Universidad de Santiago de Compostela

Ingeniería de Servicios

Autor:
Brais Míguez Varela
14 de enero de 2024

Capítulo 1

Introducción

1.1. Concepto de servicio

Un servicio consiste en un programa que hace su funcionalidad disponible a través de una interfaz predefinida. Un método invoca la ejecución de otros métodos para obtener algo que no puede alcanzar por sí mismo. Una clase utiliza los métodos que ofrece otra clase para completar su funcionalidad.

Los retos son:

↪ **¿Qué ocurre si las dos clases no están en la misma máquina?** Se utiliza la informática distribuida.

↪ **¿Dónde están las clases?** Se debe utilizar un procedimiento de descubrimiento.

↪ **¿De qué modo se envía información entre clases?** Se debe utilizar un protocolo de comunicación común.

↪ **¿Qué sucede si no se encuentra ninguna clase con la funcionalidad necesaria?** Se debe usar la composición o combinación de componentes.

Las posibles soluciones a estes problemas son:

↪ Soluciones basadas en **llamadas a procedimientos remotos (RPC)**: pueden ser DCOM (Distributed Component Object Model), CORBA (Common Object Request Broker Architecture) o RMI (Remote Method Invocation).

↪ Soluciones basadas en web con una capa software adicional: pueden ser Servlets o **Servicios Web**.

Los problemas de RPC son que normalmente se utilizan tecnologías no interoperables y que es una arquitectura independiente, lo que puede llevar a problemas de estandarización.

Los problemas asociados con Servlets son: el intercambio de información se hace mediante cadenas de texto (puede dificultar la integración de nuevos componentes y no se manejan tipos de datos) y los métodos predefinidos GET y POST obligan a que la ejecución de otros métodos requiera del uso de parámetros (poca flexibilidad que dificulta la integración y la reutilización de funcionalidades).

1.2. Definición de servicio

Ofrece una serie de funciones relacionadas entre sí. Estas funcionalidades están claramente definidas (API). El consumidor invoca el servicio siguiendo esa API. Un servicio puede invocar uno o varios servicios.

La composición de servicios consiste en varios servicios que pueden colaborar para crear un nuevo servicio.

1.3. Arquitectura Orientada a Servicios (SOA)

Consiste en descomponer un problema en partes más simples, en la que cada una resuelve un único problema. Todas las partes son agnósticas del problema completo o de las demás partes (favoreciendo la reutilización). Cada parte se implementa como un servicio en una arquitectura SOA.

Los componentes están bajamente acoplados para dar un alto nivel de independencia entre los servicios. Hay una abstracción a nivel de servicio, por lo que la lógica interna del servicio es transparente. Para descubrir los servicios, estos deben proporcionar información a cerca de su invocación. El término SOA es agnóstico de la tecnología.

Cada servicio puede ejecutarse en una máquina independiente, por lo que se debe definir un protocolo de comunicación.

1.4. SOA vs. Aplicación monolítica

↪ Reutilización de código.

↪ Facilidad para reutilizar infraestructura "legacy".

↪ Desarrollo más eficiente y ágil.

↪ Es fácil de mantener.

↪ Más escalable.

↪ Facilidad de uso por terceros.

1.5. Elementos de una SOA

↪ **Servicio**: componentes software bajamente acoplados (*loosely coupled*). Componentes independientes. Pueden combinarse entre sí para ofrecer nuevas funcionalidades.

↪ **Proveedor**: ofrece una serie de servicios con una funcionalidad dada. Los servicios deben ser accesibles. Proporciona una descripción de los servicios disponibles.

↪ **Consumidor**: invoca o consume las funcionalidades proporcionadas por los servicios. Utiliza un protocolo de invocación prefijado.

↪ **Registro**: Contiene los servicios proporcionados por el proveedor. No siempre está presente

1.6. Servicios Web

En los servicios web, hay un registro UDDI que almacena características no funcionales de los servicios. El registro contiene la URL del fichero WSDL que contiene características funcionales del servicio correspondiente.

1.6.1. WSDL: Web Services Description Language

Proporciona una descripción XML del servicio que permite generar los tipos de datos que se utilizan como parámetros de entrada/salida. También aporta información de como invocar el servicio.

1.6.2. SOAP: Simple Object Access Protocol

Es un protocolo de ejecución de operaciones descritas en WSDL. Los mensajes SOAP están basados en XML. Permite integrar implementaciones realizadas en lenguajes de programación diferentes y permite el uso de modelos de datos diferentes, porque la comunicación se produce mediante representaciones intermedias en XML.

Las desventajas de SOAP son: los mensajes consumen un gran ancho de banda, las operaciones de serializar/deserializar XML son costosas, no se especifica formato para el cuerpo del mensaje y rendimiento pobre al ser peticiones no “cacheables”.

Capítulo 2

REST (REpresentational State Transfer)

Estilo arquitectónico para construir sistemas distribuidos bajamente acoplados. Los principios REST son:

1. **Cliente-servidor**: uso de sistemas desacoplados. Las funciones deben estar claramente separadas entre cliente y servidor. Las interacciones se hacen a través de una interfaz uniforme.
2. **Interfaz uniforme**: define la interfaz de comunicación entre cliente y servidor. Se deben utilizar identificadores únicos (URI) para accedes a los recursos. Se puede acceder a recursos de varias formas diferentes.
3. **Iteraciones sin estado**: el servidor no mantiene un estado asociado al cliente. Cada petición debe contener toda la información de contexto. Los estados asociados a la sesión se manejan en el lado del cliente.
4. **Interacciones "cacheables"**: las respuestas del servidor pueden almacenarse en una caché de varias formas:
   - ↪ De forma **implícita**: cuando el cliente puede almacenar cualquier respuesta en una caché.
   - ↪ De forma **explícita**: el servidor especifica los parámetros de la caché.
   - ↪ **Negociable**: cliente y servidor negocian los parámetros de la caché.
5. **Sistema basado en capas**: el cliente no puede asumir una conexión directa con un servidor porque puede haber elementos software o hardware intermedios.
6. **Código bajo demanda (opcional)**: el servidor puede extender el cliente. Se transfiere lógica al cliente y este ejecuta el código generado por el servidor.

Estos principios aseguran una serie de características no funcionales deseables en la mayor parte de sistemas distribuidos:

- ↪ **Escalabilidad**: cantidad de componentes con nuevas funcionalidades. Número de acciones entre componentes.
- ↪ **Modificabilidad**: habilidad para realizar cambios en la arquitectura.

↪ **Extensibilidad**: es la habilidad para incorporar al sistema nuevas funcionalidades y componentes.

↪ **Capacidad de evolución**: indica en qué medida se puede cambiar la implementación de un componente sin afectar a los otros.

↪ **Reusabilidad**: es la habilidad de que los componentes y conectores de una aplicación puedan ser usados en otra sin realizar ningún cambio.

↪ **Capacidad de configuración**: es la habilidad para cambiar la conducta de un componente.

↪ **Visibilidad**: se refiere a la habilidad de un componente para monitorizar o mediar en la interacción entre ostros componentes.

↪ **Portabilidad**: se refiere a la habilidad de un componente de operar en diferentes entornos de ejecución.

↪ **Fiabilidad**: se entiende como el grado en el que una arquitectura puede seguir funcionando cuando falla alguno de sus componente.

↪ **Simplicidad**: separar funciones entre los diferentes componentes.

2.1. Tecnología Web

Se pueden desarrollar sistemas basados en el estilo REST usando cualquier tecnología, siempre que se mantengan los principios REST. La web sigue los principios REST: en la identificación de recursos se utilizan URIs, los métodos de HTTP componen una interfaz uniforme y el protocolo HTTP facilita la caché de la interacciones entre cliente y servidor.

En REST, todo es un recurso y estos deben tener un identificador único, en el caso de web, URIs. Las buenas prácticas en URIs son:

↪ Usar nombres en vez de verbos.

↪ Mantener URIs cortas.

↪ seguir un esquema posicional de pase de parámetros en vez de una codificación estilo $clave = valor \& p = v$.

↪ Se pueden utilizar plantillas para gestionar las URIs desde el cliente y desde el servidor.

Una alternativa a las URIs es utilizar la cabecera HTTP para indicar el tipo haciendo uso de los campos *Content-Type* y *Accept*.

La forma de acceder a los recursos en una aplicación REST es siempre la misma: solamente se puede hacer uso de los métodos especificados en el protocolo HTTP. Los principales métodos HTTP son:

↪ **POST**: crea un recurso subordinado.

↪ **GET**: recupera el estado actual de un recurso.

↪ **PUT**: inicializa o actualiza el estado de un recurso. Crea un recurso cuando se conoce su URI.

↪ **DELETE**: borra el recurso, de modo que la URI deja de ser válida.

Los métodos GET, PUT y DELETE son idempotentes:

↪ **PUT**: en la primera ejecución se crea el recurso, en las siguientes se actualiza. Si se llama con los mismos datos el resultado siempre es el mismo.

↪ **DELETE**: en la primera ejecución se elimina en la siguientes se devuelve un 404, pero el resultado final es el mismo.

↪ **GET**: es la única operación segura.

Las diferencias entre PUT y POST:

↪ PUT debe utilizarse para actualizar un recurso, indicando en la petición el identificador del recurso.

↪ PUT puede usarse para la creación de un recurso cuando el cliente conoce el identificador del propio recurso.

↪ El problema de PUT se da cuando varios clientes puedn intentar crear el mismo recurso.

↪ POST debe usarse para la creación de un recurso cuando el identificador se genera en el servidor.

↪ El problema de POST es que es posible que se dupliquen recursos.

Los principales códigos de estado de HTTP son 200 (OK), 204 (No Content), 404 (Not Found), 403 (Forbidden), 500 (Internal Server Error) y 405 (Method Not Allowed).

Los métodos de HTTP no tiene estado. Se utilizan cookies para mantener el estado. Están asociadas a cada navegador. Las cookies de sesión requieren de un seguimiento costoso por parte del servidor. Una alternativas es tratar el estado como un recurso.

2.2. HATEOAS (Hypermedia As The Engine Of Application State)

La interacción del cliente con la aplicación tiene lugar a través de los enlaces proporcionados de forma dinámica por el servidor. La interacción entre cliente y servidor se establece mediante enlaces a los recursos. Si un recurso está enlazado con otro recurso, la información del recurso que se transfiere entre el cliente y el servidor contiene los enlaces a dichos recursos. Las ventajas son: el cliente no necesita ningún conocimiento previo sobre la forma de interactuar con el servidor y que desacopla el cliente y el servidor.

Capítulo 3

Diseño de APIs

Los métodos estándar tienen objetivos claramente definidos y son predecibles, interpretables y con efectos secundarios limitados. Los métodos no estándar son contrarios a los principios REST. En ningún caso se debe implementar funcionalidad ya recogida en el conjunto de métodos estándar.

3.1. LROs

Las operaciones prolongadas (LRO, long running operations) son operaciones que no devuelven el resultado de forma inmediata, que necesitan mucho tiempo e ejecución y que no es factible esperar a su finalización para devolver un resultado.

El método POST devuelve metadata (con información de la LRO) de recurso que se desea crear. Las LROs se convierten en un recurso sobre el que se pueden realizar consultas. El recurso generado por una LRO puede estar disponible incluso después de que la operación finalice (persistencia).

Lo que se necesita representar es: el id de la operación, si ha terminado o no, el resultado y metadatos de la operación.

Se debe incluir una colección para las LRO dentro del recurso sobre el que se ejecutan. Puede ser difícil especificar un único recurso del que depende una LRO.

La finalización de las LRO puede ser de dos formas:

↪ **Polling**:

- Aproximación más simple.
- Comprobar activamente mediante consultas.
- Control en lado del cliente.
- Puede suponer coste extra en tiempo de ejecución y tráfico de red.

↪ **Waiting**:

- Mantener una conexión activa con el servidor.
- Control en el lado del servidor.
- Se utiliza un método no estándar de “wait”.

Los códigos de respuesta de HTTP están pensados para dar una respuesta inmediata a cada petición. Las operaciones de acceso a las LRO devuelven un código de error si no se puede

acceder al recurso. Si se produce un error durante la ejecución de la operación, el resultado contendrá el error. Se debe tener un método no estándar para cancelar una LRO (alternativa: usar el método REMOVE de HTTP). Se debe marcar la operación como finalizada y eliminar todos los posibles resultados intermedios generados durante la ejecución. No todas las LRO se pueden parar y reanudar, dependerá del caso concreto.

Se debe ofrecer la posibilidad de listar todas las LRO en ejecución y listar las operaciones ya terminadas. Las LROs son un recurso que permanece almacenado en el sistema como cualquier otro. En algunos casos puede interesar mantener todo el histórico de LROs. Se puede definir una ventana temporal en la que se mantiene el recurso y posteriormente se elimina. Se pueden establecer políticas de borrado más elaboradas que generalmente hacen el diseño del sistema innecesariamente complejo.

3.2. Trabajos reejecutables

Son acciones que una vez creadas se pueden ejecutar varias veces. Se crea un nuevo recurso que representa el trabajo. Se ejecuta la acción contenida por el trabajo creador (potencialmente una LRO). Se maneja la LRO siguiendo el patrón habitual.

3.3. Relaciones avanzadas entre recursos

3.3.1. Subrecursos: Patrón Singleton

¿Cuáles son los motivos para crear un nuevo subrecurso versus atributo de un recurso ya existente?

- ↪ **Tamaño y complejidad**: si un atributo contiene más información y con una estructura más compleja que las demás componentes del recurso.
- ↪ **Seguridad**: diferentes permisos de acceso para un atributo y que para el resto de información contenida en el recurso.
- ↪ **Volatilidad**: un atributo con una mayor frecuencia de actualización que el resto de los datos del recurso al que pertenecen.

El subrecurso Singleton es una representación intermedia entre recurso y atributo.

En los métodos GET y PUT/PATCH, el acceso al subrecurso se comporta como cualquier otro recurso. Se define un nuevo identificador para el subrecurso. El método POST tiene un comportamiento más cercano al de un atributo que a un recurso. No es necesario crear el subrecurso, generalmente se crea de forma implícita al crear el recurso del padre. En el método DELETE, el comportamiento más cercano de un atributo que a un recurso. La eliminación del recurso padre implica la eliminación del subrecurso.

Un subrecurso siempre requiere la presencia de un recurso padre. Las limitaciones de este patrón de diseño son:

- ↪ No se puede acceder de forma atómica a todos los componentes de un recurso, puesto que una parte de la información se almacena en un subrecurso.
- ↪ Cada subrecurso representa un único elemento, no se contempla la posibilidad de manejar subcolecciones de recursos.

3.3.2. Asociación de recursos

En los métodos estándar, la asociación se comporta como un nuevo recurso. La asociación de dos recursos no debe estar duplicada (código de error asociado 409 Conflict). Los metadatos de la relación entre dos recursos pueden cambiar con el tiempo. Los recursos que se relacionan deberían permanecer inmutables. Una relación entre recursos con identificadores diferentes debería suponer la eliminación de la relación actual y la creación de una nueva.

La alternativa a saber qué grupo pertenece un usuario y qué usuarios pertenecen a un grupo es crear métodos no estándar que implementen un alias para cada tipo de búsqueda.

Al trabajar con referencias a otros recursos se debe tener en cuenta si estas referencias siguen siendo válidas (integridad referencial). Cascada y poner a null pueden ocasionar un gran número de escrituras en la base da datos.

3.3.3. Relaciones: métodos no estándar

Es una aproximación más simple para implementar relaciones "n o n". En lugar de utilizar un recurso adicional se crean los métodos no estándar `add` y `remove`. Las restricciones son: que no se da soporte a metadatos de la relación y los métodos no estándar pertenecen a uno de los recursos creando una relación no simétrica.

Una vez se decide el recurso que implementará los métodos no estándar se procede a su implementación. Estos métodos modificarán la lista de usuarios que pertenecen a un grupo. Para listar los n recursos asociados a uno dado, se implementan dos métodos, uno asociado a cada recurso. Cada uno de ellos lista los elementos del otro recurso en la relación.

Se deben tener las en cuenta las mismas consideraciones acerca de la integridad de los datos. No se pueden eliminar recursos que no existen de una relación (código de error 421 Precondition Failed).

3.3.4. Polimorfismo

El poliformismo en POO permite un diseño más modular y mantenible. El polimorfismo de recursos en APIs permite tener recursos que implementan interfaces genéricos. En muchos casos tendrá sentido aplicar polimorfismo en arquitecturas orientadas a objetos, pero no en el diseño de APIs.

La implementación consiste en añadir un atributo que especifique el subtipo del recurso. Se pueden filtrar los recursos utilizando este atributo. En algunos casos es deseable permitir el cambio en el tipo del recurso, pero no es recomendable permitir las modificaciones. El contenido del recurso debe añadir atributos que den soportes a todos los posibles contenidos de los subtipos. El recurso actúa como un superconjunto de todos los posibles subtipos.

Los diferentes tipos de un recurso con polimorfismo deben almacenar información diferente. Los criterios de validación de datos deben ser diferentes para cada subtipo. Diferentes subtipos pueden compartir atributos con diferentes requisitos de validación. Se puede acceder a todos los subtipos a través del recurso que actúa como interfaz. A través del atributo tipo se puede identificar el subtipo. El servidor puede no devolver los campos no inicializados filtrando los atributos no relacionados con el subtipo actual.

El polimorfismo a nivel de método consiste en métodos capaces de operar con múltiples tipos de recursos (no recomendado). Los recursos modelan entidades atómicas sobre las que se definen una serie de métodos independientes para cada recurso.

Cambios en los recursos o en la implementación del método común puede tener efectos secundarios en múltiples componentes del sistemas. En el diseño de arquitecturas orientadas a servicios

se evitará este tipo de elementos, contrarios a la filosofía de implementar componentes bajamente acoplados.

3.4. Control de versiones y compatibilidad

Los cambios que afectan a la interfaz de comunicación con el cliente podrán ser transparentes. Los puntos críticos están en los cambios que afectan a la propia definición de esta interfaz. La solución es mantener diferentes versiones disponibles de forma simultánea.

Dos versiones de una API son compatibles si se pueden intercambiar sin afectar al funcionamiento de los clientes. Los tipos de cambios pueden ser:

↪ **Dar soporte a nuevas funcionalidades**:

- Añadir atributos a los recursos existentes.
- Añadir nuevos recursos.
- Esto determinará si las versiones son compatibles o no.

↪ Corrección de errores:

- Los cambios no suponen modificar el comportamiento esperado por el cliente.
- Los cambios pueden tener impacto en lo que recibe el cliente.

↪ Cambios obligatorios:

- Debido a cambios regulatorios.
- En algunos casos se puede definir un periodo de transición en el que las dos versiones pueden coexistir.
- La creación de una nueva versión o actualización de la ya existente dependerá de las implicaciones legales.

↪ Cambios internos:

- Optimización de los tiempos de ejecución, actualización de modelos etc.
- En general no causan incompatibilidades, pero dependerá de cada caso.

3.4.1. Estabilidad permanente

Pocos cambios que generan incompatibilidades y que introducen en una nueva versión. Se sigue dando soporte a la antigua. Cada cambio sobre la versión N, genera una versión N+1. Las versiones antiguas se pueden abandonar con el tiempo para reducir costes de mantenimiento. Estrategia adecuada para cuando la mayor parte de los cambios no generan incompatibilidades. No es una estrategia adecuada para casos de uso en los que los usuarios requieran un algo grado de granularidad.

3.4.2. Agile Instability

Se establece un mecanismo de tipo ventana deslizante para gestionar la obsolescencia de versiones antiguas. Se establece un ciclo de vida claramente definido para cada nueva versión. Cuando la versión actual pasa al estado “current” los nuevos cambios o obligatorios originarán la creación de una nueva versión.

Se mantienen varias versiones de forma simultánea pero solo una debe estar en el estado "current" en cada instante. Se establece un calendario claro de cara a los usuarios. Generalmente se buscan ciclos de vida cortos que permitan un desarrollo continuo de nueva funcionalidad.

Permite una rápida inclusión de una nueva funcionalidad de forma ágil permitiendo la creación de nuevas versiones que pasan rápido a obsoletas. La aceptación por parte de los usuarios puede ser un problema.

3.4.3. Versionado semántico

La versión se indica con una cadena de 3 números en la que cada una tiene un significado concreto:

- ↪ *Major version*: se incrementa cuando las actualizaciones rompen la compatibilidad hacia atrás.
- ↪ *Minor version*: se incrementa cuando se hacen cambios manteniendo la compatibilidad hacia atrás.
- ↪ *Patch veersion*: se incrementa cuando se realizan cambios que corrigen errores sin afectar a la compatibilidad hacia atrás.

Tiene un alto nivel de granularidad al identificar todos los cambios con una nueva versión. El alto nivel de granularidad puede afectar a la aceptación por parte de los usuarios. Se deben definir políticas de obsolescencia.