7. Plataformas para el Desarrollo de Servicios

7.1. Introducción y objetivos

En este tema se estudian algunas de las plataformas que nos ayudan a desarrollar sistemas distribuidos basados en servicios.

En primer lugar, estableceremos el contexto de la ingeniería de servicios, aclarando conceptos clave como servicio, arquitectura orientada a servicio, servicio web o microservicio entre otros. Detallaremos las principales características de arquitecturas basadas en *representational state transfer* (REST) y los estándares para modelar interfaces o contratos de servicios.

Posteriormente, veremos algunas de las plataformas y herramientas que facilitan el desarrollo de servicios, ya que aportan automatización o simplifican el proceso de definición de API, generación de documentación, emulación o interacción. Finalmente, enumeraremos algunas plataformas adicionales que simplifican la creación de esqueletos o de librerías de cliente para implementar o consumir servicios.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

* Comprender el contexto, el valor y la utilidad que tiene el desarrollo de sistemas distribuidos basados en servicios.
* Conocer los principales estándares de definición de servicios.
* Introducir algunas herramientas de automatización para la documentación, emulación, interacción de servicios.
* Conocer algunas plataformas aceleradoras del desarrollo de sistemas distribuidos basadas en servicios.

7.2. Contexto de las plataformas para el desarrollo de servicios

En el ámbito de la ingeniería del software, existen una serie de conceptos relacionados con la creación de sistemas distribuidos basados en servicios.

Un servicio software es una descripción de una o más operaciones que utilizan (múltiples) mensajes para intercambiar datos entre un proveedor y un consumidor. El efecto típico de una llamada de servicio es que el consumidor obtiene alguna información y/o modifica el estado del sistema o componente proveedor. Los servicios pueden tener diferentes atributos y pueden ser de diferentes categorías, se describen mediante una interfaz y su visión desde el punto de vista del consumidor (firma y semántica) se denomina «interfaz bien definida» o contrato (Josuttis, 2007).

La ingeniería de servicios, también llamada ingeniería de software orientada a servicios, es un proceso de ingeniería de software que intenta descomponer el sistema en unidades autoejecutables que realizan servicios o exponen servicios (servicios reutilizables). Las aplicaciones orientadas a servicios están diseñadas en torno a servicios débilmente acoplados, lo que significa que existen estándares y protocolos simples que todos los interesados siguen, mientras que detrás de ellos hay una amplia variedad de servicios tecnológicos que pueden ser mucho más complejos. Los servicios reutilizables, a menudo, son proporcionados por muchos proveedores de servicios diferentes, en los que todos colaboran dinámicamente con los usuarios de servicios y los registros de servicios (Fox, 2013).

Una arquitectura orientada a servicios (*service oriented architecture,*SOA) es aquella diseñada para construir aplicaciones comerciales como un conjunto de componentes de caja negra débilmente acoplados y orquestados para ofrecer un determinado nivel de servicio que de soporte a una serie de procesos de negocio (Bloor et al., 2009). En SOA hay tres tipos de actores:

* Proveedores de servicios: son servicios de software que publican sus capacidades (interfaces o contratos) y su disponibilidad en los registros de servicios.
* Usuarios de servicios: son sistemas de software (que pueden ser servicios en sí mismos) que utilizan los servicios proporcionados por los proveedores de servicios. Los usuarios de servicios pueden utilizar los registros de servicios para descubrir y localizar los proveedores de servicios que necesitan.
* Registros de servicios: son catálogos de información en constante evolución que se pueden consultar para ver qué tipo de servicios están disponibles.

Un servicio web (*web service,*WS) es un sistema de software diseñado para admitir la interacción interoperable de máquina a máquina a través de una red. Tiene una interfaz descrita en un formato procesable por máquina, como Web Service Description Language (Alhajj y Rokne, 2014). Otros sistemas interactúan con el servicio web de la manera prescrita por su descripción utilizando mensajes y siguiendo el protocolo Simple Object Access Protocol(W3, s. f.), normalmente transmitidos mediante HTTP (protocolo de transferencia de hipertexto) con una serialización XML junto con otros estándares relacionados con la web (W3, 2004).

Los microservicios, también conocidos como arquitectura de microservicios, son un estilo arquitectónico que estructura una aplicación como una colección de servicios que son:

* Fácilmente mantenibles.
* Débilmente acoplados.
* Desplegables de forma independiente.
* Organizados en torno a las capacidades empresariales.
* Bajo la responsabilidad de un equipo de desarrollo pequeño.

A diagram of a software application

Description automatically generated

Figura 1. Ejemplo de arquitectura de microservicios. Fuente: Microservice Architecture, s. f.

La arquitectura de microservicios permite la entrega rápida, frecuente y confiable de aplicaciones grandes y complejas. También permite que una organización evolucione su pila de tecnología. En ocasiones, se confunde el concepto de microservicio con el de servicios desplegados en contenedores, pero que forma parte de un sistema cliente-servidor monolítico global.

Para que hablemos de microservicios, en realidad, cada uno debe verse como un proyecto independiente, que tiene su propio ciclo de vida y modelo de despliegue y que, en ocasiones, interactúa con otra colección de servicios para satisfacer un objetivo de negocio común.

*Representational state transfer* (REST) es un estilo de arquitectura de software que se creó para guiar el diseño y desarrollo de la arquitectura para la *World Wide Web*. REST define un conjunto de restriccionessobre cómo debe comportarse la arquitectura de un sistema hipermedia distribuido a escala de Internet, como la web. El estilo arquitectónico REST enfatiza la escalabilidad de las interacciones entre los componentes, las interfaces uniformes, la implementación independiente de los componentes y la creación de una arquitectura en capas para facilitar el almacenamiento en caché de los componentes para reducir la latencia percibida por el usuario, reforzar la seguridad y encapsular los sistemas heredados (Fielding, 2000).

REST se basa en recursos, lo que significa que en un servicio REST trabajaremos con cosas para definir el servicio. Los clientes pueden utilizar este recurso para acceder a un identificador de recursos uniformes (URI) único y luego el recurso se devuelve a la persona que llama. REST usa sustantivos (es decir: usuario, país, estado, etc.) para definir qué tipo de recurso debe proporcionar el servicio. La representación de los recursos no está ligada al servicio en sí, lo que significa que la representación del recurso tiene un formato libre y se puede representar mediante varios tipos de formatos de datos (es decir, JSON, XML, CVS, etc.).

A continuación se muestra un ejemplo de definición de servicio REST:

![A close-up of a computer code

Description automatically generated]()

El estilo de arquitectura REST tiene seis restricciones, que son reglas de diseño aplicadas a la arquitectura y que establecen la característica distintiva de los servicios REST. Para definir un servicio de arquitectura REST, debemos tener las siguientes características arquitectónicas que hacen de cualquier servicio web una verdadera API RESTful:

* Interfaz uniforme: el servicio debe tener una interfaz definida entre el cliente y el servidor. Simplifica y desacopla la arquitectura, lo que permite que cada parte evolucione de forma independiente.
* Cliente-Servidor: la separación de preocupaciones entre cliente y servidor. A los clientes no les preocupa el almacenamiento de datos porque es un rol de servidor. Del mismo modo, los servidores no se preocupan por la interfaz de usuario porque es un rol de cliente.
* Sin estado: los servidores no almacenan ningún contexto de cliente. Cada solicitud de cliente debe contener toda la información requerida al servidor para comprender lo que se solicita. Por lo tanto, el estado de la sesión se mantiene completamente en el lado del cliente.
* Caché: para mejorar la escalabilidad y el rendimiento, se debe aplicar el almacenamiento en caché en los recursos cuando corresponda y así reducir la interacción cliente-servidor. El almacenamiento en caché se puede implementar en el lado del cliente o del servidor
* Sistema en capas: REST permite utilizar una arquitectura de sistema en capas, lo que significa que el servicio se puede implementar en un servidor y el almacenamiento de datos se puede ubicar en un servidor diferente. El cliente no puede saber si está o no conectado al servidor final.
* Código bajo demanda: los clientes pueden personalizar su aplicación para devolver un código ejecutable. Un cliente puede llamar a una API REST para obtener un código de representación del *widget*de la interfaz de usuario (UI).

7.3. Valor y utilidad de las plataformas para el desarrollo de servicios

El principal valor del uso de una arquitectura basada en servicios es la capacidad de crear sistemas distribuidos (ISO, 2017), cuyos componentes están ubicados en diferentes computadoras en red, las cuales se comunican y coordinan sus acciones pasándose mensajes entre sí desde cualquier sistema para lograr un objetivo común.

El desarrollo de sistemas distribuidos basados en servicios proporciona una serie de beneficios como:

* Separación de preocupaciones entre los diferentes servicios: cada servicio dentro de la aplicación ejecuta un grupo reducido de tareas y aprovecha la infraestructura de hardware que garantiza su mejor rendimiento.
* Bajo acoplamiento de los servicios: inspirado en el paradigma de diseño orientado a objetos, que reduce el acoplamiento entre clases para apreciar un entorno en el que las clases se pueden cambiar sin romper la relación existente, SOA fomenta mucho el desarrollo de servicios independientes que interactúan a través de interfaces o contratos.
* Desarrollo simultáneo y autónomo: cada servicio puede ser desarrollado, implementado y mantenido por un equipo independiente con mínima influencia en el resto del sistema.
* Combinación de diferentes lenguajes y tecnologías: cada servicio puede escribirse en el lenguaje que mejor se adapte a su finalidad y se comunica con el resto a través de un protocolo de comunicación estandarizado.
* Independencia de la plataforma: a través de protocolos estándares es posible desarrollar una aplicación formada por múltiples servicios que interoperan a pesar de estar creados con diversas tecnologías y desplegados en diversas plataformas.
* Escalabilidad: podemos disponer de múltiples instancias de los servicios desplegados en diversas infraestructuras, por lo que ante peticiones masivas de servicios, el sistema es capaz de escalar más fácilmente repartiendo la carga entre los entornos operativos.
* Confiabilidad y resiliencia: la posibilidad de disponer de múltiples instancias del mismo servicio replicadas en diversas infraestructuras computacionales, se aumenta la robustez del sistema ante caídas parciales del mismo.
* Reutilización de código: dado que los servicios representan paquetes de código autónomos, permiten la reutilización del código a nivel macro, lo que puede reducir significativamente el tiempo de desarrollo de nuevas funciones.

Las diversas plataformas y herramientas existentes facilitan el desarrollo de sistemas distribuidos basados en servicios, ofreciendo beneficios en base a funcionalidades como:

* Soporte para el modelado de la interfaz o contrato de un servicio atendiendo a diversos estándares.
* Autogeneración de parte del código de la implementación de un servicio.
* Autogeneración del código para consumir un servicio a partir de su interfaz.
* Autogeneración de la documentación legible asociada a un servicio.
* Autogeneración de un *front*web sobre el que podemos lanzar ejecuciones de servicios.
* Facilitación de interacción con servicios al atender a diversos protocolos.
* Emulación del comportamiento de servicios para disponer de entornos de pruebas o permitir el desarrollo de integraciones cuando los servicios de terceros no están disponibles en tiempo de desarrollo.
* Facilitación del testeo de servicios distribuidos al producir llamadas a servicios, analizar las respuestas o generar tráfico de alta demanda, entre otros.
* Simplificación la monitorización de una infraestructura basada en servicios.
* Facilitación el despliegue de servicios y microservicios.

7.4. Modelado y descripción de servicios

Un paso fundamental a la hora de afrontar el diseño de una arquitectura basada en servicios es la descripción de la interfaz, contrato o API que un servicio ofrece. Esta tarea puede hacerse desde un inicio, siguiendo una filosofía API-First o bien a medida que vamos avanzando en el desarrollo. En todo caso, si queremos que otros sistemas interactúen con nuestros servicios o debemos consumir servicios de terceros, será preciso disponer de un modelo que contenga la descripción de estos. Las principales tecnologías para ello son:

* API Blueprint (API Blueprint, s. f. a).
* OpenAPI Specification, OAS (OAS, 2021).
* RESTful API Modeling Language, RAML (GitHub, 2021a).
* Web Application Description Language, WADL (Hadley, 2009).
* Web Services Definition Language, WSDL (Christensen et al., 2001).

API Blueprint

Es un lenguaje de alto nivel para describir API web. La sintaxis es una combinación de la sintaxis Markdown—una sintaxis de formato de texto ligera— con Markdown Syntax for Object Notation(MSON),que es una extensión para describir objetos de datos. API Blueprint es simple y accesible para todos los involucrados en el ciclo de vida de la API. Su sintaxis es concisa pero expresiva. Algunas de sus características son:

* Está diseñado para fomentar el diálogo y la colaboración entre las partes interesadas del proyecto, los desarrolladores y los clientes en cualquier punto del ciclo de vida de la API.
* Es completamente de código abierto bajo la licencia MIT.
* Está diseñado para fomentar mejores diseños de API a través de la abstracción. El objetivo de API Blueprint es desacoplar elementos de API para habilitar la modularidad mientras encapsula el comportamiento de la implementación del *back-end*.
* Se basa en la filosofía de diseño primero. Conceptualmente parejo al desarrollo dirigido por las pruebas, API Blueprint representa un contrato para una API. Discutir su API y acordar el contrato antes de que se desarrolle tiende a conducir a mejores diseños de API.

API Blueprint cuenta con un conjunto de herramientas que facilitan el desarrollo de servicios especificados con esta tecnología (API Blueprint, s. f. b), como editores, *parsers*, *testers*, convertidores o analizadores léxicos entre otros.

Seguidamente, se muestra un ejemplo real de una descripción de un servicio utilizando API Blueprint (API Blueprint, s. f. c):

![A screenshot of a computer program

Description automatically generated]()

![A computer screen shot of a code

Description automatically generated]()

![A screenshot of a computer code

Description automatically generated]()

![A computer screen shot of a computer code

Description automatically generated]()

![A screenshot of a computer code

Description automatically generated]()

![A screenshot of a computer code

Description automatically generated]()

![A white screen with black text

Description automatically generated]()

![A screenshot of a computer

Description automatically generated]()

![A close-up of a computer code

Description automatically generated]()

OpenAPI (OAS)

La especificación OpenAPI (OAS) define una descripción de interfaz estándar e independiente del lenguaje de programación para las API HTTP, que permite, tanto a humanos como a computadoras, descubrir y comprender las capacidades de un servicio sin necesidad de acceder al código fuente, documentación adicional o inspección del tráfico de red. Cuando se define correctamente a través de OpenAPI, un consumidor puede comprender e interactuar con el servicio remoto con una cantidad mínima de lógica de implementación. Similar a lo que han hecho las descripciones de interfaz para la programación de nivel inferior, la especificación OpenAPI elimina las conjeturas al llamar a un servicio.

La especificación OpenAPI también proporciona:

* Un formato no propietario.
* El ecosistema de herramientas más desarrollado: OpenAPI ofrece una gran cantidad de herramientas para trabajar con él (OpenAPI Tools, s. f.).
* Un formato legible tanto por máquinas como por humanos: aunque escribir documentos OpenAPI a mano no es la forma más conveniente de hacerlo, son archivos de texto sin formato que se pueden navegar fácilmente en caso de que sea necesario depurar algo.

Seguidamente se muestra un ejemplo real de una descripción de un servicio utilizando OpenAPI (Swagger, s. f. a.):

![A screenshot of a computer code

Description automatically generated]()

Creada en 2010 y conocida hasta del 2016 como Swagger, existen herramientas que a partir de esta especificación facilitan la generación de documentación, la generación de código para generar servidores y clientes en varios lenguajes de programación, así como la automatización de las pruebas entre otros casos de uso. Describir un servicio formalmente en un formato legible por máquina permite que las herramientas automatizadas la procesen, lo que facilita:

* La validación de la interfaz: verificar que el archivo de descripción sea sintácticamente correcto y se adhiera a una versión específica de la especificación.
* La validación de datos: comprobar que los datos que fluyen a través de la API (en ambas direcciones) son correctos durante el desarrollo y una vez implementados.
* Generación dedocumentación: crear documentación tradicional legible por humanos basada en la descripción legible por máquina, que siempre se mantiene actualizada.
* Generación de código: crear código de servidor y de cliente en cualquier lenguaje de programación, lo que libera a los desarrolladores de tener que realizar la validación de datos o escribir código SDK kit de desarrollo de software), por ejemplo.
* Editores gráficos: permiten la creación sencilla de archivos de descripción utilizando una interfaz de usuario en lugar de escribirlos a mano.
* Servidores simulados: crear servidores falsos que brinden respuestas de ejemplo que usted y sus clientes pueden comenzar a probar antes de escribir una sola línea de código.
* Análisis de seguridad: descubrir posibles vulnerabilidades en la etapa de diseño de la API en lugar de mucho, mucho más tarde.

RESTful API Modeling Language (RAML)

Significa lenguaje de modelado de API RESTful. Es una forma de describir las API prácticamente RESTful de una manera que es altamente legible, tanto para humanos como para computadoras. Decimos prácticamente RESTful porque, hoy en día, en el mundo real, muy pocas API obedecen realmente a todas las restricciones de REST. RAML no es estricto: por ahora, se enfoca en describir claramente los recursos, métodos, parámetros, respuestas, tipos de medios y otras construcciones HTTP que forman la base de las API modernas que obedecen muchas, aunque quizás no todas, las restricciones RESTful.

Con RAML podemos diseñar un API visualmente, probarla y obtener comentarios de los usuarios sin tener que escribir una sola línea de código. Algunas de sus características y beneficios son:

* Diseño visual: permite describir un API en un formato legible por humanos, en otras palabras, texto sin formato. Con herramientas como API Workbench o API Designer, podemos obtener una representación visual de la API a medida que lo diseñamos.
* Reutilización de código: permite incorporar dos mejores prácticas para desarrolladores, es decir, patrones de diseño y reutilización de código. En lugar de tener que confiar en la verificación humana para garantizar que la API sea consistente, RAML permite aprovechar las bibliotecas, las superposiciones, las características y los tipos de recursos para garantizar que al API permanece consistente en todo momento.
* Prototipado de la API REST:podemos crear fácilmente un prototipo de la API y enviarla a posibles usuarios.
* Pruebas tempranas: facilita que estos desarrolladores prueben la API, con integración incorporada en herramientas populares como Postman. Incluso podemos guiar a los usuarios a través de la API simulada utilizando API Notebook*.*

Seguidamente, se muestra un ejemplo real de una descripción de un servicio utilizando RAML (RAML, s. f.):

![A screenshot of a computer code

Description automatically generated]()

![A screenshot of a computer code

Description automatically generated]()

Web Application Description Language(WADL)

Es el lenguaje de descripción de aplicaciones web y una descripción XML legible por máquina de los servicios web basados en HTTP. WADL modela los recursos proporcionados por un servicio y las relaciones entre ellos y pretende simplificar la reutilización de los servicios web que se basan en la arquitectura HTTP existente de la web. Es una especificación independiente de la plataforma y el idioma y tiene como objetivo promover la reutilización de aplicaciones más allá del uso básico en un navegador web.

Seguidamente, se muestra un ejemplo real de una descripción de un servicio utilizando WADL (Oracle, 2010):

![A computer code with many white text

Description automatically generated with medium confidence]()

![A computer code with text

Description automatically generated with medium confidence]()

Web Services Definition Language (WSDL)

Es un lenguaje de descripción de interfaces basado en XML que se utiliza para describir la funcionalidad que ofrece un servicio web. Proporciona una descripción legible por máquina de cómo se puede llamar al servicio, qué parámetros espera y qué estructuras de datos devuelve. La última versión de WSDL, que se convirtió en recomendación del W3C en 2007, es WSDL 2.0.

Los elementos que se definen dentro de un WSDL son:

* *Service*: conjunto de funciones del sistema que se exponen como interfaz.
* *Endpoint*: dirección o el punto de conexión a un servicio web. Por lo general, se representa mediante una cadena de URL HTTP simple.
* *Binding*: especifica la interfaz y define el estilo de enlace SOAP (RPC/documento) y el transporte (protocolo SOAP). La sección de enlace también define las operaciones.
* *Interface*: define un servicio web, las operaciones que se pueden realizar y los mensajes que se utilizan para realizar la operación.
* *Operation:* define las acciones SOAP y la forma en que se codifica el mensaje, por ejemplo, literal. Una operación es como una llamada de método o función en un lenguaje de programación tradicional.
* *Types:* describe los datos. El lenguaje XML Schema (también conocido como XSD) se usa (en línea o referenciado) para este propósito.

WADL y WSDL sirven ambos para describir servicios web. WSDL especifica aquellos que interaccionan usando SOAP. WADL se aplica para los que están basados en REST.

Seguidamente se muestra un ejemplo real de una descripción de un servicio utilizando WSDL (W3, 2007):

![A computer screen shot of a computer code

Description automatically generated]()

![A computer code with text

Description automatically generated with medium confidence]()

Adicionalmente, existe un conjunto de especificaciones en el ámbito de los servicios web que han ido evolucionando y que cubren aspectos de seguridad, interoperabilidad, mensajería, orquestación o soporte para procesos de negocio. Un resumen preparado por innoQ se expone a continuación en la Tabla 1.

A chart of different colored squares

Description automatically generated

Tabla 1. Web Services Standards. Fuente: InnoQ, 2007.

Accede al documento a través del aula virtual o desde el siguiente enlace: <https://www.innoq.com/soa/ws-standards/poster/innoQ%20WS-Standards%20Poster%202007-02.pdf>

7.5. Documentación e interacción con servicios

Uno de los objetivos principales de realizar una descripción de servicios en base a algunos de los estándares presentados anteriormente, es la capacidad de que otras herramientas puedan procesar esa información de manera automática para facilitar la:

* La documentación.
* La emulación.
* La interacción con los servicios.

Las principales plataformas para documentación, emulación o interacción con servicios son:

A screenshot of a computer program

Description automatically generated

Tabla 2. Plataformas para documentación, emulación o interacción con servicios. Fuente: elaboración propia.

Plataformas para la documentación

En primer lugar, a nivel de documentación, algunas de las plataformas más representativas son:

DapperDox

Es una solución de código abierto para generar documentación a partir de una especificación de API. La herramienta es muy flexible y permite diseñar las páginas de documentación de la forma que deseemos. Proporciona una interfaz de usuario clara y sencilla con dos columnas: en la primera columna, tiene su explorador y en la segunda columna, tiene los detalles del método API. Una desventaja es que no se actualiza con frecuencia y aún no es compatible con OpenAPI v3.

A screenshot of a computer program

Description automatically generated

Figura 2. Documentación con DapperDox. Fuente: DapperDox, s. f.

FlexDoc/XML - WSDLDoc

Es un poderoso generador de documentación WSDL/XSD que crea una documentación HTML o RTF (con diagramas XSD) para cualquier número de archivos WSDL y XSD.

Har2openapi

Genera automáticamente documentación de una interfaz a partir del uso de solicitudes de red capturadas en uno o más archivos HAR.

Lucybot

Permite generar documentación a partir de los principales estándares abiertos como Markdown, OpenAPI, RAML o WADL para crear un sitio web totalmente personalizable. Genera automáticamente documentación para cada operación, parámetro y respuesta. También permite usar Markdown o HTML para crear páginas de documentación personalizadas. También soporta la ejecución manual de los servicios desde el entorno web generado.

A screenshot of a computer

Description automatically generated

Figura 3. Documentación con Lucybot. Fuente: Lucybot, s. f.

RAML to HTML:

Es una herramienta de documentación que genera una única consola de página HTML basada en una definición RAML. Está escrito en Node.js y se puede ejecutar como una línea de comando.

RAML2HTML for PHP

Es una aplicación simple que utiliza múltiples plantillas para permitir crear y personalizar documentos API usando RAML.

Oxygen XML Editor

Genera documentación detallada para los componentes de un documento WSDL en formato HTML. Podemos seleccionar los componentes WSDL para incluir en la salida y el nivel de detalles para presentar para cada uno de ellos.

ReDoc

Es una herramienta de código abierto para generar documentación a partir de definiciones OpenAPI. Muestra la información en tres paneles:

* El panel izquierdo contiene una barra de búsqueda y un menú de navegación.
* El panel central contiene la documentación.
* El panel derecho contiene ejemplos de solicitudes y respuestas (GitHub, 2022).

A screenshot of a computer

Description automatically generated

Figura 4. Documentación con Redoc. Fuente: GitHub, 2022.

RapidDoc

Tiene la mejor interfaz de usuario para la documentación de API. Permite crear documentación desde OpenAPI*,*lo que genera un contenido interactivo y muy visual. Esta herramienta es muy flexible y permite personalizar el tema, la fuente y los colores. También podemos incrustar código HTML externo en la documentación o incrustar los documentos en otro código HTML. Viene con una consola lista para usar, que podemos usar fácilmente para hacer llamadas API y probar su API. Es posible cambiar sus atributos usando JavaScript. Tiene una vista de una sola columna con barras contraíbles y estas contienen la consola con las cargas útiles de JSON.

A screenshot of a computer

Description automatically generated

Figura 5. Documentación con Rapidoc. Fuente: Rapidoc, s. f.

Swagger UI

La interfaz de usuario de Swagger permite que cualquier persona, ya sea su equipo de desarrollo o sus consumidores finales, visualice e interactúe con los recursos de la API sin tener implementada ninguna lógica de implementación. Se genera automáticamente a partir de su especificación OpenAPI, con la documentación visual que facilita la implementación de *back-end* y el consumo del lado del cliente (Swagger, s. f. b.). Soporta no solo una documentación HTML, sino que es, a su vez, un entorno interactivo en el que podemos ejecutar de forma manual los servicios al introducir los valores de entrada y comprobando los resultados.

Swagger es una de las herramientas más utilizadas por los desarrolladores, ya que a partir del código fuente la plataforma se encarga de generar la especificación OpenAPI, la documentación y permite su ejecución manual

A screenshot of a computer

Description automatically generated

Figura 6. Documentación con Swagger UI. Fuente: Swagger, s. f. b.

En este vídeo, *OpenAPI y Swagger*, se hará una introducción a la especificación de servicios con OpenAPI y, en particular, al uso de Swagger como marco de documentación e interacción de servicio REST.

WidderShins

Permite crear documentación de API a partir de definiciones de OpenAPI 3.0. La interfaz de usuario de esta herramienta es muy similar a ReDoc, ya que tiene una página de tres columnas: en la primera columna hay un explorador, en la segunda columna tiene la descripción y en la tercera columna tiene su consola para hacer llamadas a la API. Usa plantillas para generar los resultados y son personalizables.

A screenshot of a computer

Description automatically generated

Figura 7. Documentación con WidderShins. Fuente: GitHub, 2021b.

En segundo lugar, una de las utilidades más comunes es autogenerar una implementación emulada de un servicio (en inglés *mock a service*). Esto es necesario en una variedad de situaciones, como:

* Agilizar el desarrollo en paralelo: En un desarrollo *API-firsto design-first*, debemos diseñar y especificar la interfaz o contrato de nuestros servicios. Antes de implementarlos, podemos generar una versión emulada de los mismos que sea capaz de responder a una serie de peticiones predefinidas. De esa forma, el equipo encargado de desarrollar piezas de código que consumirán estos servicios (típicamente el frontend) puede evolucionar en paralelo sin necesidad de esperar a que la implementación esté disponible.
* Facilitar la integración con sistemas de terceros: en el caso de que el sistema deba interaccionar con servicios de terceros, es posible que no tengamos un entorno de desarrollo o pruebas y que no podamos hacer uso del servicio real en producción. En estos escenarios es preciso emular el comportamiento de estos servicios para disponer de un entorno lo más parecido al real y que podamos verificar el correcto funcionamiento antes de su puesta en producción.
* Disponer de escenarios para pruebas automáticas: en el caso de querer hacer pruebas, generalmente automáticas, sobre un conjunto de escenarios, en ocasiones debemos disponer de un contexto en el que uno de los elementos bajo testeo (*System Under Test, SUT*) requiera de un comportamiento anómalo o erróneo de un subsistema o servicio. En esos casos, la solución pasa por disponer de un entorno en el que alguno de los servicios en lugar de tener una implementación real está emulado para devolver errores o resultados no esperados, con el objetivo de poder comprobar que incluso en esas situaciones no deseadas nuestro sistema es robusto y es capaz de reconducir la situación.

Plataformas para la emulación

A nivel de emulación, algunas de las plataformas más representativas son:

MockServer

Es una herramienta para simular/aplicar API HTTP externas. Para cualquier sistema con el que se integre a través de HTTP o HTTPS, MockServer se puede utilizar como:

* Un simulacro configurado para devolver respuestas específicas para diferentes solicitudes.
* Una grabación de proxy y, opcionalmente, modificar solicitudes y respuestas.
* Tanto un proxy para algunas solicitudes como un simulacro para otras solicitudes al mismo tiempo.

CuandoMockServer recibe una solicitud, compara la solicitud con las expectativas activas que se han configurado. Luego, si no se encuentran coincidencias, transmite la solicitud si corresponde, de lo contrario, se devuelve un 404. De forma general, la herramienta puede:

* Generar y devolver respuestas fijas.
* Reenviar una solicitud a otro servidor.
* Ejecutar devoluciones de llamada.
* Devolver respuestas erróneas.
* Verificar una solicitud.
* Registrar logs para facilitar la monitorización y depuración.

WireMock

Es una herramienta para simular API basadas en HTTP que se ejecuta en sus pruebas unitarias, en su escritorio o en su entorno de prueba. En esencia, es un servidor web que se puede preparar para brindar respuestas a solicitudes particulares (*stubbing*) y que captura las solicitudes entrantes para que puedan verificarse más tarde. También tiene una variedad de otras características útiles que incluyen grabación/reproducción de interacciones con otras API, inyección de fallas y retrasos, simulación de comportamiento con estado. Cualquier aplicación JVM puede utilizarlo como una biblioteca o ejecutarlo como un proceso independiente en el mismo *host* que el sistema bajo prueba o en un servidor remoto. Se puede acceder a todas las funciones de WireMock a través de su interfaz REST (JSON) y su API Java.

Algunas plataformas de emulación para servicios descritos con OpenAPI son:

* API Sprout: un servidor simulado ligero, ultrarrápido y multiplataforma con cliente de validación sobre Docker.
* Apitive Studio: una plataforma para gerentes de productos digitales y consultores de API para diseñar API REST con simulacros y documentación incorporados.
* Connexion: OpenAPI First framework para Python sobre Flask con validación automática de punto final y compatibilidad con OAuth2 Python.
* Fakeit: es posible crear un servidor simulado a partir de la especificación OpenAPI con generación de respuestas aleatorias y validación de solicitudes.
* Falcon Heavy: el marco para crear *back-ends* de aplicaciones y microservicios a través del flujo de trabajo de diseño de API primero.
* Meeshkan: simula las API de HTTP a través de una combinación de definiciones de API, tráfico registrado y código. Se utiliza para *sandboxes*, así como para pruebas exploratorias y automatizadas.
* Microcks: herramienta nativa de Kubernetes para simulación y pruebas de API. Se pueden convertir los ejemplos de contratos OpenAPI en simulacros.
* Mockintosh: simulacros para entornos nativos de la nube: convierte archivos de OpenAPI en emulaciones y los usa para desarrollar en entornos aislados y probar casos extremos, llamadas asincrónicas a colas como Kafka o RabbitMQ o simulación de pruebas de rendimiento y caos.
* MockLab: plataforma SaaS para cargar su especificación para crear un servidor simulado SaaS
* OpenAPI Mocker: servidor simulado OpenAPI basado en Node.js independiente, compatible con Docker con validación de solicitudes y recarga automática.
* openapi-data-mocker: pequeña biblioteca para generar tipos de datos básicos de OpenAPI. Es como un paquete Faker extendido. Puede ser útil para escribir pruebas unitarias personalizadas en PHP.
* Prisma: permite convertir cualquier archivo OpenAPI en un servidor API con simulación, transformaciones y validaciones.
* Sandbox: herramienta SaaS, autohospedada o de consola para convertir las descripciones de OpenAPI en un servidor simulado, donde podemos modificar el comportamiento y simular el tiempo de inactividad gracias a un editor de código integrado. SaaS/Java
* Unmock: biblioteca de prueba de integración de API que intercepta solicitudes salientes y devuelve datos simulados en función de las descripciones de OpenAPI sobre Node.js.
* yii2-app-api: genera código API del lado del servidor con enrutamiento, modelos, validación de datos y esquema de base de datos a partir de una descripción de OpenAPI basado en el marco yii.

Plataformas de emulación con API Blueprint o RAML

Algunas plataformas de emulación para servicios descritos con API Blueprint o RAML son:

* Apish: facilita la creación de servicios y pruebas con JavaScript desde API Blueprint.
* API Blueprint Mock Server: facilita la ejecución de un servidor simulado simple y rápido en Java desde un API Blueprint.
* API-Mock: permite generar un servidor simulado simple y rápido a partir de un API Blueprint.
* ASP.NET Blueprint Mock Server: permite crear un servidor simulado con tecnologías ASP.NET desde un API Blueprint.
* Drakov: servidor simulado que implementa la especificación API Blueprint, con algunos extras incorporados. Se puede utilizar como servidor independiente y como módulo Node.js.
* Osprey Mock Service: genera un servicio simulado de API a partir de una definición RAML utilizando Osprey de MuleSoft, un generador de *middleware* API a partir de una definición de RAML, que se puede usar local o globalmente para validar solicitudes y respuestas de API.
* RAML Mock Service: otra implementación para generar un servicio simulado de API a partir de una definición RAML usando Osprey.

En tercer lugar, tenemos herramientas que facilitan la interacción con los servicios al generar llamadas con REST o SOAP y comprobando los resultados. A diferencia de las anteriores, en las que el objetivo era documentar y permitían una cierta interacción manual, estas plataformas permiten automatizar no solo una llamada, sino un flujo de llamadas, lo que sistematiza la inserción de datos en el *header* y es posible añadir lógica para en función de la respuesta realizar los siguientes pasos, por lo que son válidas para automatizar flujos de testeo con un mayor nivel de complejidad.

Plataformas para la invocación

Algunas de las principales herramientas de invocación de servicios son:

Postman

Es una plataforma API para construir y usar API. Puede ejecutarse en múltiples sistemas operativos desde Windows y macOS hasta Linux. Les ofrece a los desarrolladores más espacio para desarrollar API. Para los probadores funcionales, permite crear pruebas para llamadas a la API sin gastar mucho esfuerzo en codificar *scripts*. Sus principales beneficios son:

* Facilidad de uso.
* Accesibilidad.
* Funcionalidades avanzadas, como ser admitir todos los métodos HTTP, guardar el progreso, convertir de API a código, etc.
* Soporta el análisis de las respuestas para condicionar los siguientes pasos del flujo.

Sus principales desventajas son:

* Aunque es ideal para pruebas de API RESTful, no está bien diseñado para API SOAP y otras API.
* Los usuarios no pueden reutilizar sus guiones prescritos ni agregar más solicitudes, por lo que los equipos de QA (análisis de calidad) tienen que crear nuevos scripts de prueba una y otra vez para cada proyecto.
* La herramienta en sí no admite muchas capacidades de integración por lo que se convierte en un obstáculo para conectar Postman con los sistemas existentes y colaborar dentro del equipo.

A screenshot of a computer

Description automatically generated

Figura 8. Pantalla de uso de PostMan. Fuente: Postman, s. f.

SoapUI

Es una herramienta para probar servicios web, tanto usando SOAP como HTTP o los servicios web RESTful. Es una solución de código abierto y completamente gratuita, cuenta con una versión comercial (ReadyAPI) que aporta funcionalidad adicional para empresas con servicios web de misión crítica. Esta herramienta se considera el estándar de facto para las pruebas de servicio de API por el número de descargas y uso en la comunidad, por lo que hay mucho conocimiento sobre la herramienta, lea los blogs en la red para obtener más información sobre el uso de SoapUI en la vida real. Con la herramienta podemos realizar pruebas funcionales, de rendimiento, de interoperabilidad y de regresión. Podemos emular servicios web, grabar pruebas y usarlas más tarde, crear apéndices de código desde el *WSDL* e incluso crear especificaciones REST (WADL) a partir de comunicaciones grabadas.

A screenshot of a computer

Description automatically generated

Figura 9. Pantalla de uso de SoapUI. Fuente: SoapUI, s. f.

7.6. Plataformas aceleradoras del desarrollo de servicios

A la hora de desarrollar sistemas distribuidos basados en servicios, existen una serie de plataformas que nos facilitan y aceleran el desarrollo. Además de las que hemos visto que ayudan a la definición, emulación o interacción, o las específicas de testeo, en esta sección expondremos algunas de las principales a la hora de simplificar la construcción, tanto de implementaciones de servicios como de consumidores de servicios.

Las principales plataformas que aceleran la creación o consumo de servicios son:

A blue squares with text

Description automatically generated with medium confidence

Tabla 3. Plataformas aceleradoras del desarrollo de servicios. Fuente: elaboración propia.

Plataformas aceleradoras para la implementación de servicios

A nivel de implementación de servicios, algunas de las plataformas aceleradoras más representativas son:

Apache CXF

Su funcionalidad wsdl2java toma un documento WSDL y genera un código Java completamente anotado a partir del cual implementar un servicio.

Eclipse Jersey

Es un marco REST que proporciona una implementación JAX-RS (JSR-370, la API de Java para servicios web RESTful). Permite realizar un seguimiento de la API de JAX-RS y proporciona versiones periódicas de implementaciones de referencia de calidad de producción que se envían con GlassFish. Facilita la creación de API para extender Jersey así como el desarrollo de servicios web RESTful utilizando Java.

OpenAPI Generator

A partir de una especificación OpenAPI, es posible generar un esqueleto con la implementación en múltiples lenguajes de programación como Ada, ASP.NET, C++, F#.NET, Go, Haskell, Java, PHP, Python, Ruby y Scala.

RAML Tools para .NET

Es una extensión de Visual Studio que permite a los desarrolladores generar una nueva implementación de API web ASP.NET desde cero utilizando un enfoque de contrato primero a partir de una RAML u OAS creada previamente.

Spring WebFlux

Proporciona soporte de programación reactivo, asíncrono y sin bloqueo para aplicaciones web en un formato de controlador anotado similar a SpringMVC. Este enfoque es similar a cómo Node.js usa un modelo asíncrono sin bloqueo que ayuda a que sea más escalable. Spring WebFlux usa un modelo similar pero con múltiples bucles de eventos. Se aleja del modelo de bloqueo de subprocesos por solicitud en SpringMVC tradicional (con Tomcat de forma predeterminada) y avanza hacia un paradigma multi-EventLoop, asíncrono, sin bloqueo (con Netty de forma predeterminada) que es más escalable y eficiente que el código de bloqueo tradicional.

Swagger Codegen

Puede simplificar el proceso de compilación al generar *stubs* de servidor para cualquier API definida con la especificación OpenAPI.

Plataformas aceleradoras para la generación de clientes de servicios

A nivel de generación de clientes de servicios, algunas de las plataformas aceleradoras más representativas son:

EasyWSDL

Genera el código para consumir cualquier servicio web SOAP especificado con WSDL, en varios lenguajes como Java para Android, Dart/Flutter Java o clases deObjective-C para iOS.

OpenAPI Generator

A partir de una especificación OpenAPI, es posible generar una librería cliente con la implementación del consumo de los servicios en múltiples lenguajes de programación como Ada, Android, C, Clojure, C++, C#.NET, Go, Java, JavaScript, Kotlin, Perl, PHP, Python, R, Ruby, Scala y TypeScript.

RAML Client Generator

Es un generador de clientes de API descritos con RAML sobre Node.js.

RAML Tools para .NET

Es una extensión de Visual Studio que permite a los desarrolladores integrar y consumir fácilmente API que exponen una definición RAML u OAS.

Swagger Codegen

Genera *software development kits*(SDK) de cliente de una API definida con la especificación OpenAPI.