralelo con los argumentos que se utilizarán para la invocación:

```
static object?[] GetArgumentsForInvokation(
  JsonSerializer serializer,
  ParameterInfo[] parameters,
  Dictionary<string, JToken>? inputs,
  CancellationToken token)
{
  object?[] arguments = new object[parameters.Length];
  for (int i = 0; i < parameters.Length; i++)
  {
    Type paramType = parameters[i].ParameterType;
```

```
    if (paramType == typeof(CancellationToken))
    {
      arguments[i] = token;
      continue;
    }

    if (inputs == null)
        continue;

    if (!inputs.TryGetValue(parameters[i].Name, out JToken? arg))
    {
      arguments[i] = paramType.IsValueType
        ? Activator.CreateInstance(paramType)
        : null;
      continue;
    }
    arguments[i] = arg.ToObject(paramType, serializer);
  }
  return arguments;
}
```

Es este método el que impone la restricción que impide ofuscar el nombre de los parámetros en el método de entrada de una fase. En el diccionario de entradas (`inputs`) se buscan los valores utilizando como clave el nombre del parámetro (`parameters[i].Name`). En caso de ofuscación, el nombre del parámetro en el código no coincidiría con el de la clave en el diccionario definido en el fichero del trabajo. Se propone una solución a esto en la sección *Permitir la ofuscación de las fases*.

Con los argumentos ya calculados, se invoca el método, se comprueba que el valor devuelto es del tipo que corresponde, y se actualiza el estado del progreso asociado:

```
object? t = methodInfo.Invoke(phase, argumentsForInvokation);
if (t is not Task<PhaseResult> task)
{
  throw new InvalidOperationException(
    $"The phase entry point {definition.PhaseId}.{definition.PhaseEntryPointId}"
    + $" returned something unexpected: '{t?.GetType().ToString() ?? "NULL"}'");
}

ctx.Progress.UpdateCurrentStatus(PhaseProgress.Status.Running);
```

Por último, se añade a la tarea la continuación que registra las salidas de la fase, y se devuelve la nueva tarea (al añadir una continuación a una `Task`, el resultado es una nueva `Task` que engloba tanto a la tarea original como a la tarea de la continuación):

```
return task.ContinueWith(
  RegisterOutputs,
  ctx,
  TaskContinuationOptions.OnlyOnRanToCompletion);
```

Los argumentos del método `ContinueWith` son, por orden:

- El método que representa la continuación, y que se ejecutará cuando termine la tarea. En este caso, es un método llamado `RegisterOutputs`.
- El argumento que admite el método que representa la continuación. A este argumento (de tipo `object` en la signatura de `ContinueWith`) se le llama el estado, y representa los datos que serán utilizados en la tarea de continuación. En este caso se le pasa el contexto de activación, representado por la variable `ctx`.
- Las condiciones en las que la continuación se debe ejecutar. En el orquestador se eligió por diseño que la continuación se ejecutase únicamente si la tarea se completaba. Es por ello por lo que las fases deben terminar, aun cuando son canceladas, saliendo ordenadamente y no lanzando una excepción. Si una fase lanza una excepción durante su ejecución, tal y como se vio en la sección *El API asíncrono y el token de cancelación*, su estado no será `RanToCompletion`, sino `Cancelled`, y la continuación que registra las salidas no se ejecutará.

Así pues, el método que se ejecuta cuando termina una fase, y antes de que se retorne el control al código que activó dicha fase es, indefectiblemente, `PhaseActivator.RegisterOutputs`, que recibirá como parámetros la tarea de la que el método es continuación y el contexto de activación de dicha fase, y como valor de retorno, `PhaseResult` (pues este método, introducido al final de la cadena de ejecución de la fase, ha de ser transparente para el código cliente y, por lo tanto, ha de devolver el resultado de la tarea original sin modificarlo).

```
static PhaseResult RegisterOutputs(Task<PhaseResult> completedTask, object? def)
```

La exportación de las salidas de una fase a las entradas de la fase madre se hace de la siguiente forma.

```
if (def is not PhaseContext context)
  return completedTask.Result;
```

```
context.Progress.UpdateCurrentStatus(
  GetProgressStatusFromPhaseResult(completedTask.Result));
```

Primero se hace una comprobación de seguridad sobre el estado pasado a la continuación. Si no es del tipo `PhaseContext`, se devuelve el resultado original. Una vez se comprueba que efectivamente es del tipo esperado, se finaliza el progreso asociado, de forma que, aunque el desarrollador de la fase no lo haya hecho, el progreso reportado es correcto, puesto que se siempre se marca como `Running` en la ejecución del método, y como `Finished` en la ejecución de la continuación.

```
if (context.Parent == null || context.Outputs == null)
  return completedTask.Result;
```

Seguidamente se comprueba si el contexto de activación tiene un contexto padre, y si la fase tiene salidas marcadas en el fichero de definición de trabajos (propiedad `Outputs` de la definición de la fase). Si no se da una de las dos condiciones, no es necesario hacer nada más.

```
if (completedTask.Result.Outputs == null
  || completedTask.Result.Outputs.Count == 0)
{
  return completedTask.Result;
}
```

A continuación, se comprueban las salidas registradas en el `PhaseResult` resultante de la ejecución de la fase. Así pues, aunque una fase haya dejado salidas en `PhaseResult`, estas serán exportadas únicamente si se marcan en el fichero de definición de trabajo.

```
foreach ((string outputsKey, string inputsKey) in context.Outputs)
{
  if (!completedTask.Result.Outputs.TryGetValue(outputsKey, out object? value))
    continue;

  context.Parent.Inputs.Add(inputsKey, value);
}

return completedTask.Result;
```

Por último, se recogen las salidas del diccionario `PhaseResult.Outputs` utilizando como clave las claves del diccionario de `PhaseDefinition.Outputs` (`outputsKey`), y se exportan al diccionario de entradas de la fase madre (`context.Parent.Inputs`) utilizando como clave el valor de la entrada correspondiente de `PhaseDefinition.Outputs` (`inputsKey`).

A partir de ese punto, las fases que se ejecuten a continuación tendrán disponible en sus entradas la salida de la fase que ha terminado, siempre y cuando estas fases se encuentren, en el árbol de activación, al mismo nivel que la fase que ha terminado, o a uno inferior por debajo de la misma rama.

5.7 Gestión de usuarios

El usuario es quien opera el orquestador. Todos los trabajos que ejecuta el orquestador lo hacen en el contexto de un determinado usuario. El orquestador da soporte a las operaciones de creación, autenticación y autorización de usuarios, así como a la creación, modificación y eliminación de secretos de usuarios.

Se detalla a continuación cómo se implementa la gestión de estos aspectos.

5.7.1 Gestión segura de los ficheros

El orquestador utiliza ficheros de texto para proporcionar soporte a las operaciones relacionadas con los usuarios. Estos ficheros están almacenados en una ubicación configurable del sistema de ficheros de la máquina en la que se ejecuta el orquestador.

Para un correcto funcionamiento, el orquestador debe por tanto garantizar:

- Que estos ficheros no son accedidos de manera concurrente para la lectura y la escritura (pues una lectura a mitad de una escritura podría derivar en datos parciales o corruptos), y que no sucede más de una escritura de manera simultánea (pues dos escrituras concurrentes derivarían en pérdida de datos).
- Que estos ficheros no se corrompen durante la escritura, pues eso supondría bloquear a un usuario fuera del sistema.

5.7.1.1 Lecturas y escrituras concurrentes

Los datos de cada usuario se almacenan en dos ficheros diferentes, cuya estructura se verá más adelante: *user.json* y *secrets.json*. Estos ficheros existen por cada usuario del sistema, y deben ser accedidos únicamente por una instancia del orquestador de manera simultánea. Es decir, no se deberían ejecutar dos instancias del

orquestador configuradas para almacenar sus datos en la misma localización del sistema de ficheros.

Esta restricción viene dada por la forma en la que el orquestador garantiza la sincronización en el acceso de lectura/escritura a los ficheros, y se podría levantar en el futuro si el mecanismo cambia. Se discute sobre esta posibilidad en la sección *Permitir más de una instancia del orquestador sobre los mismos datos*.

El mecanismo implementado para la sincronización es un bloqueo de lectura/escritura, proporcionado por la clase del API de .NET `ReaderWriterLockSlim`. Es un mecanismo estándar de protección del acceso a un recurso, que permite múltiples accesos simultáneos para lectura, pero solo un acceso simultáneo para escritura.

Una primera aproximación fue utilizar un único bloqueo de lectura/escritura para el almacenamiento completo. Sin embargo, en un hipotético escenario de carga este planteamiento tenía un problema importante de cuello de botella, pues cada vez que se necesitase escribir la información de un usuario, se tendría que bloquear la lectura de todos los demás, aunque no estuviesen afectados por la operación.

Así pues, lo que se terminó implementando es un bloqueo por recurso, a través de las interfaces `IResourceReaderWriterLock` y `ILock`.

La interfaz `ILock` define cómo adquirir y liberar los distintos tipos de bloqueos sobre el recurso mediante los mismos métodos que `ReaderWriterLockSlim` (que son, por otro lado, los métodos que cabría esperar de este tipo de mecanismo).

```
public interface ILock
{
  void EnterReadLock();
  void EnterUpgradeableReadLock();
  void EnterWriteLock();
  void ExitReadLock();
  void ExitUpgradeableReadLock();
  void ExitWriteLock();
}
```

El nombre de todos los bloqueos son auto-explicativos, salvo el *UpgradeableReadLock*. Este consiste en obtener un bloqueo de lectura que, opcionalmente, puede actualizarse a uno de escritura sin tener por ello que abandonar el bloqueo original de lectura.

La interfaz `IResourceReaderWriterLock` define cómo obtener la instancia de `ILock` que permite adquirir los bloqueos sobre el recurso.

```
public interface IResourceReaderWriterLock
{
  ILock GetLockForResource(string userName, string resourceName);
}
```

Tanto `IResouceReaderWriterLock` como `ILock` son interfaces para permitir intercambiar la forma en la que estos bloqueos se manejan de manera transparente para las clases clientes. Se discute más al respecto en la sección *Permitir más de una instancia del orquestador sobre los mismos datos*.

A la hora de utilizar estos bloqueos, es necesario obtener primero el `ILock` correspondiente al recurso, y luego obtener el bloqueo necesario sobre el recurso y liberarlo donde sea adecuado:

```
ILock resourceLock = mResourceLock.GetLockForResource(userName, RESOURCE_NAME);
resourceLock.EnterReadLock();
try
{
  // Operate with the resource
}
finally
{
  resourceLock.ExitReadLock();
}
```

5.7.1.2 Corrupción de datos

Evitar las escrituras simultáneas sobre un recurso no evita la posible corrupción de datos. Si el proceso del orquestador sale a mitad de una escritura por un error (una excepción no manejada, un fallo a nivel de Sistema Operativo o de hardware…) el contenido del fichero puede quedar ilegible.

Para proteger al orquestador de esta posibilidad se ha implementado la escritura de ficheros de una forma similar a la que utilizan conocidos editores de texto como VIM.

Las escrituras se realizan en un fichero nuevo, temporal, con extensión `swp`. Si la escritura falla, el fichero `swp` se queda detrás y el original no se ve afectado, pudiendo ser accedido nuevamente para la lectura. Por el contrario, si la escritura ha tenido éxito, el nuevo fichero reemplazará al viejo mediante dos renombrados. El fichero viejo será renombrado añadiendo la extensión `old`, y al fichero temporal se le renombrará

quitándole la extensión `swp`, de forma que ocupará el lugar del original. La seguridad de este mecanismo se basa en que las operaciones de renombrado en el sistema de ficheros son atómicas, al contrario que la escritura. Es decir, se realiza entera o no se realiza, no siendo posible que la operación no se complete y el fichero se quede en un estado intermedio.

Esta es la secuencia de pasos realizada para escribir el fichero `file.txt`:

1. Leer el contenido completo de `file.txt`.
2. Realizar en memoria las modificaciones que sean necesarias.
3. Escribir el nuevo contenido en `file.txt.swp`.
4. Mover `file.txt` a `file.txt.old`.
5. Mover `file.txt.swp` a `file.txt`.

Este mecanismo queda implementado en la clase `FileTransaction`, que tiene los métodos `Commit` y `Rollback` y la propiedad `SwpFilePath`, que proporciona la ruta del fichero que el código cliente tiene que utilizar para escribir.

El método `Commit` se usa cuando la operación de escritura se ha realizado con éxito, para reemplazar el fichero regular con el temporal.

1. Si el fichero `file.txt.old` existe, este pertenece a una transacción anterior. Se borra.
2. Si el fichero `file.txt` existe (no existirá si es la primera vez que se escribe), se renombra a `file.txt.old`.
3. El fichero `file.txt.swp` se renombra a `file.txt`.

Se llama al método `Rollback` cuando la operación de escritura ha fallado. Esta trata de dejar los ficheros en una situación adecuada para que el orquestador pueda volver a acceder al original sin intervención manual de un administrador.

1. Si el fichero temporal `file.txt.swp` existe, se elimina.
2. Si el fichero original `file.txt` existe, significa que no se llegó a mover para el reemplazo. La operación de deshacer termina.

3. Si el fichero original `file.txt` no existe, y `file.txt.old` sí existe, significa que la confirmación falló antes de mover el temporal a la localización definitiva. Se restaura renombrando `file.txt.old` a `file.txt`.

Este es un ejemplo de uso de la clase `FileTransaction`, extraído de la escritura de secretos de usuario:

```
public void WriteSecrets(string path, Dictionary<string, string> secrets)
{
  FileTransaction fileTransaction = new(path);
  try
  {
    using (FileStream fs = File.Create(fileTransaction.SwpFilePath))
    using (StreamWriter sw = new(fs))
    using (JsonTextWriter writer = new(sw))
    {
      mJsonSerializer.Serialize(writer, secrets);
    }
    fileTransaction.Commit();
  }
  catch (Exception ex)
  {
    // Log the exception...
    fileTransaction.Rollback();
    throw;
  }
}
```

Hay que tener en cuenta que tanto el `Commit` como el `Rollback` tratan o bien de renombrar, o bien de eliminar el fichero de intercambio. Por lo tanto, es necesario llamar a estos métodos fuera de los bloques *`using`* – que es cuando los recursos asociados a `FileStream`, `StreamWriter` y `JsonTextWriter` (típicamente memoria reservada, pero de manera prominente el fichero afectado) ya han sido cerrados y liberados.

5.7.2 Fichero de datos de usuario

El fichero *users.json* es donde se almacena la información que representa a un usuario del sistema. Está modelado por la clase `User`, y tiene las siguientes propiedades:

- *Name*: es el nombre del usuario en el sistema.
- *SaltedPassword*: es la contraseña del usuario, almacenada de forma segura. Más sobre el almacenamiento seguro de contraseñas a continuación.

- *PrivateKey*: es la clave privada del usuario. Sirve para cifrar los secretos de usuario. Forma parte de un par de claves pública/privada. La clave pública se envía al usuario cuando este se registra, y no se almacena en el sistema ni vuelve a estar disponible, por lo que es necesario que el usuario la guarde en un lugar seguro.
- *Tokens*: son los identificadores de los *tokens* de acceso al sistema, que se utilizan para identificarse frente al API REST, y que consisten en una cadena de texto que contiene toda la información necesaria para identificar al usuario. Más sobre los *tokens* a continuación.
    - *AccessTokenId*: el identificador del *token* de acceso.
    - *RefreshTokenId*: el identificador del *token* de refresco.

Los ficheros de datos de usuario se almacenan debajo del directorio configurado para ello, mediante *sharding*[38] para evitar que las operaciones del sistema de ficheros se ralenticen en caso de un elevado número de usuarios en el sistema[39]. El particionado se realiza en base al resumen SHA1 del nombre de usuario. Dado un nombre de usuario ***name***, el directorio donde se almacenarán sus datos de usuario será:

```
basepath / SHA1(name)[0..2] / SHA1(name) /
```

Este es un ejemplo de fichero *user.json* (con las propiedades *saltedPassword* y *privateKey* truncadas por su longitud):

```
{
```

[38] El *sharding* o *fragmentación* es una técnica de particionado y distribución de datos horizontal, basada en algún criterio que permita la distribución de datos de manera homogénea, y que permita identificar de manera unívoca la partición donde se encuentra un dato determinado.

[39] Las operaciones de creación, listado, etc. dentro de un directorio pueden ralentizarse si dicho directorio tiene un elevado número de entradas. A pesar de que no se espera que una instalación del orquestador se use por un número lo suficientemente elevado de usuarios para encontrarse con este inconveniente, el particionado es sencillo de implementar y protege este escenario de cara al futuro.

```
  "name": "sergio",
  "saltedPassword": "QzmBo(...)U0kUER",
  "privateKey": "MIIEpA(...)P/dLw==",
  "tokens": {
    "accessTokenId": "f3d62fa5-b0ba-4087-a7d8-0a8ea55ebcd2",
    "refreshTokenId": "139df0e9-bf8a-4976-9662-55566c931f2c"
  }
}
```


5.7.2.1 Almacenamiento seguro de la contraseña

Almacenar una contraseña es siempre delicado. Cualquier acceso no autorizado al sistema puede exponer datos sensibles que permitan a un atacante malicioso ganar control de cuentas de usuario.

A la hora de guardar contraseñas, independientemente del soporte (ya sea en texto plano, en una base de datos, o de cualquier otra manera), hay que garantizar que la contraseña original no se pueda obtener a partir de la información guardada, permitiendo al mismo tiempo comprobar que la contraseña en texto plano proporcionada por el usuario coincida con aquella que el sistema tenga guardada de forma segura.

Para ello, una posibilidad es almacenar el resumen de la contraseña. Una función resumen, *hash* o función de extracto es una función que convierte elementos de un conjunto de entrada a otro de salida, y que para ser considerada criptográficamente segura ha de tener las siguientes propiedades:

- Es determinista. La misma entrada provocará siempre la misma salida.
- Es rápida de computar.
- No es posible revertir, con una capacidad de cómputo y tiempo razonable, el valor original a partir del cual se calculó un resumen dado.
- No tiene colisiones. Es decir, no existen dos valores distintos cuyo resumen sea idéntico[40].

[40] En la práctica se han encontrado ataques de colisión para algoritmos resumen como SHA1, siendo poco a poco estos algoritmos relegados en favor de otros considerados más seguros como SHA256.

- Un pequeño cambio en el valor original provoca grandes cambios en el valor resumen.

Sin embargo, guardar únicamente el resumen no es seguro. Un atacante puede generar tablas de resúmenes pre-calculados para las contraseñas más habituales[41]. Una vez obtenido el resumen de una contraseña del sistema comprometido, si al compararlo con los resúmenes pre-calculados de estas tablas se encuentra una coincidencia, la contraseña original queda expuesta.

```
$ echo "1234" | sha256sum
a883dafc480d466ee04e0d6da986bd78eb1fdd2178d04693723da3a8f95d42f4
```

Una alternativa es realizar iteraciones sobre este resumen. Si para una contraseña no se almacena el resumen, sino el resumen, del resumen, del resumen (…) del resumen, se reduce la posibilidad de que la contraseña aparezca en una tabla pre-calculada. Sin embargo, si el atacante descubre el número de iteraciones que se han hecho sobre el resumen, calcular la tabla de nuevo para las contraseñas más comunes es una labor trivial.

```
$ echo "1234" | sha256sum | sha256sum
1555dfe1c98ccc6a2bd70bc6af8448479ecaf15254508eead32801e1765d2066
```

La solución segura es el resumen con salto. El salto es un prefijo aleatorio que se añade a la contraseña. Seguidamente, se calcula el resumen del prefijo mas la contraseña, con un número de iteraciones elevado.

$$\text{Value}=\text{"Salt"}+\text{Hash}(\text{Hash}(\text{Hash}(\text{"Salt"}+\text{"Password"})))$$

Lo que finalmente se guarda es el prefijo (pues es necesario para la comprobación de la contraseña) mas el resumen.

```
$ export SALT="x1A"
$ export PASSWORD="1234"
$ echo "$SALT$(echo "$SALT$PASSWORD" | sha256sum | sha256sum)"
x1A83ad07d55e708a6715a003e8647f4a7354eed0ddd61c3ac4b99711ad6eee4f6e

$ export SALT="3ep"
$ echo "$SALT$(echo "$SALT$PASSWORD" | sha256sum | sha256sum)"
3ep46f1edb064b017fd93d8e135a874f7f9cb00b4ff1b5c452ad9807103c61df091
```

[41] Conocidas comúnmente como tablas arcoíris o "*rainbow tables*".

De esta manera, tanto si varían el prefijo como el número de iteraciones, varía el resumen que se guarda. Comprobar la contraseña es trivial si se conoce cuál es la longitud del prefijo. Se separa el prefijo del resumen, se añade el prefijo a la contraseña en texto plano pasada por el usuario, y se calcula de nuevo el resumen con el mismo número de iteraciones. Si los resúmenes coinciden, la contraseña es la misma.

Sin embargo, ahora no es computacionalmente posible atacar todas las contraseñas del sistema con tablas pre-calculadas. Cada contraseña tiene su propio prefijo aleatorio, y la misma contraseña con dos prefijos distintos no generará el mismo resumen, por lo que, aunque el atacante conozca la longitud del salto y el número de iteraciones en el cálculo del resumen, sería necesario generar una tabla de resúmenes pre-calculados por cada prefijo distinto (es decir, por cada contraseña que se quiera intentar vulnerar). Una tarea muy costosa tanto en tiempo como en recursos.

En el orquestador los valores de la longitud de prefijo, la longitud del resumen de la contraseña y el número de iteraciones en el cálculo del resumen son configurables (ver sección *El fichero de configuración*). Sus valores por defecto son 16 caracteres para el prefijo, 20 para el resumen de la contraseña, y 1000 iteraciones en el cálculo del resumen. El resumen se calcula utilizando el algoritmo PKCS #5 v2.0, descrito en el RFC 2898[42] e implementado de manera criptográficamente segura en el API de .NET por la clase `Rfc2898DerivedBytes`[43].

5.7.2.2 Tokens de acceso, de refresco y de trabajos

El orquestador se opera mediante un API REST basado en HTTP. La autenticación del usuario se realiza mediante la cabecera HTTP *Authorization*. Si se quisiese utilizar autenticación basada en usuario y contraseña, la forma estándar de hacerlo sería utilizando el esquema de autorización Basic[44], que consiste en codificar en base 64,

---

[42] https://datatracker.ietf.org/doc/html/rfc2898

[43] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rfc2898derivebytes?view=net-5.0

[44] https://datatracker.ietf.org/doc/html/rfc7617

separados por dos puntos (':') el usuario y la contraseña concatenados. Este es un método reversible – el único motivo para codificar el dato en base 64 es asegurar que los caracteres de la cabecera estén en una codificación bien conocida y soportada por todos los servidores Web.

```
$ echo "sergio:1234" | base64
c2VyZ2lvOjEyMzQK

$ echo c2VyZ2lvOjEyMzQK | base64 --decode
sergio:1234
```

El orquestador por defecto no utiliza HTTPS (HTTP sobre SSL), aunque puede configurarse para ello. No obstante, intercambiar el usuario y la contraseña continuamente por la red, aunque fuese sobre SSL, abre una superficie de exposición susceptible a ataques (de máquina en el medio[45], por ejemplo) destinados a robar las credenciales del usuario.

Si dichas credenciales fuesen sustraídas, el usuario necesitaría cambiarlas. Para reducir esta superficie de exposición, para la autenticación se utilizan *tokens* JWT (*JSON Web Token*), una forma de autenticación rotatoria y opaca, que tiene un periodo de vida limitado (y que puede invalidarse en cualquier momento sin por ello afectar a la contraseña original del usuario) y que no expone a terceras partes información tal como el nombre de usuario o la contraseña.

Los *JSON Web Tokens*, definidos en el RFC 7519[46] (y actualizado en 7797[47] y 8725[48]) son una manera compacta de intercambiar un conjunto de *pretensiones* o *claims* entre dos partes, en forma de objetos JSON.

Una *pretensión* es una pieza de información, que ambas partes acuerdan que es verdadera, sobre un determinado sujeto (normalmente el usuario o el propio *token*), y

---

[45] Una máquina se coloca en el centro de una comunicación legítima, capturando y examinando el tráfico que por ella pasa de manera silenciosa.

[46] https://datatracker.ietf.org/doc/html/rfc7519

[47] https://datatracker.ietf.org/doc/html/rfc7797

[48] https://datatracker.ietf.org/doc/html/rfc8725

que consiste en un par clave-valor. Las pretensiones pueden ser públicas, definidas por el RFC (por ejemplo, el nombre de quien posee el *token*, su rol en el sistema, la fecha en la que se extendió el *token* o la fecha tras la cuál dicho *token* ya no es aceptable), o pretensiones privadas implementadas por el desarrollador. Cada JWT, además, va criptográficamente firmado, de forma que es necesario una clave para leerlo.

Las pretensiones almacenadas en un JWT extendido por el orquestador a un usuario se modelan en la estructura `Token`, definida dentro de la clase `UserAuthentication`, y son las siguientes:

- *Token ID*: es un GUID que identifica el token de manera única. Es la única pieza de información que el sistema guarda sobre el *token*. Se almacena bajo la pretensión `Jti`[49].
- *UserName*: el nombre del usuario al que se le expidió el JWT. Se almacena bajo la pretensión `Sub`[50].
- *IssueDateUtc*: la fecha, en UTC, en la que se expidió dicho *token*. El *token* no será aceptado en una fecha anterior a la de expedición. Se almacena bajo las pretensiones `Iat`[51] y `Nbf`[52].
- *ExpireDateUtc*: la fecha, en UTC, en la que el *token* expedido deja de tener validez. En *token* no será aceptado en una fecha posterior a la de expiración. Se almacena bajo la pretensión `Exp`[53].

---

[49] Del RFC 7519, la pretensión `Jti` (*JWT ID*) proporciona un identificador único al JWT.

[50] Del RFC 7519, la pretensión `Sub` (*Subject*) identifica al sujeto del JWT.

[51] Del RFC 7519, la pretensión Iat (Issued At) expresa la fecha de expedición del token.

[52] Del RFC 7519, la pretensión Nbf (Not Before) expresa la fecha mínima a partir de la cual el token se puede considerar válido.

[53] Del RFC 7519, la pretensión Exp (Expire Time) expresa la fecha a partir de la cual el token no debe ser procesado.

Por defecto, para un *token* de acceso, la fecha de expiración es siete días posterior a la de expedición. Para un *token* de refresco, esta diferencia es mayor, de treinta días. Ambos rangos de fecha son configurables.

Adicionalmente y por completitud, se guardan en el token las dos siguientes pretensiones, no modeladas por la clase `Token`:

- Pretensión *Issuer* (`Iss`): identifica la autoridad que expidió el JWT. Según el RFC 7519 es opcional y la forma de procesarlo es dependiente de la aplicación. En el orquestador, su valor es *Orchestrator.Server*.
- Pretensión *Audience* (`aud`): identifica el público al que va destinado el JWT. Según el RFC 7519 es opcional y la forma de procesarlo es dependiente de la aplicación. En el orquestador, su valor también es *Orchestrator.Server*, pues el JWT es tanto expedido como consumido por la misma aplicación.

Para obtener un par de *tokens* (uno de acceso y otro de refresco), el usuario tiene que autenticarse frente al sistema con su nombre de usuario y su contraseña. Si la autenticación tiene éxito, el sistema expedirá los *tokens* con las pretensiones señaladas anteriormente, y guardará en el fichero de datos de usuario (*users.json*) los identificadores de los *tokens* expedidos.

La autenticación en ocasiones posteriores se realizará mediante el *token* de acceso. Cuando el orquestador reciba la cadena de texto que representa el *token* (mediante la cabecera HTTP *Authorization*, siguiendo el esquema de autenticación *Bearer Token*[54]), comenzará a procesar su validez. Las comprobaciones se realizarán en las siguientes clases y etapas:

[54] https://datatracker.ietf.org/doc/html/rfc6750

1. En un primer momento, el valor de la cabecera *Authorization* será capturado por una *etapa intermedia*[55] de ASP.NET. Dicha etapa está implementada en la clase `BearerTokenAuthenticationHandler`.
    a. La etapa intermedia examina la ruta a la que iba dirigida la petición. Si el código que se encarga de manejar las peticiones a dicha ruta (típicamente un método en una clase que implementa el patrón controlador) está decorado con un atributo que permite el acceso anónimo (`AllowAnonymous`), la etapa termina y la petición sigue su curso normal.
2. Con el *Bearer Token* (el JWT) extraído de la petición, la etapa intermedia hace uso de la clase `UserAuthentication` para verificar el contenido del JWT. El resultado de la verificación se devolverá en un objeto de tipo `AuthenticateTokenResult`, que contiene un código de resultado y, si el código indica éxito, el objeto `User` que modela al usuario (y que será el contenido deserializado de *user.json*).
3. Si la verificación del JWT ha fallado, la etapa intermedia corta el procesamiento de la petición y escribe en el cuerpo de la respuesta HTTP un objeto JSON (con una estructura documentada) el motivo del fallo. Adicionalmente, cambia el código de respuesta a HTTP 401 (*Unauthorized*).
4. Si por el contrario la verificación del JWT ha tenido éxito, la etapa intermedia añade a la petición los datos del usuario para que estén disponibles en etapas posteriores, y no tenga que repetirse el proceso de

[55] En ASP.NET el procesado de una petición HTTP se hace por etapas. Un *middleware* (a falta de una traducción mejor en castellano, "*etapa intermedia*") es una parte del código que se introduce en la cadena de procesado HTTP para encargarse de una tarea concreta, y que tiene la capacidad de modificar cómo se procesa la petición: bien evitando que pase a la etapa posterior, bien añadiendo o quitando datos, bien derivando la petición a una etapa distinta…

autenticación. Seguidamente, la etapa intermedia devuelve el control para que la petición pueda seguirse procesando de manera normal.

En cuanto a cómo la clase `UserAuthentication` verifica el contenido del JWT, se hace de la siguiente manera:

1. La primera acción que realizar es descifrar el contenido del JWT. Hay que recordar que la cadena de texto intercambiada con el usuario es un objeto JSON que está cifrado con una clave.
    a. Esta es la primera prueba de la autenticidad del *token*. Si este no hubiese sido expedido por el orquestador, no sería posible descifrarlo con la clave privada. El que sea posible descifrarlo con una clave que no es compartida demuestra al orquestador que el *token* fue expedido por este, por lo que se puede confiar en que las comprobaciones posteriores son ciertas.
2. A continuación, se extraen del *token* las fechas de expedición y de caducidad. Si la fecha actual del sistema no se encuentra entre ambas, el *token* es considerado inválido (bien por encontrarse en un momento anterior a la fecha de expedición, o en un momento posterior a la de expiración).
    a. Sobre esta comprobación cabe señalar una particularidad: el RFC 7519 fuerza de manera indirecta a que las fechas tengan una precisión máxima de segundos. Para almacenar las pretensiones `Iat`, `Nbf` y `Exp` se utiliza un valor JSON de tipo *Numeric*, que es un entero de 64 bits. El RFC especifica que en este entero de 64 bits ha de guardarse la época UNIX (tiempo pasado desde el 1 de enero de 1970), que, debido a su representación, se podrá almacenar con una precisión máxima de segundos. Si la fecha de expedición se comprueba inmediatamente después de que el token sea expedido, la precisión en la comprobación puede causar que el token sea considerado inválido. Por lo tanto, para compensar esta diferencia en las precisiones de las fechas en el token respecto a las

fechas del sistema, al comparar las pretensiones de expedición y de expiración se otorga un margen de un segundo de diferencia. Si la diferencia entre fechas es menor que ese margen, se consideran iguales y la comparación tiene éxito. Esta es una posibilidad ya advertida en el RFC 7519[56].

3. Si el *token* está dentro de su periodo de validez, la última comprobación consiste en comparar el identificador con los datos guardados en el sistema. Para ello, se lee del *user.json* del usuario especificado los identificadores de *token*, y se comparan con el especificado en las pretensiones del propio *token* pasado por el usuario. Si la comparación tiene éxito, el *token* es válido y se permite el acceso del usuario al sistema. Si no, el acceso queda rechazado.

El fichero *user.json* solo permite, de manera simultánea, un identificador para el *token* de acceso y otro para el *token* de refresco. Esto impide al usuario acceder al orquestador desde dos máquinas distintas. En el momento en el que se autentique de nuevo con su nombre de usuario y contraseña en otra máquina, todos los *tokens* ya expedidos quedan invalidados al generarse un nuevo par. Esta es una limitación de la que se habla en la sección *Permitir al usuario más de un par de tokens*.

En cuanto al token de trabajo, es de igual manera un JWT que se genera en el momento en el que el orquestador activa la fase raíz de un trabajo. Este token de trabajo es necesario para que las fases puedan suplantar al usuario en el momento de hacer operaciones autenticadas – por el momento, únicamente descifrar secretos de usuario.

Para ello, el *token* de trabajo (que queda modelado por la estructura `JobToken`) tiene las siguientes pretensiones:

- *Token ID*: mismo significado y pretensión que anteriormente.
- *UserName*: mismo significado y pretensión que anteriormente.

---

[56] "*Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew.*"

- *IssueDateUtc*: mismo significado y pretensiones que anteriormente.
- *ExpireDateUtc*: mismo significado y pretensión que anteriormente.
- *PublicKey*: contiene la clave pública del usuario. Más sobre esto en la sección *Fichero de secretos de usuario*. Se almacena bajo la pretensión privada `pkey`.

En los *tokens* de trabajo, la fecha de expiración es 24 horas posterior a la fecha de expedición. Este valor también es configurable. Pasado ese tiempo, el token deja de ser considerado como válido, por lo que su uso para suplantar al usuario fallará, haciendo fallar con él el trabajo completo. Aunque este valor es también configurable, 24 horas parece un margen razonable de tiempo para que un trabajo termine.

5.7.3 Fichero de secretos de usuario

Un usuario puede almacenar secretos en el orquestador. Estos secretos están pensados para guardar información sensible que no debería ser compartida, y, por tanto, no debería ser publicada o incluida en el control de versiones, al que todo el equipo tiene acceso. Por ejemplo, nombres de usuario y contraseñas de máquinas remotas, claves API de sistemas Cloud…

De esta manera, y referenciando los secretos de usuario, un fichero de definición de trabajo se puede parametrizar con información sensible sin que dicha información sea divulgada:

```
{
  "name": "Deploy Windows machine",
  "phaseId": "awsDeployPhase",
  "phaseEntryPointId": "deployFromTemplate",
  "inputs": {
    "awsAccessKeyId": "${secret:aws_access_key}",
    "awsSecretAccessKey": "${secret:aws_secret_key}",
    "awsDefaultRegion": "${secret:aws_region}",
    "deployScript": "${windowsDeployTemplate}",
    "deployTimeout": "${deployTimeout}"
}
```

El fichero de definición de trabajo puede ser incluido en versión de controles y compartido entre varios usuarios, que lo parametrizarán utilizando sus propios secretos de usuario.

La información sensible vive en el orquestador, y se descifra únicamente en el momento en el que se necesita mediante las claves del usuario.

5.7.3.1 Claves de cifrado

La información sensible, como ya se ha señalado, se encuentra en el orquestador. Para cifrar la información es necesario únicamente la clave privada, mientras que para descifrarla hacen falta tanto la privada como la pública. Esto es lo que determina cómo se distribuyen las claves.

Cuando el usuario se registra en el sistema, se genera un par de claves RSA. La clave privada (que se puede utilizar en solitario para cifrar datos) queda almacenada en el orquestador, en el fichero *user.json* del usuario correspondiente.

Por el contrario, la clave pública es enviada de vuelta al usuario en la respuesta a la petición de registro. Esta es la única vez que dicha clave está disponible, por lo que se recomienda encarecidamente al usuario copiarla y guardarla en un lugar seguro, pues esta clave es imprescindible para el manejo de secretos de usuario.

5.7.3.2 Almacenamiento y recuperación de los secretos

Cuando un usuario almacena un secreto en el orquestador, este se envía en texto plano. Una vez recibido, el orquestador lo cifra con la clave privada del usuario, recuperada de *user.json,* y lo almacena en el fichero *secrets.json*, que no es más que un diccionario clave/valor.

```
{
  "aws_secret_key": "sxdD0(...)R28YDA=="
}
```

A partir de ese momento, un usuario puede listar los secretos almacenados (en cuyo caso únicamente se recuperan las claves del diccionario) o leerlos, tal y como se vio en la sección *Sesión de ejemplo*. Para descifrar los secretos, se envía al orquestador la clave pública, el orquestador descifra el contenido, y lo manda de vuelta al cliente.

Esta implementación comparte los secretos y la clave pública del usuario en texto plano a través de la red. Los secretos permanecen cifrados únicamente en el almacenamiento del orquestador. Tanto el cifrado como el descifrado suceden pues del lado del orquestador. Esta decisión se tomó balanceando la seguridad de dichos

secretos frente a la facilidad de implementar aplicaciones cliente que se conecten al orquestador.

Es por esto por lo que se recomienda encarecidamente utilizar un certificado SSL para el acceso al API REST del orquestador. Aunque el certificado no esté firmado por una autoridad confiable, proporciona un mayor grado de seguridad en las comunicaciones HTTP que no usarlo en absoluto. El orquestador tiene la capacidad de crear un certificado auto-firmado y utilizarlo si se configura para ello. Sin embargo, esta opción no está habilitada por defecto porque la práctica totalidad de implementaciones HTTP rechazarán de manera automática (e incluso silenciosa) dicho certificado, dejando al orquestador inaccesible.

5.8 El API REST

La forma que tienen los usuarios de interactuar con el orquestador es a través del API REST. Cuando el orquestador arranca, levanta un servidor HTTP ligero que escucha peticiones en determinadas rutas. Para operar el orquestador, pues, los usuarios han de realizar peticiones HTTP de la forma:

```
<protocolo>://<dirección>:<puerto>/<ruta>
```

Donde `protocolo` será "*http*" o bien "*https*" (dependiendo de si se está usando o no un certificado), `dirección` será el nombre o dirección IP de la máquina donde se esté ejecutando el orquestador, el `puerto` será aquel que esté configurado para realizar las escuchas, y la `ruta` será una de las posibles rutas en las que el orquestador acepta peticiones.

5.8.1 Filosofía de diseño

Cada una de las rutas en las que el servidor escucha peticiones HTTP se conocen, comúnmente, como URL. Estas son las siglas de *Unified Resource Locator*, o *localizador de recursos uniforme*. El hecho de que una ruta identifique un recurso (una imagen o un documento en Web, pero también un usuario o un trabajo en REST) debería traducirse en que dichas rutas contengan únicamente sustantivos, no verbos. Este es el principio básico que ha guiado la definición de rutas del API REST del orquestador. Cada entidad se identifica por un verbo (*user*, *job*, *secrets*), y, en caso de que haya una jerarquía en

las entidades (por ejemplo, un secreto pertenece a un usuario, y un estado de trabajo pertenece a un trabajo concreto) la ruta refleja dicha jerarquía.

Algunos ejemplos que reflejan dicha jerarquía son:

- `/api/v1/user`
- `/api/v1/user/tokens`
- `/api/v1/user/secrets`
- `/api/v1/user/secrets/{secretName}`

Adicionalmente, todas las rutas del API comienzan por el prefijo `/api`, para diferenciarlas en el futuro, en caso de que sea necesario, de otras partes de la aplicación (ver sección *Añadir una interfaz Web para el manejo del orquestador*). Seguidamente hay un identificador de versión (actualmente, todas las rutas son `/v1`). Esto permite, en caso de que fuese necesario, versionar cada ruta por separado, de manera que se pueden introducir cambios para ofrecer nueva funcionalidad bajo la misma ruta manteniendo la compatibilidad hacia atrás a través de la versión anterior.

Como las rutas identifican recursos, y no contienen verbos, para expresar la intención de una llamada sobre un recurso se utiliza un verbo HTTP. El estándar define (entre otros) los siguientes verbos (RFC 7231, sección 4.3[57]):

- *GET*: el mecanismo primario para transmisión de información. En REST se usa para obtener un recurso.
- *POST*: el cliente pide al servidor que procese el contenido de la petición siguiendo las semánticas de la aplicación. En REST se usa para lanzar acciones o procesos.
- *PUT*: el cliente pide al servidor que cree un nuevo recurso, o que sustituya uno existente. En REST se usa para crear nuevas instancias.
- *PATCH*: el cliente pide al servidor que modifique un recurso existente.

[57] https://datatracker.ietf.org/doc/html/rfc7231#section-4.3

- *`DELETE`*: el cliente pide al servidor que elimine un recurso existente.

Estos verbos HTTP se utilizan en conjunto con las rutas y sus sustantivos para expresar la intencionalidad de la petición. Así, por ejemplo, para listar los secretos no existe una ruta `/api/v1/user/secrets/list`, sino que se hace una petición HTTP GET a `/api/v1/user/secrets`. Para lanzar un trabajo, no existe una ruta `/api/v1/jobs/launch`, sino que se hace una petición HTTP POST a `/api/v1/jobs`.

Adicionalmente, HTTP define códigos de respuesta para indicar el resultado de procesar la petición. En el API REST del orquestador se utilizan únicamente los siguientes:

- *`200 OK`*: indica éxito en el procesado de la petición.
- *`201 Created`*: indica que un recurso se ha creado correctamente.
- *`400 Bad Request`*: indica que la petición estaba mal formada.
- *`401 Unauthorized`*: indica que no se tiene permiso para acceder a un recurso.
- *`500 Internal Server Error`*: indica un fallo no recuperable en el procesado de la petición.

Además del código de respuesta HTTP, todas las respuestas del API incluyen, en el cuerpo, un código de respuesta específico de la aplicación. Estos códigos de respuesta se pueden consultar en el apéndice *Documentación del API REST*. Salvo en el caso de que la petición haya terminado de manera correcta (código 0, OK), el resto de los códigos dan una descripción exacta de cuál ha sido el error.

Estos códigos propios de la aplicación permiten dar un detalle más específico que el de HTTP. Además, el código HTTP 200 indica que la petición se ha procesado correctamente, lo cuál no quiere decir, necesariamente, que se haya realizado la acción que el usuario esperaba. Por eso, el código HTTP 500 se reserva para transmitir al cliente fallos no controlados en el manejo de la petición, mientras que los fallos que sí están controlados se retornan con código HTTP 200 acompañado del código de error específico de la aplicación.

5.8.2 Operaciones

Las operaciones implementadas en el API REST permiten llevar a cabo la operativa descrita en la presente memoria.

- Crear usuarios.
- Autenticar usuarios (mediante nombre de usuario y contraseña, para obtener los correspondientes *tokens* de acceso y de refresco, que serán necesario adjuntar en las operaciones que requieran que el usuario se identifique).
- Listar, leer y escribir secretos de usuario.
- Lanzar trabajos, listarlos, y consultar el estado o parar un trabajo concreto.

Durante el desarrollo del API REST se elaboró, de manera simultánea, una documentación básica que se encuentra disponible en el repositorio de código debajo del directorio `/src/server/web/areas/restapi/apidoc.md`. Debido a que es documentación, y por completitud de esta memoria, se adjunta completa en el apéndice *Documentación del API REST*.

Capítulo 6 - El servicio

En las siguientes secciones se detallan los aspectos más relevantes de la segunda pieza de software desarrollada en este Trabajo de Fin de Máster – el *daemon* o servicio. Al ser una pieza accesoria al propio orquestador, no se entrará en tanto detalle como en el primero.

6.1 Propósito

El orquestador es un software diseñado desde una perspectiva neutra, agnóstica del propósito final para el que se vaya a utilizar. Es extensible mediante fases implementadas por terceros, proporcionando el orquestador únicamente la infraestructura para la operativa necesaria: descubrimiento de fases, activación de trabajos y gestión de estos, y soporte para usuarios y secretos de usuario. Además, se plantean para el futuro nuevas funcionalidades que siguen igualmente este diseño neutro (para más información, consultar la sección *Trabajo futuro*). Sin embargo, el ámbito de ejecución del orquestador queda limitado a la máquina local en la que se esté ejecutando.

El servicio es una manera de extender las capacidades del orquestador a otras máquinas de una forma simple. El servicio ofrece, pues, una interfaz unificada para ejecutar tareas en otras máquinas (por ejemplo, subir y bajar ficheros, pero también ejecución de procesos, y, en realidad, lo que necesite el desarrollador), independientemente del Sistema Operativo que se esté ejecutando en essas máquinas.

Los requisitos de uso desde el orquestador son mínimos. Para que una fase pueda beneficiarse del servicio, únicamente necesita:

- Enlazar la biblioteca que incluye el código cliente del servicio. Como se vio en la sección *Estructura del proyecto*, esta biblioteca se encuentra localizada en el proyecto bajo `/src/daemon/daemonclient`.
- Poder establecer, desde la máquina donde se esté ejecutando el orquestador, una conexión TCP contra la máquina donde se esté ejecutando el servicio.

La conexión se realiza a través de la dirección IP de la máquina, y el puerto donde esté escuchando el servicio.

6.2 El protocolo RPC

Se describen a continuación los detalles más importantes de la implementación del protocolo de invocación remota a métodos, siguiendo para ello el curso natural de una petición. Establecer la conexión, negociar las capacidades del protocolo, identificar el método a ejecutar, transmitir los argumentos desde el cliente al servidor, y transmitir el resultado desde el servidor hacia el cliente (que, en caso de error, puede ser una excepción).

6.2.1 Establecimiento de la conexión y negociación de las capacidades

El protocolo RPC se estructura en forma de cliente-servidor. La aplicación servidor escucha peticiones en un puerto concreto. Cuando un cliente quiere invocar uno o más métodos (una vez negociada la conexión, esta se mantiene abierta hasta que una de las dos partes la cierra, bien explícitamente y de manera ordenada, bien por inactividad en la conexión para poder liberar sus recursos) establece una conexión contra el servidor.

El protocolo está preparado para ser versionado y compatible hacia atrás. El cliente primero envía al servidor el número de versión máxima soportada por este. El servidor responde entonces con el número de versión del protocolo que se va a utilizar a partir de ese punto, que puede ser el mismo que declaró el cliente, o uno inferior si el servidor no lo soporta.

El cliente espera a leer la versión del protocolo que se va a utilizar, y manda entonces sus capacidades. El servidor, sin embargo, no espera a nada, y envía sus capacidades inmediatamente después de mandar el número de versión del protocolo que se va a utilizar.

Las capacidades de cliente y de servidor indican funcionalidades extra que el protocolo puede soportar si se amplía para ello. Por ejemplo, puede servir para indicar si durante el resto de la conexión se va a utilizar compresión, qué algoritmo de compresión se va a utilizar, si se requiere autenticación a nivel de protocolo, etc.

Además, las capacidades pueden marcarse como "opcionales" u "obligatorias". De esta forma, cliente y servidor pueden acordar qué funcionalidades son deseables, pero no críticas (por ejemplo, compresión), y qué funcionalidades son críticas y, si no se soportan, no se desea continuar con la conexión (por ejemplo, cifrado SSL).

- Si ambas partes especifican una capacidad, bien de manera opcional u obligatoria, ambas partes pasarán a utilizarla.
- Si una parte especifica una capacidad de manera opcional, y la otra no la soporta, no se utilizará.
- Si una parte especifica una capacidad de manera obligatoria, y la otra no la soporta, la negociación fallará y se abortará la conexión.

Ambas partes procesan las capacidades del otro de manera simultánea, por lo que ambos llegarán a la misma conclusión sobre las capacidades que se van a utilizar en el mismo punto de la negociación, tal y como se refleja en la *Figura 6-1. Establecimiento de la conexión*. De esta forma no es necesario realizar más intercambio de mensajes, cliente y servidor han acordado qué características del protocolo van a activar, y lo que es más importante: todos los mensajes de red enviados hasta ese momento ya han sido consumidos, por lo que ambos pueden saltar a utilizar por encima otras formas de comunicación (compresión, SSL) sin problemas.

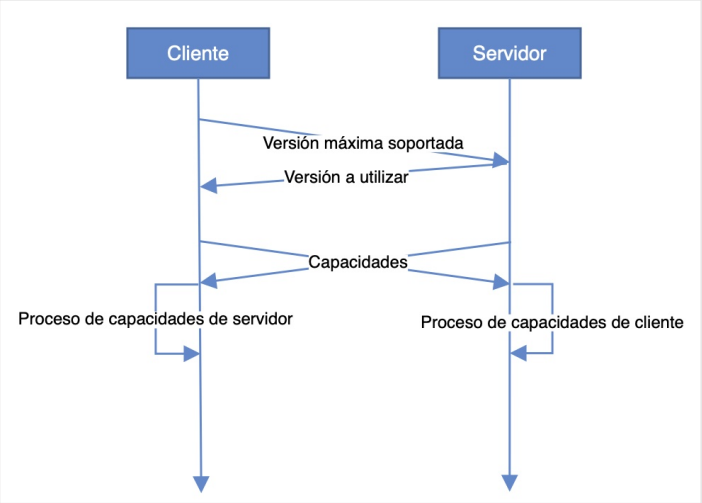

Figura 6-1. Establecimiento de la conexión

Pasado ese punto, la conexión se considera establecida.

6.2.2 Llamada a métodos remotos

Los métodos disponibles en el RPC se dividen en interfaces, que proporcionan una serie de servicios agrupados por funcionalidad. Tanto cliente como servidor disponen de la misma definición de interfaz (por ejemplo, `IFileSystemAccess`). El servidor proporcionará para esa interfaz la implementación real (`FileSystemAccess`) y una forma de acceder a ella a través del protocolo RPC, el "*stub*" (`FileSystemAccessStub`). El cliente implementará la manera análoga de acceder a esta funcionalidad a través del RCP también, el "*proxy*" (`FileSystemAccessProxy`).

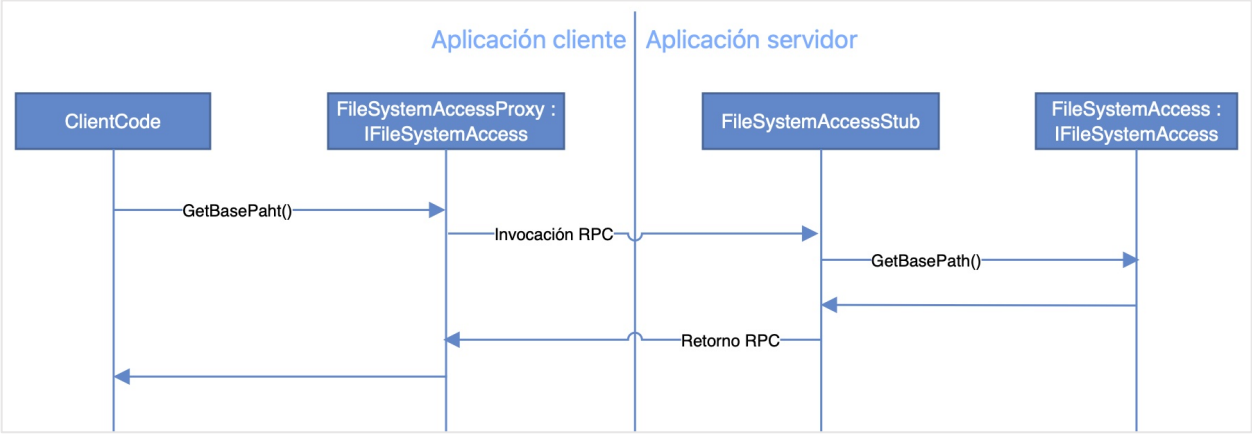

Figura 6-2. Llamada RPC de lado a lado

Los métodos se identifican por dos `unsigned short`, que consisten en método y versión. De esta forma, se pueden versionar métodos en caso de ser necesario (por ejemplo, sobrecarga). El cliente escribe primero en la red el identificador de método, seguido por los parámetros necesarios para su invocación.

La serialización de los parámetros se hace de manera manual, utilizando un *BinaryWriter*[58]. De esta forma, si se quiere escribir por ejemplo una lista de *Int32*, primero se escribiría la longitud de la lista, y luego cada uno de sus miembros, por orden. No existe ninguna restricción que impida utilizar cualquier otro mecanismo de serialización (basado, por ejemplo, en reflexión). Se escogió este por sencillez y rendimiento (como ya se ha señalado en otros puntos de esta memoria, la reflexión es un mecanismo

[58] https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=net-5.0

computacionalmente costoso, y otros mecanismos más rápidos como *Google Protocol Buffers* requieren pasos adicionales para funcionar que no se detallarán aquí).

El servidor leerá el identificador de método, y con ello obtendrá el *stub* correspondiente. Seguidamente, cederá el control al *stub* para que este realice la deserialización de los parámetros de manera adecuada e invoque al código real. Finalmente, el *stub* escribirá en la red el valor de respuesta (los métodos que no retornan nada, de tipo *void*, también tienen un valor de respuesta, para señalar que la invocación ha terminado de una manera uniforme). Cuando el proxy lea la respuesta, la devolverá de igual forma a las capas superiores.

Un ejemplo de llamada lado a lado se puede observar en la *Figura 6-2. Llamada RPC de lado a lado*.

6.2.3 Propagación de excepciones

En caso de que se produzca una excepción durante el procesado del método, el servidor se lo indicará al cliente con un código de error, seguido de la excepción serializada. Como las excepciones pueden ser de cualquier tipo derivado de *System.Exception*, para serializarlas y deserializarlas sí que se utiliza un mecanismo basado en reflexión ofrecido por la propia plataforma. Cuando la excepción llegue al cliente, este sencillamente la lanzará.

Propagar la excepción desde el servidor hasta el cliente permite que este segundo implemente los mecanismos de control de excepciones de una manera abstraída del RPC que existe por debajo. Además, existe la ventaja añadida de que información como la traza original del fallo en el servidor también se propaga al cliente.

6.3 Funcionalidades implementadas

Al término de este Trabajo de Fin de Máster, las funcionalidades que se han implementado en el servicio son limitadas. Establecen pues un ejemplo de cómo seguir expandiendo la funcionalidad del servicio y una demostración de su potencial, más que un producto completamente operativo y con un amplio rango de funcionalidades listas para ser utilizadas.

6.3.1 Interacción con el sistema de ficheros

El servicio puede interactuar con el sistema de ficheros de la máquina donde se esté ejecutando mediante los siguientes métodos RPC, definidos en la interfaz `Orchestrator.Daemon.Common.Interfaces.IFileSystemAccess`.

El servicio maneja el concepto de directorio base (que se consulta mediante el método `GetBasePath` y se modifica mediante el método `SetBasePath`). El directorio base es la raíz a partir de la cuál se realizarán el resto de las operaciones, y ha de corresponderse con un directorio existente en la máquina.

El servicio puede crear ficheros vacíos (mediante el método `CreateFile`), con un tamaño reservado determinado (sin escribir ningún contenido) y pudiendo sobrescribirlo en caso de que existiese (de forma que se pueden truncar también ficheros). Es útil en caso de que la existencia del fichero en sí mismo tenga algún significado para el desarrollador. Así mismo, puede eliminar ficheros (mediante el método `DeleteFile`).

Se puede de igual forma crear un único directorio (método `CreateDirectory`) o crear un conjunto de directorios en una única llamada (método `CreateDirectories`). La creación de directorios crea todos los directorios especificados en una dirección hasta la raíz, por lo que para crear un árbol de directorios es suficiente con especificar las hojas, no hay que pedir explícitamente la creación de los nodos intermedios.

En cuanto a la subida y bajada de ficheros, hay que introducir el concepto de bloque.

Las operaciones de escritura (ficheros que el cliente envía al servicio) y de escritura (ficheros que el cliente lee desde el servicio) se hace en bloques. De esta forma, el sistema es más tolerante a fallos (si falla una lectura o una escritura puede retomarse en el bloque en el que falló, sin necesidad de repetir todos los bloques anteriores), y algunas operaciones (como la lectura sobre un fichero grande) pueden paralelizarse en el cliente.

Además, los bloques no se usan únicamente con ficheros grandes. También sirven para agrupar varios ficheros en un mismo bloque, de manera que en una única llamada RCP pueden leerse o escribirse varios ficheros pequeños (siempre y cuando el conjunto

no supere un determinado tamaño). Esto se implementa mediante los métodos `WriteFiles` (que escribe un conjunto de ficheros en un único bloque), `ReadFiles` (que lee en un único bloque un conjunto de ficheros), `WriteBytes` (que escribe un bloque de una determinada longitud en un determinado fichero, a una determinada distancia del inicio), y `ReadBytes` (que lee un bloque de una determinada longitud de un determinado fichero, a una determinada distancia del inicio).

En cuanto al tamaño de bloque, el servicio tiene configurado un tamaño en bytes máximo (por defecto, de 4MiB). Las operaciones de lectura y de escritura están optimizadas para reciclar búferes tanto como sea posible, por lo que el tamaño de bloque que maneja el servicio es, en realidad, el tamaño máximo de búfer que el servicio está dispuesto a reservar en memoria.

Es necesario balancear bien el tamaño de bloque: un tamaño demasiado pequeño causará que operaciones de subida grandes haya que dividirlas en muchas llamadas, causando un trabajo extra invertido en el mecanismo de RPC. Un tamaño de bloque grande causará que el servicio deba reservar más memoria para las operaciones, lo que, dependiendo del grado de utilización, puede ser problemático, y, dependiendo del tamaño de los ficheros a leer o escribir, innecesario (infrautilizando la memoria física de la máquina).

Sin embargo, el tamaño de bloque es negociable. El servicio impone un máximo, pero el cliente puede escoger utilizar un tamaño menor. El tamaño real que se usará como máximo puede ser consultado por el cliente mediante el método `NegotiateBlockSize`. El cliente envía el tamaño de bloque que desearía utilizar, y el servicio responde con el tamaño que acepta (que puede será igual o menor al sugerido por el cliente).

En las operaciones de escritura de ficheros los bloques de bytes que superen en tamaño al máximo aceptado por el servicio serán rechazados, y causarán que la operación falle. Negociar previamente el tamaño del bloque puede evitar este tipo de fallos. No hay inconveniente por otro lado en que en las operaciones de escritura el cliente utilice un tamaño de bloque menor al máximo soportado por el servicio.

De manera análoga, en las operaciones de lectura (ficheros que el cliente lee desde el servicio), el cliente puede pedir como máximo en cada lectura el tamaño de bloque. En las operaciones de lectura en las que se pidan bloques más grandes que el máximo soportado serán truncadas al tamaño máximo (aquí la operativa no falla), pero de igual forma se pueden pedir bloques más pequeños que el máximo en cada llamada.

Para terminar, el cliente puede pedir información al servicio sobre un fichero o sobre un directorio (que, por defecto, incluye información sobre todos los ficheros y directorios contenidos en este en el primer nivel, pero que, opcionalmente, puede pedirse de manera recursiva). En la petición de información de un fichero o de un directorio puede pedirse así mismo que el servicio calcule los resúmenes de los ficheros (*hash* mediante el algoritmo Md5, lo suficientemente simple y estable como para dar na idea correcta de la integridad del fichero). Esta información se pide mediante los métodos `GetFileInfo` y `GetDirectoryInfo`.

6.3.2 Información del servicio y de la máquina

Es deseable que los clientes puedan obtener del servicio información del propio servicio y de la máquina donde este se está ejecutando. Se discute en la sección *Aumentar las capacidades del servicio*.

De momento, a este respecto, el servicio implementa únicamente la capacidad de comprobar la conexión con el cliente mediante el método `Hello`, definido en la interfaz `Orchestrator.Daemon.Common.Interfaces.IInformation`. Si el servicio responde correctamente a la llamada, se le puede considerar preparado para trabajar.

Capítulo 7 - Conclusiones y trabajo futuro

Este Trabajo de Fin de Máster proporciona las bases para un orquestador software de propósito general. Para ello se han implementado, además del orquestador, un servicio o *daemon* que permite manejar máquinas remotas a través de una interfaz unificada, un protocolo RPC binario para comunicar al orquestador con el servicio, una aplicación de terminal cliente para manejar el orquestador, que accede a su funcionalidad mediante un API REST, y una aplicación cliente para el servicio, que permite acceder a su operativa con el propósito de poder hacer pruebas manuales.

Dicho orquestador puede ampliar su funcionalidad mediante la carga de fases implementadas por terceras partes. El mayor reto durante el desarrollo ha sido implementar la arquitectura que permite esta modularidad y flexibilidad al software; tanto la carga y descubrimiento de fases como su activación, en la pieza de código descrita en esta memoria como el motor de activación. Particularmente, el proceso de obtener los argumentos correctos con los que invocar los métodos adecuados, mediante deserialización perezosa del fichero de definición de trabajos, y resolución de variables de entrada y secretos de usuario a los valores correctos.

Para la extensibilidad, se ofrece una biblioteca con las dependencias software necesarias para que cualquier desarrollador pueda implementar dichas fases y cargarlas posteriormente en el orquestador. El lenguaje de programación escogido para ello ha sido C#. Dicha decisión responde, entre otras, al API asíncrono basado en *async* y *await*, y a la capacidad de carga dinámica de código mediante reflexión, dos características clave para la extensibilidad de la plataforma desarrollada.

La neutralidad en el diseño del orquestador permite su implantación en equipos y organizaciones para satisfacer un amplio abanico de casos de uso. Dichos casos de uso se amplían aún más cuando el orquestador se usa en conjunto con el servicio, que actualmente permite trabajar con el sistema de ficheros de una máquina remota y que, en el futuro, puede permitir tareas tales como ejecución y control de procesos, o arrancado y apagado remoto de máquinas. El desarrollo del protocolo de llamada a procedimiento remoto que permite comunicar al orquestador con el servicio ha sido el

segundo mayor reto, primándose la sencillez del funcionamiento a cambio de que el desarrollador, cada vez que quiera añadir un nuevo método, tenga una pequeña carga extra de implementación para enlazar las llamadas en los lugares necesarios.

Todas las piezas software que componen el ecosistema se han desarrollado con el menor número de dependencias posible, con el objetivo de disminuir los requisitos de despliegue y reducir la complejidad, para bajar la barrera de entrada al desarrollo y facilitar la adopción.

7.1 Trabajo futuro

Se detallan en las siguientes secciones posibles acciones a realizar en un futuro si se desea continuar evolucionando el proyecto del servidor orquestador de propósito general.

7.1.1 Distribuir los binarios

El primer paso de cara a facilitar la adopción del orquestador ha de ser distribuir los binarios, generados en versiones bien conocidas y marcadas como estables.

Para ello, puede utilizarse la misma plataforma en la que el código fuente ha sido alojado y compartido con las partes interesadas durante su desarrollo: el repositorio de GitHub. La plataforma de control de versiones de Microsoft ofrece la posibilidad de subir publicaciones o *releases*, referenciando una etiqueta o *tag* concreta del repositorio de Git asociado, y cualquier persona, disponga o no de cuenta de usuario, puede descargar estas publicaciones en su ordenador.

7.1.2 Inyección de dependencias de terceros

Como se ha señalado anteriormente, las capacidades de inyección de dependencias del orquestador son muy limitadas.

La inyección de dependencias se hace a través del activador de fases (clase `PhaseActivator`, método `AddService<T>`).

La primera limitación se encuentra en lo que se puede registrar como dependencia. Únicamente se permiten instancias concretas:

```
SomeClass instance = new SomeClass();
phaseActivator.AddService<SomeClass>(instance);
```

De esta forma, al poder registrar únicamente una instancia de un tipo, que durará toda la vida de la instancia del orquestador, lo que se tiene en realidad es un patrón *singleton* escondido.

Una de las funcionalidades básicas de un inyector de es ser capaz de establecer un grafo de dependencias y crear las instancias requeridas bajo demanda, instanciando a su vez las dependencias de estas, sin haber registrado en el propio inyector instancias concretas, sino simplemente los tipos:

```
phaseActivator.AddService<FirstClass>();
phaseActivator.AddService<SecondClass>();
```

De esta forma, si `SecondClass` dependiese a su vez de `FirstClass`, en el proceso de instanciar `SecondClass`, el inyector debería ser capaz de instanciar la dependencia `FirstClass`.

Además, para una correcta inversión de dependencias, el inyector debería permitir relacionar interfaces con su implementación concreta, de manera que las dependencias recaigan sobre dichas abstracciones, permitiendo un reemplazo de las implementaciones sencillo en caso de que fuese necesario:

```
// AddService<I, T>() where T : I
paseActivator.AddService<IInterface, InterfaceImplementation>();
```

Otra funcionalidad básica es la de permitir al desarrollador manejar el tiempo de vida de las instancias que se crean. Como se ha dicho antes, el inyector de dependencias actual es una forma de camuflar un patrón *singleton* (con la ventaja de que este se podrá cambiar a posteriori por un inyector de dependencias real sin afectar a las clases cliente). Un inyector de dependencias completo debería permitir escoger al desarrollador si la dependencia va a ser un *singleton*, si se debe instanciar de nuevo cada vez que se requiera, o un punto intermedio: por ejemplo, si las instancias tienen una caducidad determinada (basada en tiempo o en cualquier otra condición), si se debe tener un *pool* de ellas… incluso debería dejar proporcionar una manera concreta de instanciar la dependencia:

```
phaseActivator.AddSingleton<IInterface, InterfaceImplementation>();
phaseActivator.AddTransient<IInterface, InterfaceImplementation>();
```

```
phaseActivator.AddSingleton<IInterface>(interfaceImplementationInstance);
phaseActivator.AddTransient<IInterface>(() => new
  InterfaceImplementation(DateTime.UtcNow.AddDays(-1));
```

Sin embargo, el orquestador ya tiene un inyector de dependencias completo: el implementado por ASP.NET para instanciar los controladores que dan servicio al API REST a través del cuál se maneja el orquestador. Sin embargo, este inyector de dependencias es una implementación concreta de ASP.NET, y su API no permitía de manera sencilla reciclarlo para la activación de fases.

Por lo tanto, se deja como trabajo futuro implementar en el orquestador un inyector de dependencias flexible, con aquellas funcionalidades esperable para realizar una correcta inversión de dependencias, y que permita a terceras partes (a los desarrolladores de fases) inyectar sus propias dependencias.

7.1.3 Carga y descarga de bibliotecas en caliente

Mientras una biblioteca está en uso por una aplicación, esta queda (si el sistema de ficheros subyacente lo permite) bloqueada. Por lo tanto, a no ser que se cargue con técnicas de *shadow copying*[59], esta no se puede cambiar en disco mientras no se descargue (y, aún con *shadow copying* activado, a la hora de volver a cargar la nueva biblioteca sería necesario descargar la anterior para evitar colisiones en los tipos cargados).

Si el orquestador se destina a labores de integración continua en un equipo de desarrollo, pararlo significaría que las ramas se dejan de probar y de integrar, y que se deja de generar nuevas versiones del producto. Dependiendo de la cantidad de trabajo pendiente y del tamaño del equipo, esto puede no ser aceptable – hay equipos donde el sistema de integración continua trabaja las 24 horas del día, 7 días a la semana sin descanso.

La carga y descarga de bibliotecas en caliente permitiría expandir las funcionalidades del orquestador sin que por ello haya que pararlo, evitando la

[59] Copiar la biblioteca a una localización temporal, y cargar esta copia en lugar del fichero original.

necesidad de encontrar una ventana temporal de mantenimiento donde se bloquee el trabajo.

Además, poder cargar y descargar las bibliotecas en caliente puede ser útil para labores de depuración, facilitando el desarrollo de las fases al reducir el número de pasos necesarios para probar nuevos cambios.

En último término, puede además ser deseable que dicha carga y descarga suceda a través del API REST. De esta manera, un usuario no necesitaría acceso a la máquina en la que se ejecuta el orquestador para extender su funcionalidad.

7.1.4 Permitir únicamente la carga de bibliotecas confiables

Cargar código desconocido es siempre peligroso. Es uno de los motivos por los que, por ejemplo, las plataformas móviles (Android, iOS) no permiten (por defecto) la instalación de aplicaciones fuera de los orígenes confiables.

El orquestador está diseñado para ser ejecutado en ambientes de confianza, en equipos u organizaciones en los que se mantenga en control en todo momento cuáles son las bibliotecas que se cargan en el orquestador y quiénes son los usuarios que utilizan el sistema. Dicho de otra forma, el orquestador no está diseñado para ser una plataforma software abierta sin control.

Sin embargo, si en el futuro se desea que el orquestador pueda cargar y descargar bibliotecas sin necesidad de reinicios, y que dichas bibliotecas puedan ser cargadas directamente por los usuarios usando el API REST, puede implementarse un sistema de seguridad básico basado en la firma de las bibliotecas. Las bibliotecas pasarían de esta manera por un proceso de certificación y/o confianza.

Para la certificación, sería necesario que un administrador revisase previamente el código de la biblioteca (manualmente mediante un desensamblado, o automáticamente mediante un analizador). Si la biblioteca pasa el análisis con éxito, esta se firma con un certificado en el que el orquestador confíe, y que permitiría su carga.

Si por el contrario el proceso está basado en confianza, cada usuario dispondría de un certificado propio. Usando este certificado personal, cada usuario firmaría la

biblioteca, y el orquestador cargaría únicamente aquellas bibliotecas cuya firma sea confiable (es decir, que coincida con alguno de los certificados que el orquestador considera como seguros). En el momento en el que un administrador del sistema detecte de alguna forma que una biblioteca se está comportando de manera no deseada, revocar el certificado con el que se firmó y descargar la biblioteca garantizaría que el usuario que la firmó originalmente no puede subir ni esa, ni nuevas bibliotecas firmadas con el mismo certificado.

7.1.5 Permitir la ofuscación de las fases

En el desarrollo de software, la ofuscación es el acto deliberado de transformar el código fuente en una versión con una funcionalidad idéntica pero más difícil de leer y comprender por el ser humano. Es una forma de protección por oscuridad, lo que significa que puede ser eludida mediante técnicas tales como la ingeniería inversa. Esto no significa, sin embargo, que muchos equipos la sigan utilizando para proteger software comercial como una capa extra que, aunque no asegure por completo el secreto sobre el funcionamiento del software, sí añade una capa extra de protección.

Permitir ofuscar las fases podría proporcionar un incentivo a la profesionalización en el desarrollo de estas. Un escenario hipotético en el que existan equipos que diseñan, desarrollan y comercializan fases como modelo de negocio.

Sin embargo, como ya se ha visto en las secciones *El fichero de definición de trabajos* y *Activación de fases*, el paso de argumentos a los métodos de entrada de las fases se hace mediante la coincidencia en el nombre de los parámetros. Esto impide que se pueda ofuscar el nombre de los argumentos, lo que, dependiendo de la herramienta de ofuscación utilizada, puede impedir que se ofusque el método completo.

Una forma de circunvalar esta limitación sería mediante el uso de atributos para decorar también los argumentos de las funciones, de forma que el atributo marca su valor. Mientras que la implementación actual del método de entrada de una fase podría parecerse mucho al siguiente código…

```
[PhaseEntryPointId("deployFromTemplate")]
public async Task<PhaseResult> DeployFromTemplate(
```

```
  string awsAccessKeyId,
  string awsSecretAccessKey,
  string awsDefaultRegion,
  string deployScript,
  int deployTimeout,
  CancellationToken ct)
{
  // Code…
}
```

Si se introdujese un atributo `PhaseInputId` en el que se etiquetase el nombre, la signatura del método, una vez ofuscada, podría parecerse a lo siguiente:

```
[PhaseEntryPointId("deployFromTemplate")]
public async Task<PhaseResult> M_0(
  [PhaseInputId("awsAccessKeyId")] string s_0,
  [PhaseInputId("awsSecretAccessKey")] string s_1,
  [PhaseInputId("awsDefaultRegion")] string s_2,
  [PhaseInputId("deployScript")] string s_3,
  [PhaseInputId("deployTimeout")] int i_0,
  CancellationToken ct)
{
  // Code…
}
```

El token de cancelación no necesitaría un atributo, ya que la coincidencia de este argumento se busca por tipo.

No se implementó este atributo por la verbosidad y redundancia añadidas en caso de que no se ofusquen las fases. Como solución intermedia de compromiso, podría implementarse una heurística en la que primero se intente buscar coincidencia con el nombre del parámetro, y, si no se encuentra, se busquen coincidencias con el valor del atributo. De esta manera se permitirían los dos escenarios: hacer coincidir entradas con parámetros por nombre, tal y como funciona actualmente, y, opcionalmente, poder especificar el nombre con un atributo.

7.1.6 Permitir al usuario más de un par de tokens

La forma en la que se guardan los identificadores de los *tokens* expedidos al usuario impide tener, en cualquier momento dado, más de un token de refresco y un token de acceso.

Si un usuario necesita acceder al orquestador desde otra máquina distinta a la que usa habitualmente tiene dos opciones:

1. Copiar los ficheros de configuración de la aplicación cliente para utilizar los mismos *tokens* de acceso y de refresco.
2. Identificarse de nuevo con su nombre de usuario y contraseña para que el sistema expida un par nuevo de *tokens*.

En el segundo caso, identificarse nuevamente con usuario y contraseña invalidará los *tokens* expedidos con anterioridad, lo que obligará al usuario a identificarse de nuevo cuando vuelva a una máquina cliente anterior.

Pero la primera solución, en la que el usuario copia las credenciales entre máquinas, tampoco está exenta de este problema. Si el token de acceso caduca y se usa el de refresco para obtener un nuevo par, para volver a una máquina cliente anterior es necesario, de igual forma, copiar las nuevas credenciales.

Una posible solución es separar los datos de usuario de los datos que identifican al usuario, y guardar cada identificador de *token* en un fichero separado. Cuando el usuario se identifique con un *token* caducado, se puede borrar del almacenamiento el fichero que lo representa. Sin embargo, sería necesario hacer una recolección de basura periódica: si un usuario crea *tokens* de manera indefinida, y nunca se llega a autenticar con ellos (o no se intenta autenticar con ellos cuando ya están caducados), no será posible detectar qué ficheros representan *tokens* inválidos, por lo que no se limpiarían del sistema de ficheros. Sería en este caso necesario un trabajo en segundo plano que periódicamente se asegure de mantener el almacenamiento de *tokens* en un buen estado de salud.

7.1.7 Permitir más de una instancia del orquestador sobre los mismos datos

Únicamente una instancia del orquestador puede utilizar de manera simultánea los ficheros donde se guardan los datos de los usuarios. Esto es debido a que la seguridad en el acceso de lectura y escritura se maneja mediante bloqueos en memoria. Los bloqueos se comparten entre distintos hilos del mismo proceso, pero no entre procesos distintos.

Sin embargo, pueden existir situaciones en las que sea deseable tener más de una instancia del orquestador funcionando de manera simultánea. Por ejemplo, si se

realiza balanceo de carga mediante un proxy inverso, puede servir para realizar actualizaciones sin que por ello el servicio deje de estar disponible.

Esta posibilidad se ha contemplado durante el desarrollo, por lo que se ha abstraído en las interfaces `ILock` y `IResourceReaderWriterLock` las implementaciones concretas del mecanismo de bloqueos.

Una forma sencilla de implementar sincronización entre procesos en el acceso a los recursos del usuario es mediante el propio sistema de ficheros, creando ficheros especiales[60] que indiquen si un recurso está bloqueado para la escritura o para la lectura.

Las operaciones de creación y de borrado en el sistema de ficheros son atómicas, lo que significa que no existe posibilidad de que dos procesos creen el mismo fichero de manera simultánea – el primero que llegue lo podrá crear, mientras que todos los siguientes fallarán. Esta propiedad daría soporte al mecanismo de sincronización entre procesos basado en ficheros. Si existe un fichero determinado, es que el recurso está bloqueado (bien para lectura o bien para escritura, dependiendo, por ejemplo, del nombre del fichero). Mediante una espera activa con tiempo de expiración el proceso que vaya a obtener el bloqueo para la escritura comprueba si el fichero especial sigue existiendo. Si el fichero desaparece, se crea de nuevo (obteniendo el bloqueo), y se escribe en el recurso. Si el fichero sigue existiendo cuando expire el tiempo de espera, significa que el bloqueo no se ha podido obtener, y la operación falla.

7.1.8 Permitir la carga de ficheros en el orquestador

Es posible que algunas fases, dependiendo de la implementación, necesiten acceder a ficheros externos. Por ejemplo, a una plantilla de *Amazon AWS CloudFormation*. Si no se posee acceso a la máquina donde se está ejecutando el orquestador, la interacción con el sistema de ficheros de esta puede ser delicada – se

[60] Entiéndase como "*especial*" que tiene un significado especial para la aplicación, no que sean ficheros distintos a cualquier otro del sistema de ficheros.

corre el riesgo de sobrescribir ficheros que pertenezcan a otros usuarios o programas, o de acceder a datos a los que no se debería tener acceso.

Una forma de facilitar este escenario es permitiendo la carga de ficheros en el orquestador. Los ficheros estarían asociados al usuario que los subió. El orquestador permitiría la subida, bajada, listado y borrado de ficheros a través del API REST. Los ficheros tendrían asignado un nombre, al igual que los secretos de usuario, y se podrían referenciar en los ficheros de definición de trabajos mediante una sintaxis especial:

```
{
  (...)
  "cloudFormationTemplatePath": "${file:aws_template}"
  (...)
}
```

De la misma forma que el activador de fases resuelve los secretos y las variables durante la activación, resolvería la ruta real del fichero referenciado para pasársela a la fase en forma de parámetro.

Adicionalmente, se podrían implementar mecanismos de protección como el cifrado de los ficheros: los ficheros se cifrarían y descifrarían igual que los secretos de usuario. Únicamente en el momento en el que el fichero sea necesario para una fase, este se descifra (haciendo uso de la clave almacenada en el token de trabajo – para más información consultar la sección *Tokens de acceso, de refresco y de trabajos*) y se copia en plano a una ubicación temporal del sistema de ficheros. Una vez la fase que hacía uso del fichero termina, el fichero temporal es borrado y únicamente queda la copia cifrada. Este mecanismo no protegería completamente frente a fases malintencionadas que busquen ficheros desprotegidos, pero ofrecería una capa extra de protección frente a guardar los ficheros directamente en claro.

7.1.9 Añadir información extra a los trabajos

Actualmente, el fichero de definición de trabajos tiene pocos campos: el nombre del trabajo, sus entradas y su fase raíz. Los siguientes campos y su funcionalidad asociada podrían resultar de utilidad:

- Descripción: una descripción del trabajo que entre en detalle sobre el propósito de este.

- Tiempo de vida: siempre cabe la posibilidad de que una fase esté mal implementada y no se proteja frente a escenarios tales como un bloqueo. Añadir un tiempo de vida a los trabajos permitiría que el orquestador acabe con una tarea si esta no ha salido de manera ordenada después de un tiempo determinado.
- Visibilidad del trabajo: por defecto, todos los trabajos son públicos para todos los usuarios. Si se implementa un sistema basado en roles (consultar sección Restringir la operativa de los usuarios) puede ser interesante definir una visibilidad para el trabajo, restringir qué roles pueden conocer su el estado de su ejecución, cancelarlo, etc.
- Momento de lanzamiento: puede ser también interesante que los trabajos no se lancen inmediatamente, sino que dicha ejecución se pueda programar para un momento en el futuro, o incluso de manera recurrente. Tanto el trabajo programado para el futuro como el trabajo recurrente (este segundo antes y después de su ejecución) se encontrarían en un estado de "pendiente". Mediante la operativa actual de cancelación de trabajos, se podría cancelar para evitar su ejecución llegado el momento.

7.1.10 Añadir una interfaz Web para el manejo del orquestador

Actualmente, el orquestador se opera por parte de los usuarios únicamente a través de una API REST, mediante llamadas HTTP. Aunque existen herramientas gráficas para elaborar y ejecutar dichas peticiones, estas están pensadas más para tareas de depuración que para un uso cotidiano, estando pues las API REST enfocadas a ser consumidas por otras aplicaciones.

Una interfaz Web podría facilitar el uso del orquestador sin necesidad de conocimientos de programación o HTTP. Además, esta interfaz Web podría ser servida por el propio orquestador (al disponer ya de un servidor HTTP), y, para evitar duplicidades, la interfaz podría implementarse mediante una aplicación Web que ofrezca su funcionalidad consumiendo la misma API REST que ya está funcionando.

7.1.11 Restringir la operativa de los usuarios

A lo largo de esta memoria, el término "usuario" se ha utilizado para referirse a toda aquella persona que utiliza el orquestador, mientras que "administrador" hace referencia a aquella persona que opera el arranque y parada del orquestador, su configuración, y que tiene acceso a la máquina donde este se ejecuta.

Si se quiere continuar añadiendo funcionalidad tal como cargar bibliotecas con fases directamente a través del API REST, o subir ficheros, puede resultar de interés la posibilidad de restringir la operativa de los usuarios mediante roles y cuotas.

Un rol determinaría las capacidades que un usuario tiene en el sistema. Esto podría restringir el tipo de operativa a la que tiene acceso o los recursos a los que tiene acceso. Todos los usuarios que participen de un rol (por ejemplo, Administrador), heredarían las capacidades propias de ese grupo. Algunos ejemplos de capacidades que deberían poder ser configurables mediante el uso de roles podrían ser:

- La capacidad de consultar el estado de los trabajos, pero no de lanzarlos.
- La posibilidad de consultar únicamente el estado de los trabajos propios, pero no de los ajenos.
- La posibilidad de cancelar cualquier trabajo, tanto propio como ajeno.
- La posibilidad de subir o no subir bibliotecas que contengan fases.
- La posibilidad de usar o no la interfaz Web.

En cuanto a las cuotas, determinarían una *cantidad límite* sobre un determinado recurso u operativa. Por ejemplo:

- Cantidad máxima de secretos de usuario que se pueden almacenar.
- Cantidad máxima de ficheros, de tamaño de fichero, o de tamaño del conjunto de ficheros que se pueden almacenar.
- Cantidad máxima de trabajos que se pueden lanzar por unidad de tiempo (por ejemplo, límite de trabajos por día).
- Duración máxima de un trabajo en ejecución (por ejemplo, límite de treinta minutos por trabajo, antes de que este sea cancelado).

7.1.12 Aumentar las capacidades del servicio

Es deseable que los clientes puedan obtener del servicio información del propio servicio y de la máquina donde este se está ejecutando. Esta información debería cubrir, al menos:

- Tipo de Sistema Operativo y versión concreta.
- Tiempo transcurrido desde que la máquina arrancó.
- Información básica sobre hardware: modelo de procesador y arquitectura, núcleos disponibles, memoria física instalada, interfaces de red disponibles junto con su dirección IP y máscara de subred…
- Discos / volúmenes montados en la máquina, junto a su capacidad y los puntos de montaje.
- Usuario actual bajo el que se está ejecutando el servicio.

Además, sería de utilidad poder ejecutar procesos en la máquina remota. Entre las operaciones básicas para el manejo de procesos, estarían:

- Ejecutar un nuevo proceso.
- Consultar el estado de un proceso (en ejecución o terminado).
- Obtener el código de salida del proceso, una vez ha terminado.

Obtener la salida y error estándar del proceso, tanto durante su ejecución como una vez finalizado

Chapter - Introduction

In Computer Engineering, orchestration refers to the act of configuring, coordinating, and, in general, operating hardware and software systems in an automated manner. It includes the definition of a workflow in order to achieve specific goals. It is intimately related to concepts such as version control, machine provisioning, and automatic testing and delivery.

With the explosion in popularity of "architecture as a service" (provided by cloud computing services), the "infrastructure as code" services (such as Terraform or CloudFormation) that offer automation in creating infrastructure are highly demanded. Additionally, since more than ten years ago, there are orchestration services such as Jenkins or TeamCity, focused on automatic integration and delivery.

The orchestration process can be focused (among other goals) in operating these services (that exists independently to each other) as a whole, orchestrating them together with a more complex goal than the one that each one of these services can achieve on their own.

Motivation

Regardless of the software product a team or company develops, we most probably can distinguish some common needs. The most important one will be delivering tested versions of their product.

The steps to generate these tested versions can be similar or completely different across products and teams. Most of them will probably obtain a copy of the source code from a repository. But maybe not all of them need to build that code. Maybe not all of them run the same kind of tests or deliver the product using the same channel or platform.

There are many ways to develop software. But in all of them, rather sooner than later the need to automate a part or all the process will appear. There are many possible causes for that: an increase in the complexity of the process, in the time it requires, in how many times it needs to run each day…

At first, and maybe because how easy they are, this automation will rest on scripts. If the team is careful, these scripts will always run on the same machine, and they will be

preserved under version control. However, as time passes, these scripts might begin to multiply and mutate. Additionally, the build and deliver process might involve complex scenarios that require parallelism, synchronization, error management… the earlier scripts won't be able to survive – they were not designed for this complexity, anyway.

At this point, the team will look in the market for a software that can be adapted to their needs, and then will have to rewrite a huge part of the already implemented flow in order to adapt it to said software. If the team had used since the beginning a framework that allowed them to build their automations on top of a solid and extensible product, they wouldn't have faced the necessity to migrate.

The goal of this Master Thesis is to define and implement said framework.

Goals

These are the main and secondary goals that guided the implementation of this Master Thesis.

Main goals

The main goal is to design and implement a **multi-user, extensible, orchestration software**. It will act as a framework so any team or individual can build on top of it the automation of their tasks in an easy way.

This orchestrator will execute **jobs**. A job is the necessary series of ordered steps to achieve a goal (for example, generating a new version of a software product).

These jobs are defined by the **job definition file**, a plain text file in JSON. This file reflects the input variables for the job, and the series of steps that the job must run.

The **phases** will be the code implementation of each one of the job's steps. The phases will be implemented by developers using the framework and can be as specific or generic as the developer wants.

To implement these phases, the developer will have a library that provides the necessary dependencies (common attributes, classes, and methods) defining the orchestrator public API. Once the developer implements their own phases in a library, it can be loaded into the orchestrator as an extension. Then, the user can reference these

phases in their job definition file and request their execution by using the REST API that the server exposes.

Secondary goals

Because the orchestrator is meant to be used by anyone, it is impossible to foresee the needs of its users. And these users are the ones that will extend the orchestrator's capabilities by implementing new phases in extensions.

A secondary but important goal is preventing any possible future issue caused by the orchestrator's dependencies. The amount of software a user would need to install in a machine in order to develop or to use the orchestrator is selected as a measure of the success of this secondary goal.

Developing the following software around the orchestrator is also a secondary goal.

A client application for the orchestrator

First, a command-line application that allows remote access to the orchestrator by using its REST API.

The orchestrator can be operated by using any application able to understand the HTTP protocol. This is because the REST API works through HTTP. However, developing a client application is a secondary goal of this thesis in order to protect the user from the initial complexity of the API.

A daemon for accessing and operating remote machines

Second, an agent or daemon that is meant to run in any machine that the orchestrator needs in order to run its jobs. The purpose of this agent is providing a unified interface to the user to execute certain tasks, regardless of the underlying Operating System. This way, implementing tasks such as uploading or downloading files, or running remote processes, is identical across different families of Operating Systems such as Unix-like or Windows. Without the daemon, the implementation of a task for each OS would vary depending on the connection protocol and the available applications.

A lightweight Remote Procedure Calling protocol

Third, a simple and lightweight Remote Procedure Calling protocol. This protocol makes a minimum use of the network thanks to a custom object serialization and deserialization, and minimum overhead to stablish the connection and executing each remote method. This RPC protocol is meant to be used to communicate the daemon and the orchestrator.

Using an already existing RPC solutions such as RabbitMQ or implementing serialization using frameworks like Google Protocol Buffers would have been both quicker and simpler. However, developing the protocol alongside its serialization complies with the goal of having a self-contained project with minimum dependencies.

A client application for the daemon

Fourth and last, a minimal command-line client application that allows the user to interact with the daemon without the need to develop a daemon phase. This eases the manual testing of the actions the user might need to implement in a phase, without the extra complexity of the necessary steps to run them on the actual orchestrator.

Work plan

The work plan defines the necessary stages to achieve the goals above.

All the work was meant to be done by a single person. Therefore, in order to ease development, a strict work methodology was not imposed. The general principles of Agile Software Development were followed however, by quickly iterating once the general requirements were in place, aiming to have deliverable software as early as possible (*"Working software is the primary measure of progress"*).

Stage 0 – initial proposal

The initial proposal on what to implement was well defined since the very beginning. Because of this, there was no place for a requirement elicitation stage. All the requirements regarding application functionality were defined in the initial proposal.

Stage 1 – Orchestrator core and RPC protocol

The parts of the application with a higher uncertainty regarding their success were prioritized since the beginning of the development process. This way, the most important

road blockers (if any) could have been identified early enough in order to shift courses if necessary.

Because of this, the first piece of software to be implemented was the RPC protocol and the phase activation engine. The former communicates the daemon, the orchestrator, and the daemon command-line client. The latter instantiates the classes that implement the phases and invokes their methods with the necessary parameters.

Stage 2 – REST API to operate the server and polishing

Once a first draft of the RPC protocol was in place and having an activation engine able of taking a JSON file and running what was defined on it, it was time to implement the upper layers (regarding abstraction) of the orchestrator. The daemon was left to be implemented in a later stage because it is a simpler piece of software.

In this second stage, the different REST methods that allow operating the orchestrator were implemented. These endpoints allow the user to run and cancel jobs, retrieve progress, create users, and manage secrets. During this phase, the activation engine and the RPC protocol were under active development too, with the goal of adding more features, fixing bugs, and fine tuning their behavior.

Stage 3 – Interlayer communication and dynamic extension loading

Once the orchestrator is able of managing users, secrets, and jobs, the project is considered to have reached a third stage in which the main goal is to integrate the different layers that form the application. The REST API must be able of truly launching jobs. The jobs must be able to impersonate users in order to decipher the necessary secrets. The phases must be able of reporting back their progress. The users must be able of stopping jobs in case they need to. It is necessary to integrate the pieces developed in earlier stages: the REST API, the activation engine, the user management… Additionally, once all the different components are now well defined, it is the moment to divide them across different libraries. This allows users to import the orchestrator public API into their code in order to develop phases that later will be loaded into the orchestrator as extensions.

Stage 4 – Daemon and Thesis

Once the orchestrator is able of correctly running the jobs requested by the users, it is time to extend its capabilities to other machines. For that, and using the RPC protocol, the development of the daemon starts in a fourth stage.

Thesis structure

Capítulo 2 - Estado de la cuestión summarizes the common features of the existing orchestration and automation software.

Capítulo 3 - Lenguaje de programación, entorno de desarrollo y dependencias de terceros introduces the decisions regarding development tools and environment. It also presents the third-party dependencies and the chosen programming language.

Capítulo 4 - Estructura del proyecto, ejecución de las pruebas y ejecución de las aplicaciones introduces the structure of the project, and the necessary steps to navigate and build the different software components that are part of it.

Capítulo 5 - El orquestador presents the orchestrator, which is the main piece of software implemented on this Master Thesis. This chapter also walks through some of the most important implementation details of said orchestrator. *Capítulo 6 - El servicio* introduces the daemon, which is an optional extension of the orchestrator.

Finally, *Capítulo 7 - Conclusiones y trabajo futuro* presents the conclusions of this Master Thesis and presents a few possible lines of future work.

Chapter - Conclusions and future work

This Master Thesis provides the foundation for a general-purpose orchestration and automation framework. This framework consists, on top of the orchestrator server, on a daemon or agent that allows operating remote machines through a unified interface, an RPC protocol that communicates the server and the daemon, and a client application to operate the orchestrator.

The orchestrator is designed to extend its functionality by loading phases implemented by its users and third-party developers. The biggest challenge developing the orchestrator was designing the architecture that supports this modularity with flexibility. This makes the activation engine the most complex and important piece of software implemented in this Master Thesis. It provides support to the operations of discovering, instantiating, and invoking the code that represents a phase. To do so, the activation engine must deserialize the job definition file in a special, custom way, by solving variables and secrets as necessary, to pass down the correct arguments to the phases.

The users and third-party developers have the public API of the orchestrator available on an isolated library. By referencing and implementing this public API anyone can implement phases that the orchestrator can load. The programming language of choice for this was C#. It was chosen because of its asynchronous API (based on async and await), and its reflection mechanism, two crucial features for the extensibility of the developed framework.

The orchestrator design is neutral, so it is fit to be used on a big number of different use cases. These possibilities increase once the orchestrator and the daemon work together. Now the daemon can only interact with its host's File System, but in the future, it can offer a wider range of features. For example, it can support running processes, or starting and shutting down machines. The development of the Remote Procedure Calling protocol that communicates the orchestrator and the daemon was the second biggest challenge of this Master Thesis. The protocol remains simple, in exchange of some extra

manual labor to wire RPC calls from the client to the actual functionality implementation on the server.

Every piece of software that is part of this orchestration and automation framework was developed with the least possible number of dependencies. The goals of this are keeping deployment requirements to a minimum to ease adoption and reducing the complexity to lower the entry barrier for new developers.

Future work

These are some possible lines of work to keep evolving this orchestration and automation framework.

Distribute the binaries

The first step to ease orchestrator adoption should be distributing the binaries. These binaries should be generated on well-known, stable points of the repository.

GitHub, the same place where the source code lives, can be used to distribute the binaries as well. This Microsoft-branded online repository based on Git offers the possibility of uploading software releases and referencing from each release the point of the repository where the release was generated on. Anyone can access these releases, regardless of them having a user account.

Third party dependency injection

Right now, the orchestrator implements a very restricted dependency injection mechanism. Dependency injection happens on the phase activation engine (`PhaseActivator` class, `AddService<T>` method).

The first limitation is that a user can only register instances as dependencies.

```
SomeClass instance = new SomeClass();
phaseActivator.AddService<SomeClass>(instance);
```

For each type, only one instance can be registered. This instance will live for as long as the orchestrator does, hiding a *singleton* pattern.

One of the most basic features of a dependency injection mechanism is being able of stablishing a type dependency graph. The injector then uses said graph to instantiate both the required instances of a type and its dependencies:

```
phaseActivator.AddService<FirstClass>();
phaseActivator.AddService<SecondClass>();
```

This way, if `SecondClass` depends on `FirstClass` (a dependency that in no way is specified when registering types), the injector should be able of instantiating a `FirstClass` instance first and then using it to instantiate `SecondClass`.

In addition to that, and to achieve a correct dependency inversion, the injector should allow the user to pair interface types with the class type that implements them. This way, a user can swap different implementations for an interface without having to modify the code that depends on the interface:

```
// AddService<I, T>() where T : I
phaseActivator.AddService<IInterface, InterfaceImplementation>();
```

Another desirable feature is allowing the user to choose the lifecycle of each created instance. As stated before, right now the dependency injector hides a singleton pattern. A proper dependency injector should allow singletons as well, but along other possibilities such as creating a dependency once for each different job, or even providing a specific way of instantiating dependencies:

```
phaseActivator.AddSingleton<IInterface, InterfaceImplementation>();
phaseActivator.AddTransient<IInterface, InterfaceImplementation>();
phaseActivator.AddSingleton<IInterface>(interfaceImplementationInstance);
phaseActivator.AddTransient<IInterface>(() => new
  InterfaceImplementation(DateTime.UtcNow.AddDays(-1));
```

The orchestrator already has a proper dependency injector in place – the one provided by ASP.NET to instantiate REST API controllers. The disadvantage of this is that using said injector ties dependency injection with ASP.NET, and its API does not allow reusing it for phase activation easily.

It is left as future work implementing a flexible dependency injector with the most common features a user might expect to properly implement the dependency inversion principle.

Loading and unloading extensions without restarting the orchestrator

When an application is using a DLL (at least on Windows) it is left blocked (if the underlying File System allows blocking files). This means that unless the library is loaded using shadow copying, it cannot change on disk until the application stops using it.

If the orchestrator is for example used for continuous integration tasks on a development team, stopping it would mean that branches stop getting built, tested, and integrated, and that no new versions of the product get generated. Depending on the amount of pending work and the size of the team, this might not be acceptable – there are teams whose CI system works 24/7.

Loading and unloading libraries without stopping the orchestrator would allow to expand its functionality without needing to find a maintenance timeframe in which the work the orchestrator carries on gets blocked.

Moreover, this feature could ease developing and debugging new phases by reducing the number of necessary steps to test new changes.

Lastly, this load and unload could happen through the REST API. This way, a user would not need access to the machine in which the orchestrator is running to extend its capabilities.

Allow trustworthy extensions only

Loading unknown code is always dangerous. It is one of the reasons that mobile platforms such as Android or iOS don't allow, by default, installing applications from outside trustworthy sources.

The orchestrator is designed to run in trustworthy environments, by teams that keep under control which extensions are loaded by the orchestrator, and who are the users of the system. In other words, the orchestrator is not meant to be a software platform opened to a wide, uncontrolled audience.

But that could be the case if the libraries that implement the phases go through a security mechanism based either on certification or trust.

For certification, an administrator would need to review the code of the library (either manually by disassembling it, or automatically by using a software analyzer). If the library passes the certification process, it is then signed with a certificate in which the orchestrator trusts. The orchestrator would only load libraries signed by this trusted certificate.

On the other hand, on a security system based on trust, each user would have their own certificate. By using this personal certificate, users would sign their libraries, and then again, the orchestrator would only load libraries signed with a trusted certificate. If a system administrator detects that a library is misbehaving, they could revoke the certificate used to sign it, and then unload it from the system. Because the personal certificate of a user was revoked, said user can no longer upload the same library, nor sign new versions of it to be uploaded.

Allow phase obfuscation

In software development, obfuscation means transforming on purpose the source code of an application to an equivalent version that behaves the same, but that is harder to read and understand by human beings. Obfuscation is a way of protection by obscurity, which means that it can be defeated by techniques such as reverse engineering. This however does not mean that obfuscation is no longer used. Many teams trust in obfuscation not to completely protect their applications, but to raise the bar of the necessary set of abilities to do so. Obfuscation is not the only protection, but an extra layer.

Allowing developers to obfuscate their phases could incentivize implementing these phases in a professional way. This opens a hypothetical scenario in which designing and implementing phases is a business model.

However, as stated in sections *El fichero de definición de trabajos* and *Activación de fases*, the orchestrator uses name matches to pass down to the phase code the correct parameters. This prevents obfuscating method parameters, which, depending on the obfuscation tool, might mean preventing the obfuscation of the whole code.

A way of overcoming this could be by using attributes on the arguments of the functions. The attribute would reflect the actual name of the argument. This is a valid phase entry point:

```
[PhaseEntryPointId("deployFromTemplate")]
public async Task<PhaseResult> DeployFromTemplate(
  string awsAccessKeyId,
  string awsSecretAccessKey,
  string awsDefaultRegion,
  string deployScript,
  int deployTimeout,
```

```
  CancellationToken ct)
{
  // Code…
}
```

By using a `PhaseInputId` attribute to indicate the name, the method signature, once obfuscated, could look as follows:

```
[PhaseEntryPointId("deployFromTemplate")]
public async Task<PhaseResult> M_0(
  [PhaseInputId("awsAccessKeyId")] string s_0,
  [PhaseInputId("awsSecretAccessKey")] string s_1,
  [PhaseInputId("awsDefaultRegion")] string s_2,
  [PhaseInputId("deployScript")] string s_3,
  [PhaseInputId("deployTimeout")] int i_0,
  CancellationToken ct)
{
  // Code…
}
```

The `CancellationToken` does not need an attribute, as the match is made by type rather than by name.

This attribute was not implemented due to the verbosity and redundancy it adds. As an intermediate solution to solve this, the orchestrator could implement the following heuristic: first, try to match parameter name. If no argument matches the parameter, then check if it has a `PhaseInputId` attribute, and if so, use it to find the argument.

Allow more than two tokens per user

The way the orchestrator saves the IDs of the expedited tokens prevents the user from having at any given time more than a single pair of access token and refresh token.

If a user needs for any reason access to the orchestrator from a different machine from which they usually use, they have two options:

1. Copy the client application's configuration across machines in order to use the same access and refresh tokens.
2. Identify themselves again with their username and password so the system expedites a new pair of tokens.

If they choose the second option, all the already emitted tokens will expire. This will force the user to identify themselves again once they go back to their usual computer.

But the first solution in which the user copies the tokens is not exempt from nuisances either. If the access token expires and the refresh one is then used to obtain a new pair of tokens, once the user goes back to the previous computer it will be necessary to copy the tokens again, as the old refresh token is no longer valid.

A possible solution to this issue is splitting user data from the data used to identify the user and save each token ID in a separate file. When the user identifies themselves with an expired token, the orchestrator can then delete the file that represents said token. However, it would be necessary to make a periodic garbage collection. If a user creates tokens indefinitely, and they never use the created tokens (or they never try to use them once they expire), detecting which files represent invalid tokens is not possible with the suggested approach. A background job could periodically review and clean unnecessary files to keep the storage in a healthy state.

Allow more than one orchestrator instance using the same data

Only one orchestrator instance can use the same user data files at the same time. This is because the access to said files is synchronized by memory locks. Locks are shared across different threads within the same process but not across different processes.

However, there could be situation in which it is desirable to have more than one orchestrator instance working simultaneously. For example, by balancing load using a reverse proxy the orchestrator can be upgraded without stopping.

This possibility was contemplated during development. The `ILock` and `IResourceReaderWriterLock` are abstractions of the underlying interlocking mechanism.

A simple way to implement inter-process synchronization to access resources in the File System is using the FS itself, by creating special files that indicate whether a resource is locked for reading or for writing operations.

The create and delete operations in a File System are atomic, meaning that there is no possibility of two processes creating the same file at the same time. The first process doing the creation call will succeed, whilst subsequent attempts will fail. This feature could support a sync mechanism based on files. If a specific file exists, it means that a resource is locked. By running an active wait with a timeout, the process trying to acquire the lock would check if the special file still existed. If the file disappears, then the resource is free

again and the process could take the lock. If the timeout expires, then the read or write operation would fail.

Allow loading files into the orchestrator

Depending on their purpose, some phases might need access to files in the File System. For example, to read an Amazon CloudFormation template file. If the user does not have access to the machine where the orchestrator is running on, interacting with the File System might be delicate. Users might overwrite legitimate files from other users, or access sensitive data they should not.

This scenario could be better supported by allowing users to load files into the orchestrator. Files would then be associated with the user who uploaded them. The orchestrator would allow uploading, downloading, listing, and deleting files through the REST API. Files would have a name or alias, just as user secrets do, and users could reference files from the job definition file using a special syntax:

```
{
  (...)
  "cloudFormationTemplatePath": "${file:aws_template}"
  (...)
}
```

Just like with user secrets and variables during phase activation, the activation engine would solve the actual file path of the referenced file to pass it down to the phase as a parameter.

Additionally, there could be a security mechanism like the one implemented for user secrets. Files would be encrypted when the user uploads them, then decrypted just in time for phase execution. This mechanism doesn't protect user data completely, as malicious phases could still look for unprotected files both in memory and in the File System, but it would offer an extra protection layer than just writing files in clear.

Add extra information to the jobs

Now the job definition file has few fields: the job name, its inputs, and its root phase. The following fields and their associated functionality could be of use:

- Description: A description of the job going into detail regarding its purpose.

- Time to live: There is always the possibility of a bug in a phase. This bug could block phase execution indefinitely. Adding a TTL to the jobs would allow the orchestrator to cancel a job if it didn't end in an orderly manner after some time.
- Job visibility: by default, jobs are only visible for the user who launches them. If a mechanism for role-based security gets implemented (check section *Restrict user capabilities*), it might be interesting to define a visibility for a job, define which roles can check job's progress, cancel the job, etc.
- Start date: it might be useful to have jobs scheduled for the future, or even running these jobs recurrently. Both the scheduled and recurrent jobs would be in a "pending" status when they are added to the system. The jobs would get removed from the system by using the current job cancelation operation.

Add a web interface to operate the orchestrator

Now the only way to operate the orchestrator is by using its REST API through HTTP calls. Although there are graphic tools to craft and send this kind of request, they are engineered for debugging, not for day-to-day use. REST APIs are designed to be consumed by other applications, not by end users.

A Web interface could ease operating the orchestrator to those users who are not comfortable running a command-line application, or that are not proficient in programming their own tools. This Web interface could be served by the orchestrator itself, and, to prevent feature duplication, it could work by invoking the already existing REST API.

Restrict user capabilities

In this document, the term "user" has been used to refer to every person who uses the orchestrator, whilst "administrator" refers to the person who starts and stops the orchestrator, configures it, and has access to the machine it runs on.

Being able of restricting what a user can and cannot do might not be a priority now that the orchestrator implements a limited functionality. But once said features expand with the option to load and unload libraries, and upload and download files

through the API, restricting the features a user has access to might be something of interest for the administrators. This could be implemented with a system based on roles and quotas.

A role would determine the features a user can access. All the users with a given role would inherit the associated feature access. Some features that can be limited by roles could be:

- Being able to check the status of a job, but not being able of running a new one.
- Being able to check the status of a job owned by the user, but not by others.
- Being able of cancelling any job, both the ones owned by the user and by other users.
- Being able or not of uploading libraries containing phases.
- Being able or not of using the Web interface.

A quota determines a limit amount over some resource or operation. Some quota examples could be:

- The maximum number of secrets a user can store.
- The maximum number of files a user can store. Or the maximum size for a file, or the maximum size for the total of the files.
- The maximum number of jobs a user can launch by time unit (for example a limit on the jobs per day).
- The maximum time a job can take to complete before it is canceled (for example, 30 minutes per job).
- The maximum time a user can take to run their jobs per day (for example, up to 30 minutes per day).

Extend daemon features

Clients should be able of retrieving from the daemon information regarding the host machine. This information should be, at least:

- Operating System and its version.
- Machine uptime.

- Basic hardware information: CPU model and architecture, available cores, physical memory, network interfaces…
- Mounted disks, along with used and free capacity, and mount point.
- Current user the daemon is running under.

Additionally, the daemon should be able of running process in the remote machine. Among the basic operations we would find:

- Spawn a new process.
- Check a process' status.
- Get the exit code and standard output and error from a process once it is finished.

Apéndice A - Ejemplo del fichero de configuración del orquestador

```
{
    "Urls": [
        "http://*:9999"
    ],
    "Certificate": {
        "UseCertificate": false,
        "CertificatePath": "orchestrator.pfx",
        "CertificatePassword": "some-random-password"
    },
    "AssembliesToLoad": [
        "self",
        "/Users/sluisp/gitrepos/ucm-orchestrator-
tfm/src/externalphases/bin/Debug/net5.0/externalphases.dll"
    ],
    "ApiMessages": {
        "GenericError": "There was an error trying to perform the operation. Try
again later or contact an administrator",
        "AuthenticationUserAlreadyExists": "Either the creation data is wrong, or the
user already exists",
        "AuthenticationUserDoesNotExist": "Either the authentication data is wrong,
or the user does not exist",
        "AuthenticationMissingData": "Missing user or password",
        "AuthenticationNameBreaksPolicy": "The user must be at least three characters
long, and can only contain letters and numbers",
        "AuthenticationPasswordBreaksPolicy": "The password must be at least eight
characters long, and must contain at least one lowercase letter, one uppercase
letter, and one number",
        "AuthenticationInvalidPassword": "Either the authentication data is wrong, or
the user does not exist",
        "AuthenticationInvalidAccessToken": "Either the authentication data is wrong,
or the user does not exist",
        "AuthenticationInvalidRefreshToken": "Either the authentication data is
wrong, or the user does not exist",
        "AuthenticationCorruptData": "Data corruption detected whilst reading user
information. Please contact a system administrator.",
        "UserSecretsNoSecrets": "The user has no secrets",
        "UserSecretsMissingData": "Missing secret name or secret value",
        "UserSecretsNameAlreadyExists": "A secret with the specified name already
exists",
        "UserSecretsNameDoesNotExist": "No secret with the specified name exists",
        "UserSecretsCouldNotEncryptData": "Could not encrypt user secret",
        "UserSecretsCouldNotDecryptData": "Could not decrypt user secret",
        "UserSecretsSecretOverridden": "User secret overridden",
        "JobsErrorLaunchingJob": "Could not launch the job due to an internal error",
```

```
        "JobsMissingData": "Either the job definition lacks a root phase or the job
token could not be created",
        "JobsJobDoesNotExist": "Either the job does not exist or it belongs to a
different user"
    },
    "ApiCodes": {
        "Unknown": -1,
        "Ok": 0,
        "GenericError": 10,
        "AuthenticationMissingData": 100,
        "AuthenticationInvalidData": 101,
        "AuthenticationBreakingPolicyData": 102,
        "AuthenticationCorruptData": 103,
        "UserSecretsNoSecrets": 110,
        "UserSecretsMissingData": 111,
        "UserSecretsNameAlreadyExists": 112,
        "UserSecretsNameDoesNotExist": 113,
        "UserSecretsCouldNotEncryptData": 114,
        "UserSecretsCouldNotDecryptData": 115,
        "UserSecretsSecretOverridden": 116,
        "JobsErrorLaunchingJob": 120,
        "JobsMissingData": 121,
        "JobsJobDoesNotExist": 122
    },
    "ApplicationPaths": {
        "BasePath": "./data"
    },
    "UserPolicy": {
        "NameRegEx": "^[a-zA-Z0-9]{3,}$",
        "PasswordRegEx": "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(.*){8,}$"
    },
    "SaltedPassword": {
        "SaltLength": 16,
        "PasswordLength": 20,
        "HashIterations": 10000
    },
    "Jwt": {
        "SecretKey": "xQcBDRvnJY4DRqE6RrEhFWs6WTRS83yy",
        "AccessTokenLifeTime": "0.00:00:30.000",
        "RefreshTokenLifeTime": "30.00:00:00.000",
        "LifeTimeCheckPrecission": "00:00:01.000"
    },
    "Encryption": {
        "KeyLength": 2048
    }
}
```

Apéndice B - Documentación del API REST

Se adjunta aquí la documentación del API REST elaborada durante a modo de referencia para su consulta durante el desarrollo de la aplicación cliente, escrita en inglés, en texto plano. Está formateada como documento *Markdown*[61], por lo que se puede transformar con cualquier herramienta compatible de manera sencilla a HTML o PDF.

El formato *Markdown* permite además poder explorar el documento correctamente formateado en el repositorio de código de GitHub directamente desde un navegador Web.

Se adjunta a la memoria por completitud, al tratarse de documentación, para su revisión y consulta.

```
# Orchestrator REST API

The following document describes how to use the orcherstrator REST API.

## Authentication

The Orchestrator REST API requires user authentication, except when specified
otherwise (`AllowAnonymous` tag). User authentication is done either using Json
**Web** Tokens or Basic authentication.

### JSON Web Tokens (JWT) authentication

The REST API uses JWT for authentication for most operations, except when
creating a new user (as the user still does not exist, and thus, using tokens is
not possible), and when obtaining tokens without a valid refresh token (when the
user accesses the system for the first time, or when the user's refresh token is
outdated).

When the endpoint requires an access JWT, it is specified using the
`Bearer(AccessToken)` tag. When the endpoint requires a refresh JWT, it is
specified using the `Bearer(RefreshToken)` tag.

To add the token to a request, use the `Authorization` header. For the
value, use `Bearer <token>`. The token must either be the `AccessToken` or the
```

[61] Lenguaje de marcado ligero desarrollado para dar formato a texto manteniendo la legibilidad, diseñado para ser usado fácilmente con editores de texto plano, y desarrollado originalmente por John Gruber y Aaron Swartz en 2004.

````
`RefreshToken`, depending on the situation. Client applications should renew
credentials silently and automatically using the `RefreshToken`, and store both
tokens (`AccessToken` and `RefreshToken`) for accessing the API and renewing
credentials in the future.

### Basic authentication

The REST API requires Basic authentication for renewing tokens, in case the
refresh token is outdated. Basic authentication makes the password travel the
network in plain text (unless using SSL). For this reason, user name or password
should **never** be stored client side. User name and password should be asked
for to the user only in exceptional situations, and rely on `AccessToken` for
access and `RefreshToken` for refreshing authentication data in the future.

To add user and password to a request, use the `Authorization` header. For the
value, use `Basic <credentials>`. The `credentials` field uses the following
format:

```text
base64(<user>:<password>)
```

For more information regarding Basic authentication, refer to
[RFC7617](https://tools.ietf.org/html/rfc7617).

## User handling

The following endpoints define how to handle users.

### Create user

Call this endpoint when you need to create a new user in the system.

**Facts**:

```text
Method:             POST
Endpoint:           /api/v1/user
Authentication:     AllowAnonymous
```

**Request body**:

```json
/* Body */
{
    "user": <string>,
    "password": <string>
}
```

**Response body**:

```json
/* Body */
````
````

```
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "publicKey": <string?>
}
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `102`: AuthenticationBreakingPolicyData
- `103`: AuthenticationCorruptData

Authenticate user

Call this endpoint when you need to generate a pair of access and refresh tokens. Use this endpoint either with `Basic` authorization, or `Bearer(RefreshToken` authorization.

Facts:

```text
Method:             GET
Endpoint:           /api/v1/user/tokens
Authentication:     Basic | Bearer(RefreshToken)
```

Response body:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "accessToken": <UserToken?>,
    "refreshToken": <UserToken?>
}

/* UserToken */
{
    "token": <string>,
    "issueDateUtc": <Date>,
    "expireDateUtc": <Date>
}
```

Response codes:

- `-1`: Unknown error

````
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `102`: AuthenticationBreakingPolicyData
- `103`: AuthenticationCorruptData

## User secrets

The following endpoints define how to handle user's secrets.

### List user secrets

Call this endpoint when you need to list the user secrets in the system.

**Facts**:

```text
Method:             GET
Endpoint:           /api/v1/user/secrets/
Authentication:     Bearer(AccessToken)
```

**Response body**:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "secretNames": <string[]?>
}
```

**Response codes**:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `110`: UserSecretsNoSecrets

### Write user secrets

Call this endpoint when you need to create a new user secret in the system, or
when you need to override an existing user secret.

**Facts**:

```text
Method:             POST
Endpoint:           /api/v1/user/secrets/{secretName}
````
````

````
Authentication:     Bearer(AccessToken)
```

**Request body**:

```json
/* Body */
{
    "secretValue": <string>,
    "overrideIfExists": <bool>
}
```

**Response body**:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "secretValue": <string?>
}
```

**Response codes**:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `112`: UserSecretsNameAlreadyExists
- `114`: UserSecretsCouldNotEncryptData
- `116`: UserSecretsSecretOverridden

### Read user secrets

Call this endpoint when you need to read an existing user secret from the
system.

**Facts**:

```text
Method:             GET
Endpoint:           /api/v1/user/secrets/{secretName}
Query parameters:
                    publicKey (url-encoded, mandatory)
Authentication:     Bearer(AccessToken)
```

**Response body**:

```json
````

```
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "secretValue": <string?>
}
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `110`: UserSecretsNoSecrets
- `113`: UserSecretsNameDoesNotExist
- `115`: UserSecretsCouldNotDecryptData

Jobs

The following endpoints define how to handle jobs.

Launch a job

Call this endpoint when you need to launch a new job on the system.

Facts:

```text
Method:             POST
Endpoint:           /api/v1/jobs
Authentication:     Bearer(AccessToken)
```

Request body:

```json
/* Body */
{
    "publicKey": <string>,
    "jobDefinition": <JobDefinition>
}

/* JobDefinition */
{
    "inputs": <Dictionary<string, object>?>,
    "rootPhase": <JobPhaseDefinition>
}

/* Job phase definition */
{
    "name": <string>,
```

```
    "phaseId": <string>,
    "phaseEntryPointId": <string>,
    "inputs": <Dictionary<string, object>>,
    "outputs": <Dictionary<string, string>>
}
```

Response body:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "message": <string?>,
    "jobId": <string?>
}
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `120`: JobsErrorLaunchingJob
- `121`: JobsErrorMissingData

List jobs

Call this endpoint when you need to list the jobs running in the system.

Facts:

```text
Method:             GET
Endpoint:           /api/v1/jobs/all
Authentication:     Bearer(AccessToken)
```

Response body:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "runningJobs": <RunningJob[]?>
}

/* RunningJob */
{
    "jobId": <string>,
```

```
    "jobName": <string>,
    "startDateUtc": <Date>,
    "status": <enum<Status>>
}

/* enum<Status> */
"unknown"
"running"
"finishedSucceeded"
"finishedFailed"
"finishedCancelled"
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData

Get specific job status

Call this method when you need to get the detailed status of a specific job.

Facts:

```text
Method:             GET
Endpoint:           /api/v1/jobs/{jobId}
Authentication:     Bearer(AccessToken)
```

Response body:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>,
    "jobId": <string?>,
    "jobName": <string?>,
    "startDateUtc": <Date?>,
    "status": <enum<Status>>,
    "progress": <RunningJobPhaseProgress?>
}

/* RunningJobPhaseProgress */
{
    "description": <string>,
    "startDateUtc": <Date>,
    "status": <enum<Status>>,
    "isIndeterminateProgress": <bool>,
    "progressPercent": <float>,
```

```
    "childrenProgress": <RunningJobPhaseProgress[]?>
}

/* enum<Status> */
"unknown"
"running"
"finishedSucceeded"
"finishedFailed"
"finishedCancelled"
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `122`: JobsJobDoesNotExist

Stop a job

Call this method when you need to stop a running job.

Facts:

```text
Method:             DELETE
Endpoint:           /api/v1/jobs/{jobId}
Authentication:     Bearer(AccessToken)
```

Response body:

```json
/* Body */
{
    "ok": <bool>,
    "responseCode": <int>
}
```

Response codes:

- `-1`: Unknown error
- `0`: OK
- `10`: Generic error
- `100`: AuthenticationMissingData
- `101`: AuthenticationInvalidData
- `103`: AuthenticationCorruptData
- `122`: JobsJobDoesNotExist

Full response codes catalogue

```
- `-1`: Unknown error
  - Indicates an error that went unhandled. It represents a logic error within
    the application. Should be reported back to an administrator, if possible.
- `0`: OK
  - Indicates success.
- `10`: Generic error
  - Indicates an error handling the request that floated up uncontrolled until
    the very last moment. Unlikely to appear. Should be reported back to an
    administrator, if possible.
- `100`: AuthenticationMissingData
  - Indicates that a request lacks the necessary data to authenticate the user.
- `101`: AuthenticationInvalidData
  - Indicates that a request contains invalid data to authenticate the user.
- `102`: AuthenticationBreakingPolicyData
  - Indicates that the data specified in the request to authenticate the user
    (most likely, data to signup a user) contains fields that violate the
    imposed policy (either the user name or the password).
- `103`: AuthenticationCorruptData
  - Indicates that the data stored for the user into the system is corrupt and
    could not be read. Should be reported back to an administrator, if possible.
- `110`: UserSecretsNoSecrets
  - Indicates that the system did not found any secrets for the user.
- `111`: UserSecretsMissingData
  - Indicates that a request lacks the necessary data to operate with user
    secrets.
- `112`: UserSecretsNameAlreadyExists
  - Indicates that there is an already existing secret identified by the
    provided name.
- `113`: UserSecretsNameDoesNotExist
  - Indicates that there is no secret identified by the provided name.
- `114`: UserSecretsCouldNotEncryptData
  - Indicates that there was an error when encrypting user secrets.
- `115`: UserSecretsCouldNotDecryptData
  - Indicates that there was an error when decrypting user secrets.
- `116`: UserSecretsSecretOverridden
  - Indicates that a user secret was overriden when writing new data.
- `120`: JobsErrorLaunchingJob
  - Indicates that there was an error launching a job.
- `121`: JobsErrorMissingData
  - Indicates that a request lacks the necessary data to operate with a job.
- `122`: JobsJobDoesNotExist
  - Indicates that the job specified in the request does not exist.
```