

Diseño de APIs



Presentado por Iván López
www.consultec-ti.com

Módulo 5

Diseño de APIs







Objetivo

El objetivo de este curso es que los estudiantes adquieran los conocimientos necesarios para diseñar APIs REST efectivas y seguras utilizando diferentes tecnologías y plataformas, comprendiendo la importancia de las APIs en la integración de aplicaciones y aplicando los conceptos básicos de REST en el diseño de las mismas.

Además, al finalizar el curso, los estudiantes podrán implementar medidas de autenticación, autorización, gestión de errores y códigos de respuesta HTTP, seguridad en el intercambio de datos, gestión de versiones, documentación y optimización de la performance y escalabilidad de la API, utilizando herramientas como Swagger y otras.

Agenda Día 1



-  Introducción a las APIs y su importancia en la integración de aplicaciones
-  Conceptos básicos de REST y su utilización en el diseño de APIs
-  Diseño de la estructura de la API: rutas y métodos HTTP.
-  Autenticación y autorización en APIs REST
-  Gestión de errores y códigos de respuestas HTTP
-  Utilización de HTTPS y seguridad en el intercambio de datos.

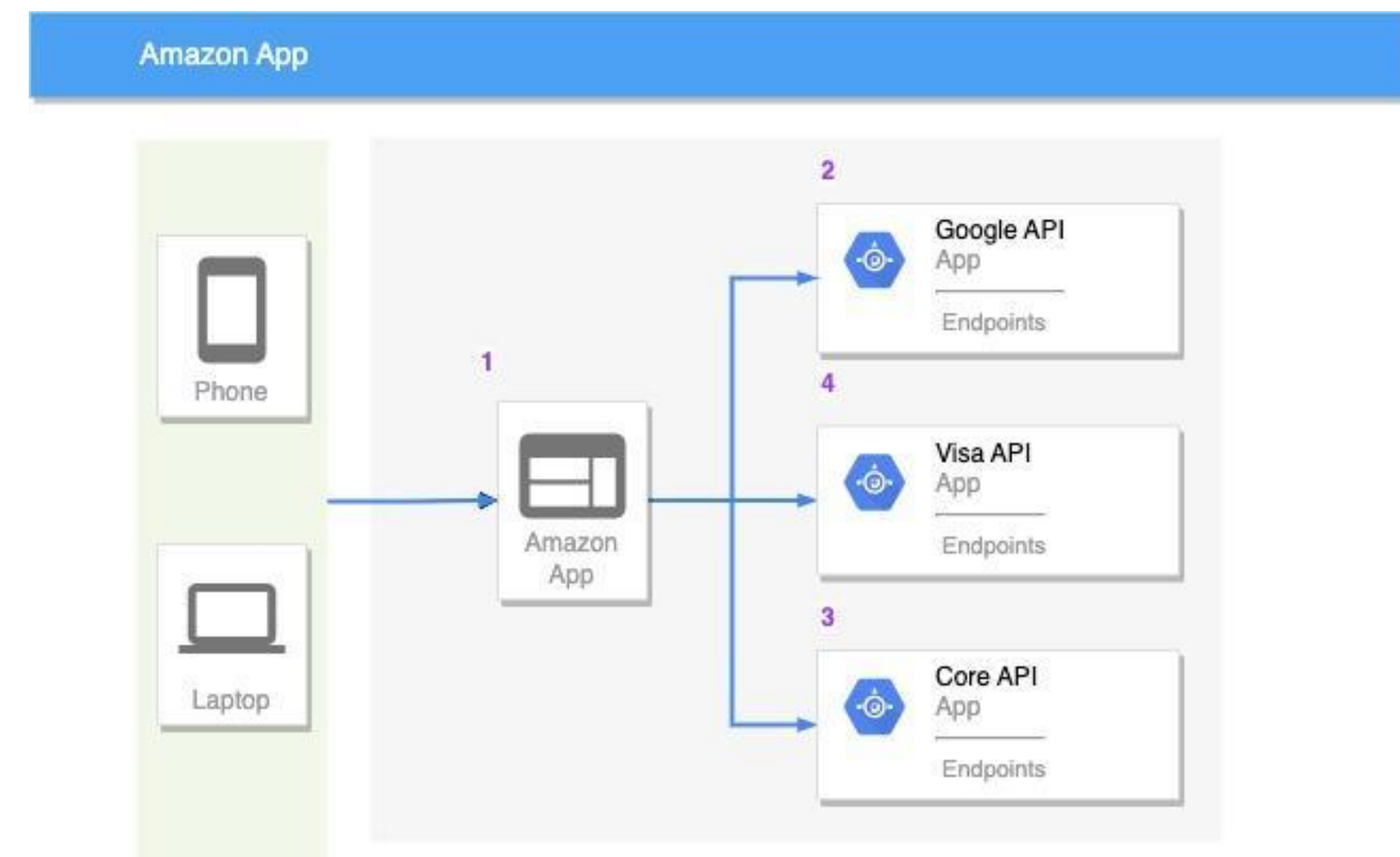


Introducción a las APIs

¿Qué es una **API** y por qué son **importantes**?

Una API (Application Programming Interface) es un conjunto de definiciones y protocolos que se utilizan para la integración de diferentes aplicaciones. En otras palabras, una API es un medio por el cual diferentes aplicaciones pueden comunicarse entre sí.

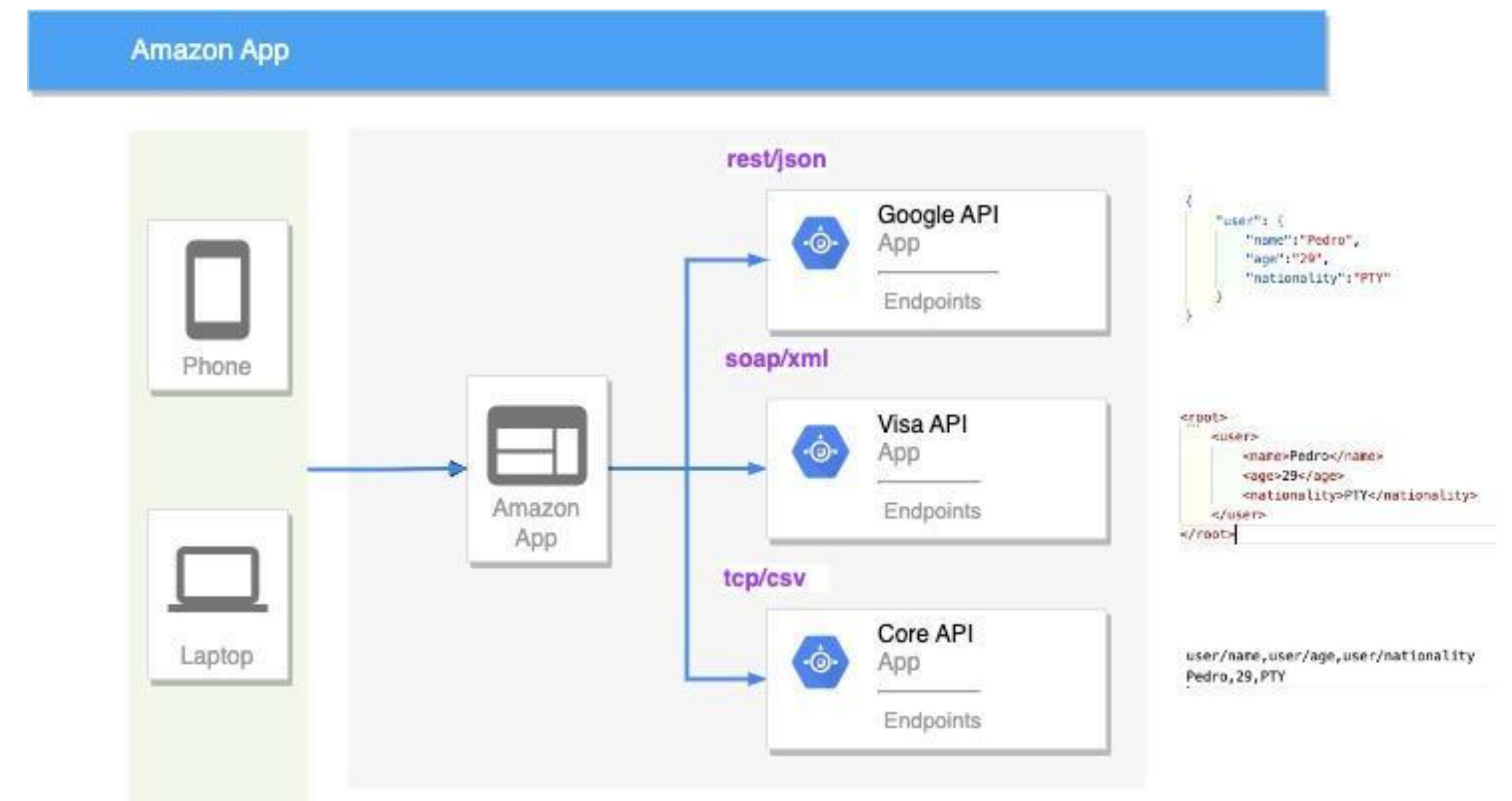
Son importantes porque permiten a los desarrolladores crear aplicaciones que se integran fácilmente con otras aplicaciones o servicios existentes. Esto ahorra tiempo y recursos, ya que los desarrolladores no tienen que crear todas las funcionalidades desde cero. Además, las APIs permiten a las empresas ofrecer servicios más completos y personalizados a sus clientes.



¿Cómo las APIs permiten la integración de aplicaciones?

Las APIs permiten la integración de aplicaciones proporcionando un conjunto de interfaces de programación de aplicaciones (API) que permiten que una aplicación se comuniquen con otra.

Las aplicaciones pueden enviar solicitudes y recibir respuestas a través de la API de una aplicación externa para acceder a sus datos y funcionalidades. Por ejemplo, una aplicación de redes sociales puede proporcionar una API que permita a los desarrolladores de aplicaciones externas acceder a datos de usuario, publicaciones y otras funciones.





Ejemplos de uso de APIs

Google Maps API

Es una API de geolocalización que permite a los desarrolladores integrar mapas, direcciones y lugares en sus propias aplicaciones.

Por ejemplo, una aplicación de reservas de hoteles puede utilizar la API de Google Maps para mostrar la ubicación del hotel en un mapa y proporcionar direcciones de conducción.

Twitter API

Permite a los desarrolladores de aplicaciones acceder a los datos de Twitter, como tweets, mensajes directos y listas de usuarios. Los desarrolladores pueden utilizar la

API de Twitter para crear aplicaciones de análisis de sentimientos que analicen los tweets para determinar el tono general de los comentarios sobre un tema en particular.

Facebook API

Proporciona acceso a datos de Facebook, como publicaciones, comentarios y perfiles de usuario. Los desarrolladores pueden utilizar la API de Facebook para integrar la funcionalidad de inicio de sesión de Facebook en sus propias aplicaciones o para mostrar publicaciones de Facebook en su aplicación.

Ejercicio práctico

Labs 01

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-01>





Conceptos básicos de **REST** y su utilización en el **diseño de APIs**

¿Qué es **REST** y cómo se relaciona con las **APIs**?

REST (Representational State Transfer) es un conjunto de principios y reglas para el diseño de arquitecturas web que utilizan HTTP para la transferencia de datos. REST se basa en el uso de recursos y en la representación de los mismos mediante URLs y métodos HTTP.

Una API RESTful expone recursos a través de URLs y utiliza los métodos HTTP (GET, POST, PUT, DELETE) para realizar operaciones en esos recursos. Un cliente podría entonces utilizar un método HTTP como GET para obtener información sobre ese usuario o un método HTTP como PUT para actualizar la información.

```
GET http://ac.apis.com/user/1 HTTP/1.1
{
  "name": "pedro",
  "age": 23
}

POST http://ac.apis.com/user/1 HTTP/1.1
content-type: application/json

{
  "age": 30
}
```

Principios fundamentales de **REST**

Los principios fundamentales de REST incluyen:

Recursos

Los recursos son la pieza fundamental de una API RESTful. Un recurso es cualquier cosa que pueda ser identificada con una URL, como un objeto, una imagen, un archivo, etc. Un recurso puede tener múltiples representaciones, como JSON, XML, HTML, etc.

Verbos HTTP

REST utiliza los verbos HTTP (GET, POST, PUT, DELETE) para indicar la acción que se debe realizar en un recurso.

URIs

Las URIs (Uniform Resource Identifiers) son las URLs que se utilizan para identificar los recursos en una API RESTful. Las URIs deben ser únicas e identificar de manera unívoca cada recurso.

Representación de recursos

Cada recurso en una API RESTful puede tener múltiples representaciones, como JSON, XML, HTML, etc. Las aplicaciones que consumen la API pueden solicitar una representación específica utilizando encabezados HTTP como Accept.

```
Shell
POST https://example.com/comments HTTP/1.1
content-type: application/json

{
  "name": "sample",
  "time": "Wed, 21 Oct 2015 18:27:50 GMT"
}
```

Ventajas de utilizar **REST** en el diseño de **APIs**

Escalabilidad

Una API RESTful puede ser fácilmente escalada horizontalmente, lo que significa que se pueden agregar más servidores para manejar una mayor cantidad de tráfico.

Flexibilidad

REST permite que los clientes soliciten recursos específicos y que los servidores devuelvan sólo la información relevante para esa solicitud.

Bajo acoplamiento

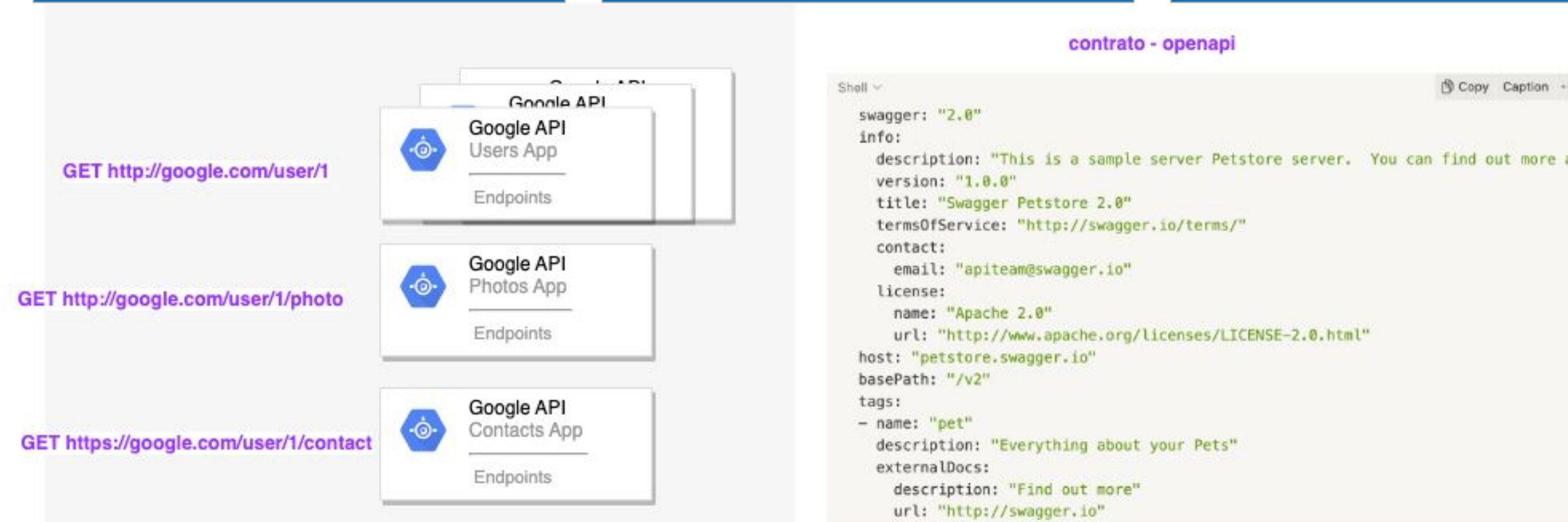
REST se basa en un conjunto de principios y reglas bien definidos que permiten que los clientes y los servidores se comuniquen de manera independiente.

Alta visibilidad

Al utilizar URIs y verbos HTTP para acceder a los recursos, una API RESTful se convierte en una interfaz muy visible para los desarrolladores y usuarios finales

Seguridad

REST utiliza el protocolo HTTPS para asegurar la comunicación entre los clientes y los servidores.



Ejercicio práctico

Labs 02

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-02>



Receso

15 min





Diseño de la estructura de la API: **rutas y métodos HTTP**

¿Cómo se diseña la **estructura de una API**?

1

Define los recursos

Identifica los recursos que serán parte de la API y asigna una URI única a cada uno de ellos.

2

Define los métodos HTTP

Cada recurso debe tener un conjunto de métodos HTTP que se puedan utilizar para interactuar con él. Los métodos HTTP más comunes son GET, POST, PUT, DELETE y PATCH.

3

Define los formatos de respuesta

Define los formatos en los que se devolverán las respuestas de la API. Los formatos más comunes son JSON, XML y HTML.

4

Define los códigos de estado HTTP

Define los códigos de estado HTTP que se utilizarán en las respuestas de la API. Los códigos de estado HTTP son códigos numéricos que indican el resultado de una operación. Por ejemplo, un código de estado 200 indica que la operación se ha completado con éxito.

5

Considera la escalabilidad

Asegúrate de que la API sea escalable y pueda manejar un gran número de solicitudes. Utiliza patrones como la caché, la paginación y la compresión para mejorar el rendimiento.

6

Considera la documentación

La documentación es esencial para una API bien diseñada. Proporciona documentación clara y concisa sobre cómo utilizar la API, incluyendo ejemplos y descripciones detalladas de cada recurso y método.



Métodos HTTP comunes

Los métodos HTTP son una parte esencial del diseño de una API RESTful. Existen varios métodos HTTP, pero los más comunes son GET, POST, PUT y DELETE.

GET

Se utiliza para recuperar información de un recurso. Por ejemplo, si tenemos una API de películas, podemos usar GET para obtener una lista de todas las películas.

POST

Se utiliza para crear un nuevo recurso. Por ejemplo, si queremos agregar una nueva película a nuestra base de datos, podemos usar POST para enviar los datos de la película a la API.

PUT

Se utiliza para actualizar un recurso existente. Por ejemplo, si queremos actualizar la información de una película existente, podemos usar PUT para enviar los nuevos datos de la película a la API.

DELETE

Se utiliza para eliminar un recurso existente. Por ejemplo, si queremos eliminar una película de nuestra base de datos, podemos usar DELETE para indicar a la API que elimine ese recurso.

Rutas del API

Las rutas de una API son las URLs que se utilizan para acceder a los diferentes recursos de la API. La estructura de una ruta depende del diseño de la API, pero suele seguir un patrón similar para cada recurso.

Ejemplos de rutas de API:

- /movies: devuelve una lista de todas las películas disponibles.
- /movies/123: devuelve los detalles de la película con ID 123.
- /movies/123/reviews: devuelve una lista de reseñas de la película con ID 123.
- /users: devuelve una lista de todos los usuarios registrados
- /users/456: devuelve los detalles del usuario con ID 456.
- /users/456/movies: devuelve una lista de todas las películas que el usuario con ID 456 ha visto.
- /search?query=terminator: devuelve una lista de todas las películas que coinciden con el término de búsqueda "terminator".

Formatos de Mensajería

Los formatos de solicitud y respuesta más comunes son JSON (JavaScript Object Notation) y XML (eXtensible Markup Language).

JSON es un formato de texto ligero para intercambio de datos. Es fácil de leer y escribir para los humanos, y fácil de analizar y generar para las máquinas. Es el formato más utilizado en las APIs RESTful debido a su simplicidad, compatibilidad con todos los navegadores web y lenguajes de programación, y soporte para estructuras de datos complejas.

Por otro lado, XML es un formato de texto más pesado y estructurado que JSON. Se utiliza principalmente en entornos empresariales y aplicaciones legacy, y es útil para intercambiar datos estructurados entre diferentes sistemas. Sin embargo, XML puede ser más complicado de leer y escribir para los humanos que JSON.

Ejemplo



Shell ▾

Copy Caption ...

```
GET http://ac.apis.com/user/1 HTTP/1.1
```

```
{  
  "name": "pedro",  
  "age": 23  
}
```

```
---
```

```
POST http://ac.apis.com/user HTTP/1.1  
content-type: application/json
```

```
{  
  "name": "miguel",  
  "age": 23  
}
```

```
HTTP/1.1 201
```

```
---
```

```
PUT http://ac.apis.com/user/2 HTTP/1.1  
content-type: application/json
```

```
{  
  "age": 30  
}
```

```
HTTP/1.1 200
```

```
---
```

```
DELETE http://ac.apis.com/user/1 HTTP/1.1
```

```
HTTP/1.1 200|
```

Ejercicio práctico

Labs 03

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-03>





Autenticación y autorización en APIs REST

¿Qué es la autenticación y la autorización en el contexto de las APIs?

La autenticación es el proceso de verificar la identidad del usuario que está realizando la solicitud de la API, mientras que la autorización es el proceso de determinar si el usuario tiene permisos para realizar la acción solicitada.

La autenticación puede ser realizada de diversas maneras, como mediante el uso de tokens de acceso, contraseñas, certificados digitales, etc. En algunos casos, se puede utilizar un proveedor de identidad externo, como OAuth, para autenticar al usuario.

Una vez que el usuario ha sido autenticado, se debe determinar si tiene los permisos necesarios para realizar la acción solicitada. Esto se logra mediante la autorización. Los permisos pueden ser otorgados en función de roles, grupos, permisos específicos, etc.

Métodos de autenticación comunes



Existen varios tipos de autenticación utilizados en el contexto de las APIs, entre ellos:

Autenticación básica

Es un método de autenticación simple en el que se envía una clave de API y un secreto de API como encabezados de autenticación en cada solicitud. Aunque es fácil de implementar, la autenticación básica es menos segura que otros métodos de autenticación.

Autenticación mediante OAuth

OAuth es un protocolo de autenticación y autorización que permite a los usuarios autorizar una aplicación para acceder a sus datos sin proporcionar su contraseña. OAuth se utiliza comúnmente en aplicaciones que necesitan acceder a recursos de terceros, como cuentas de redes sociales o servicios de almacenamiento en la nube.

Autenticación basada en token

En este método, el cliente envía un token que se genera previamente por el servidor de autenticación y que contiene información sobre el usuario autenticado. Este token se utiliza para validar la identidad del usuario en cada solicitud posterior.

Shell ▾

```
### 200
GET http://localhost:8000/api/v1/private HTTP/1.1
Authorization: Basic dXN1YXJpbyBjb250cmFzc2E=|
```

Shell ▾

```
###200
GET http://localhost:8000/api/v1/api-key
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1
```

###200

```
|
POST https://oauth2.ejemplo.com/token
Content-Type: application/x-www-form-urlencoded
-d 'grant_type=password&username=john.doe&password=myspassword'
```



¿Qué son los **token JWT y JWE?**

JWT (JSON Web Token) es un formato de token estándar abierto que se utiliza para transmitir información entre dos partes de forma segura. El token JWT consta de tres partes: un encabezado que describe el tipo de token y el algoritmo de cifrado utilizado; un cuerpo que contiene la información que se va a transmitir; y una firma digital que se utiliza para verificar la integridad del token. Los tokens JWT son muy populares para la autenticación en aplicaciones web y móviles debido a su simplicidad, escalabilidad y flexibilidad.

Por otro lado, JWE (JSON Web Encryption) es un formato de token utilizado para cifrar datos sensibles y transmitirlos de manera segura entre dos partes. JWE permite a los desarrolladores cifrar el cuerpo de un mensaje utilizando una clave pública, y solo el destinatario autorizado puede descifrarlo utilizando su clave privada. JWE también puede utilizarse en combinación con JWT para crear tokens cifrados y firmados digitalmente para una mayor seguridad.



¿Qué son las **cabeceras HMAC?**

(Hash-based Message Authentication Code) Son una forma de autenticación basada en claves secretas compartidas entre el cliente y el servidor. Estas cabeceras se utilizan para proteger la integridad de los datos que se transmiten entre el cliente y el servidor, así como para verificar la autenticidad del mensaje.

En este proceso de autenticación, el cliente crea una cadena de autenticación que incluye información adicional, como una marca de tiempo, un nonce (número utilizado una vez), y los datos que se envían al servidor. Luego, esta cadena se firma digitalmente utilizando una clave secreta compartida entre el cliente y el servidor. La firma resultante se incluye en una cabecera de la solicitud HTTP junto con la información adicional.

El servidor puede verificar la autenticidad del mensaje recibido mediante la verificación de la firma digital en la cabecera de la solicitud HTTP utilizando la misma clave secreta compartida. Si la firma es válida, el servidor puede estar seguro de que el mensaje no ha sido modificado durante la transmisión y que proviene de un cliente autenticado.

Almuerzo

45 min



Ejemplo de HMAC

En este ejemplo, la solicitud HTTP utiliza el método POST para enviar datos a la URL `https://api.example.com/resource`. La solicitud incluye una cabecera de contenido `Content-Type` para indicar que los datos se envían en formato JSON.

La autenticación se realiza utilizando la cabecera `Authorization`, que especifica que se está utilizando HMAC para autenticar la solicitud. La cabecera incluye la clave API (en lugar de la clave secreta) y un hash de los datos que se envían en la solicitud. El hash se genera utilizando la clave secreta compartida entre el cliente y el servidor.

La opción `-d` de `curl` se utiliza para especificar los datos que se envían en la solicitud. En este caso, se envía un objeto JSON que contiene una propiedad `"data"` con un valor `"example"`.

Al utilizar cabeceras HMAC, se asegura que la solicitud se transmita de forma segura y que la integridad de los datos se mantenga durante la transmisión.

```
Shell ▾  
curl -H "Content-Type: application/json" \  
-H "Authorization: HMAC <api_key>:<hash>" \  
-X POST \  
-d '{"data": "example"}' \  
https://api.example.com/resource
```


Autorización en una **API REST**

Algunos de los tipos de autorización comunes incluyen:

OAuth

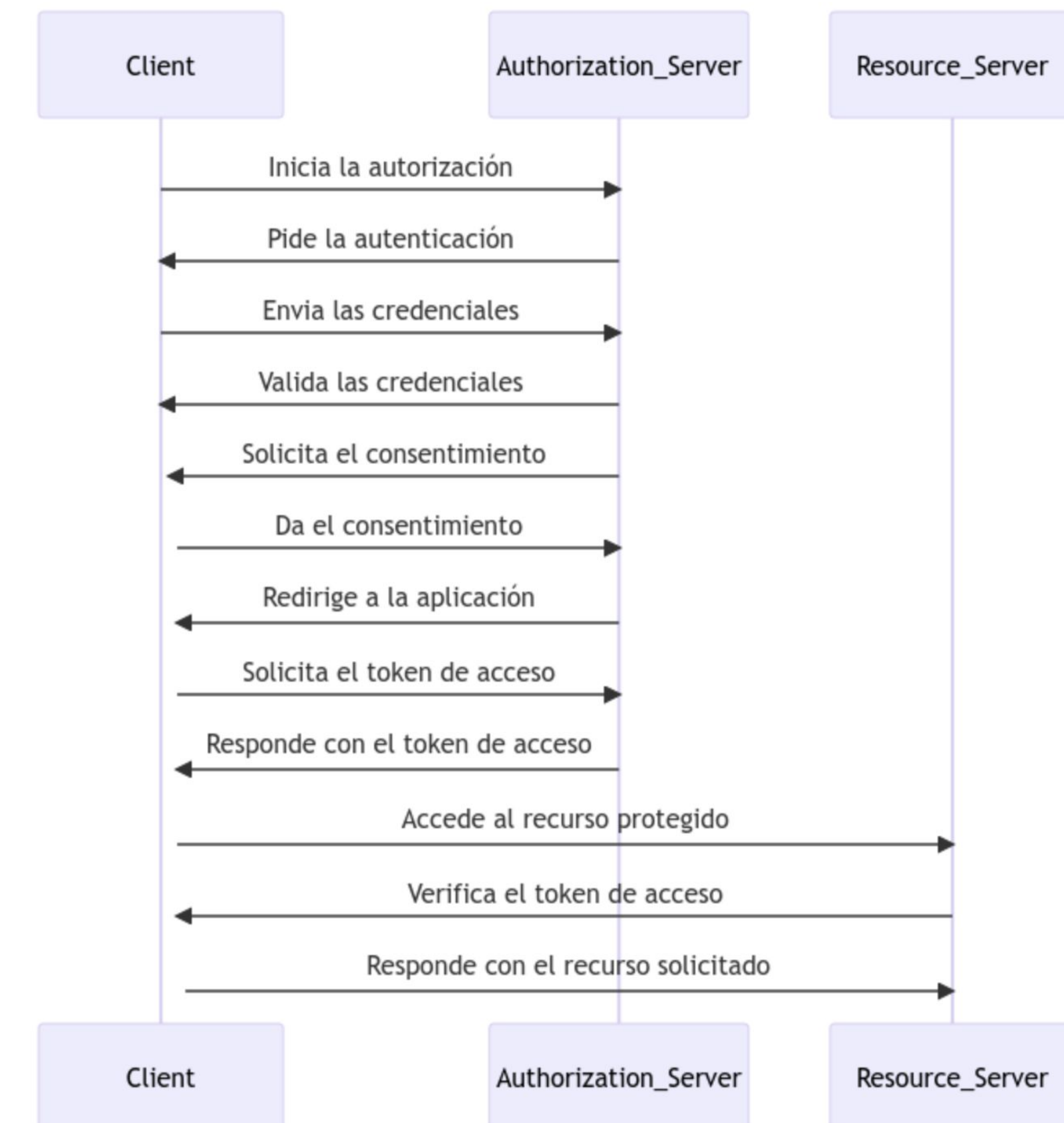
Es un protocolo de autorización abierto y ampliamente utilizado que permite que los usuarios autoricen aplicaciones de terceros a acceder a sus recursos protegidos sin tener que compartir sus credenciales de inicio de sesión.

Token de acceso

Es un tipo de autorización en el que se emite un token de acceso al usuario para acceder a los recursos de la API. Este token se utiliza para autenticar al usuario en cada solicitud subsiguiente.

Autenticación basada en certificados

Este método de autenticación utiliza certificados de seguridad para autenticar a los usuarios. Es uno de los métodos de autenticación más seguros, pero también es más complejo de implementar que otros métodos.



Ejercicio práctico

Labs 04

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-04>





Gestión de errores y **códigos de respuesta HTTP**

¿Por qué es importante la **gestión de errores** en una **API REST**?

La **gestión de errores** es una parte crucial del diseño de cualquier API REST, ya que puede marcar la **diferencia entre una buena y una mala experiencia del usuario**. Los errores pueden surgir por diversas razones, como datos incorrectos, falta de autorización o problemas de conectividad, entre otros.

Es importante que los errores se gestionen adecuadamente para que el cliente tenga una idea clara de lo que ha sucedido y cómo solucionarlo. Los errores no gestionados pueden provocar que los clientes se frustren, abandonen la aplicación y, en última instancia, pierdan la confianza en la API.











Las APIs REST deben estar diseñadas para manejar y devolver errores en un formato coherente y fácil de entender. Los mensajes de error deben ser claros y proporcionar detalles suficientes para que el cliente sepa qué salió mal y cómo solucionarlo. Además, es importante proporcionar **códigos de estado HTTP adecuados para que el cliente pueda entender rápidamente la naturaleza del problema y tomar las medidas necesarias**.

Una buena gestión de errores también puede ayudar a los desarrolladores de la API a identificar problemas y resolverlos de manera más rápida y eficiente, lo que puede mejorar la calidad y la estabilidad de la API.

Códigos de respuesta HTTP comunes

En el contexto de una API REST, estos códigos se utilizan para indicar si una solicitud ha sido exitosa o si ha habido algún tipo de error. Algunos de los códigos de respuesta HTTP más comunes incluyen:

-  **200 OK:** indica que la solicitud ha sido exitosa y que se ha devuelto una respuesta.
-  **201 Created:** indica que una nueva entidad ha sido creada en el servidor.
-  **204 No Content:** indica que la solicitud ha sido exitosa, pero no se ha devuelto ningún contenido.
-  **400 Bad Request:** indica que la solicitud ha sido malformada o no ha podido ser procesada por el servidor.

-  **401 Unauthorized:** indica que se requiere autenticación para acceder al recurso solicitado.
-  **403 Forbidden:** indica que el cliente no tiene permisos para acceder al recurso solicitado.
-  **404 Not Found:** indica que el recurso solicitado no ha sido encontrado en el servidor.
-  **500 Internal Server Error:** indica que se ha producido un error interno en el servidor.

Es importante que los desarrolladores de API utilicen los códigos de respuesta HTTP de manera adecuada para que los clientes de la API puedan entender correctamente el estado de sus solicitudes. De esta manera, se puede proporcionar una experiencia de usuario más clara y fácil de usar.

Receso

15 min



¿Cómo diseñar una respuesta de error efectiva?

Algunas prácticas recomendadas para diseñar una respuesta de error efectiva incluyen:

Proporcionar un mensaje de error claro y conciso

El mensaje de error debe ser lo suficientemente claro para que los desarrolladores comprendan lo que salió mal, pero también debe ser breve y conciso para facilitar la lectura y la comprensión.

Proporcionar un código de estado HTTP apropiado

El código de estado HTTP indica el tipo de error que ha ocurrido y debe ser incluido en la respuesta de error. Algunos ejemplos de códigos de estado HTTP comunes incluyen 400 para errores de solicitud incorrecta, 401 para errores de autenticación, 404 para recursos no encontrados y 500 para errores del servidor.

Proporcionar detalles adicionales

A veces, un mensaje de error claro y un código de estado HTTP no son suficientes para solucionar un problema. En tales casos, se pueden proporcionar detalles adicionales, como una descripción más detallada del error, una URL de soporte o un enlace a la documentación relevante.

Seguir una estructura consistente

Es importante seguir una estructura consistente en todas las respuestas de error para facilitar la comprensión y el manejo de los errores.

Ejercicio práctico

Labs 05

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-05>





Utilización de HTTPS y seguridad en el intercambio de datos

¿Por qué es importante **la seguridad** en el **intercambio de datos**?

La seguridad en el intercambio de datos es importante porque los datos transmitidos entre el cliente y el servidor pueden ser sensibles y confidenciales. Sin la seguridad adecuada, los datos pueden ser interceptados y manipulados por terceros malintencionados, lo que puede llevar a graves consecuencias, como la pérdida de la privacidad, la exposición de información personal y financiera, el robo de identidad y el acceso no autorizado a sistemas y recursos críticos.

Además, la falta de seguridad también puede afectar la integridad y la disponibilidad de los datos, lo que puede provocar la corrupción de datos, la interrupción del servicio y la pérdida de confianza de los usuarios. Por lo tanto, es importante implementar medidas de seguridad adecuadas, como la encriptación de datos, la autenticación y autorización, para garantizar que los datos transmitidos sean seguros y confiables.

¿Cómo utilizar HTTPS para proteger el intercambio de datos?

HTTPS (Hypertext Transfer Protocol Secure) es un protocolo de comunicación seguro utilizado para proteger el intercambio de datos en la web. Utiliza el cifrado de extremo a extremo para proteger los datos que se transfieren entre un servidor y un cliente, como un navegador web.

Para utilizar HTTPS en una API REST, es necesario adquirir un certificado SSL (Secure Sockets Layer) o TLS (Transport Layer Security) válido y configurar el servidor para que acepte conexiones HTTPS. Una vez que el servidor está configurado para HTTPS, cualquier cliente que se conecte a la API debe hacerlo a través de una URL que comience con "https://" en lugar de "http://".

La utilización de HTTPS es importante para proteger el intercambio de datos en una API REST porque permite la autenticación y el cifrado de los datos que se transfieren, lo que reduce el riesgo de que los datos sean interceptados o manipulados por terceros malintencionados. Además, el uso de HTTPS ayuda a garantizar que los datos se transmitan de forma segura y que no se vean comprometidos durante la transmisión.

Mejores prácticas para la seguridad en el intercambio de datos

Cuando se trata de la seguridad en el intercambio de datos a través de APIs, hay varias mejores prácticas que se deben seguir para garantizar la protección de los datos sensibles. Algunas de estas prácticas incluyen:

Utilizar HTTPS en lugar de HTTP para el intercambio de datos. HTTPS utiliza SSL/TLS para cifrar los datos durante la transmisión, lo que significa que son mucho más difíciles de interceptar o manipular.

Utilizar la autenticación y la autorización para asegurarse de que solo los usuarios autorizados puedan acceder a los datos sensibles. Esto puede incluir el uso de tokens de acceso, claves API y otros métodos de autenticación.

Utilizar cifrado para proteger los datos sensibles mientras están almacenados en la base de datos. Esto puede incluir el uso de cifrado de extremo a extremo y técnicas de hashing para proteger la información confidencial.

Implementar un control de acceso adecuado para limitar el acceso a los datos sensibles sólo a aquellos que los necesitan. Esto puede incluir el uso de roles y permisos para limitar el acceso a ciertos datos y funcionalidades.

Realizar pruebas de penetración regulares para identificar posibles vulnerabilidades y problemas de seguridad. Esto puede incluir pruebas de seguridad automatizadas y manuales para asegurarse de que la API esté protegida contra ataques.

Siguiendo estas mejores prácticas, se puede garantizar que la seguridad sea una prioridad en el intercambio de datos a través de APIs, lo que puede ayudar a proteger los datos de los usuarios y reducir el riesgo de violaciones de seguridad.

Ejercicio práctico

Labs 06

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-06>



Agenda

Día 2



Gestión de versiones de la API



Documentación de la API: Swagger y otras herramientas.



Optimización de la performance y escalabilidad de la API.



Pruebas de API's usando Postman y manejo de mock services.



Ejemplos prácticos de diseño de APIs REST utilizando diferentes tecnologías y plataformas.



Evaluación del módulo.



Gestión de versiones de la API

¿Por qué es importante la gestión de versiones en una API REST?

La gestión de versiones es importante en una API REST porque permite controlar y evolucionar el desarrollo de la API sin afectar a los usuarios que ya están utilizando versiones anteriores de la misma. Al implementar cambios en una API, se pueden introducir nuevas funcionalidades, solucionar errores o mejorar la eficiencia de la misma, lo que podría tener un impacto negativo en las aplicaciones que dependen de ella. Por lo tanto, al utilizar la gestión de versiones, se pueden introducir cambios gradualmente y de manera controlada, lo que permite a los usuarios actualizar sus aplicaciones de manera coordinada con los cambios en la API.

Entre los principales beneficios de la gestión de versiones en una API REST se incluyen:



Facilita el mantenimiento y evolución de la API



Mejora la experiencia del usuario, ya que les permite actualizar sus aplicaciones de manera coordinada con los cambios en la API.



Permite a los desarrolladores tener un mayor control sobre los cambios y la compatibilidad.



Evita posibles problemas de compatibilidad y errores en las aplicaciones que dependen de la API.

Métodos comunes para la gestión de versiones en una API REST

Existen varias formas de gestionar versiones en una API REST, entre los métodos más comunes se encuentran:

Versionado en la URL

se puede incluir la versión en la URL de la API. Por ejemplo, si la primera versión de la API se llama "v1", la URL puede ser "https://api.example.com/v1/users".

Versionado en el header

se puede incluir la versión en un header personalizado en la solicitud HTTP. Por ejemplo, un header personalizado llamado "API-Version".

Versionado por parámetros

se puede incluir la versión en los parámetros enviados. Por ejemplo, si la primera versión de la API se envía en el query string, la URL puede ser "https://api.example.com/users?version=v3".

Versionamiento Semántico

Estrategia – MAYOR.MENOR.PARCHE:

En donde:

- La versión MAYOR cuando realizas un cambio incompatible en el API,
- La versión MENOR cuando añades funcionalidad que compatible con versiones anteriores, y
- La versión PARCHE cuando reparas errores compatibles con versiones anteriores.

Especificaciones del versionado semántico:

Un número de versión normal DEBE tener la forma de X.Y.Z donde X, Y y Z son números enteros no negativos, y NO DEBEN ser precedidos de ceros. X es la versión mayor, Y es la versión menor, y Z es la versión parche. Cada elemento DEBE incrementarse numéricamente. Por ejemplo: 1.9.0 -> 1.10.0 -> 1.11.0

Una vez que el paquete versionado ha sido publicado, el contenido de esa versión NO DEBE ser modificado. Cualquier modificación DEBE ser publicada como una nueva versión.

Una versión mayor en cero (0.y.z) se considera como desarrollo inicial. Todo PUEDE cambiar en cualquier momento. El API público NO DEBERÍA ser considerado estable.

La versión 1.0.0 define el API público. La manera en que cada número de versión es incrementado después de esta publicación dependerá de su API público y cómo cambia.

Cambios **retrocompatibles**

No todos los cambios en el API tienen impacto sobre los consumidores de la misma (al menos, no deberían tenerlo). A estos cambios se les suele denominar cambios retrocompatibles.

Ejemplo de cambios en un API que NO deberían afectar a los consumidores:

Añadir nuevas operaciones al servicio. En REST sería añadir nuevas acciones sobre un recurso (PUT, POST, DELETE, etc.)

Añadir parámetros de entrada opcionales a peticiones sobre recursos ya existentes. Ej: un nuevo parámetro de filtrado en un GET sobre una colección de recursos.

Modificar parámetros de entrada de obligatorios a opcionales. Ej: al crear un recurso, una propiedad de dicho recurso que antes fuese obligatoria y que pase a opcional.

Añadir nuevas propiedades en la representación de un recurso que nos devuelve el servidor. Ej: ahora a un objeto de "Persona" que anteriormente estuviera compuesto por DNI y nombre, le añadimos un nuevo campo edad.

Cambios **retrocompatibles**

Ejemplos de cambios en un API que SI deberían afectar a los consumidores:

Eliminar operaciones o acciones sobre un recurso. Ej: Ya no se aceptan peticiones POST sobre un recurso.

Añadir nuevos parámetros de entrada obligatorios. Ej: ahora para dar de alta un recurso hay que enviar en el cuerpo de la petición un nuevo campo requerido.

Modificar parámetros de entrada de opcional a obligatorio. Ej: ahora al crear un recurso Persona, el campo edad, que antes era opcional, ahora es obligatorio.

Modificar un parámetro en operaciones (verbos sobre recursos) ya existentes. También aplicable a la eliminación de parámetros. Ej: al consultar un recurso ya no se devuelve determinado campo. Otro ejemplo: un campo que antes era una cadena de caracteres, ahora es numérico.

Añadir nuevas respuestas en operaciones ya existentes. Ej: ahora la creación de un recurso puede devolver un código de respuesta 412.



Ejemplos de la implementación de **versionado de API en aplicaciones reales**

Twitter utiliza una estrategia de versionado en la URL para sus API. Cada versión de la API tiene una URL específica, lo que permite a los desarrolladores de aplicaciones apuntar a una versión específica de la API.

GitHub utiliza una estrategia de versionado basada en la cabecera de la solicitud HTTP. Los desarrolladores de aplicaciones deben incluir una cabecera de versión en sus solicitudes para apuntar a una versión específica de la API.

AWS utiliza una estrategia de versionado basada en la URL. Cada versión de la API tiene una URL específica, lo que permite a los desarrolladores de aplicaciones apuntar a una versión específica de la API.

Ejercicio práctico

Labs 07

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-07>



Receso

15 min





Documentación de la API: **Swagger y otras herramientas**



¿Por qué es importante documentar una API REST?

Facilita la comprensión

La documentación clara y completa de la API REST proporciona información detallada sobre los recursos, parámetros y formatos de datos que se pueden utilizar en la integración. Esto ayuda a los desarrolladores y usuarios a entender cómo deben interactuar con la API y qué esperar como respuesta.

Ahorra tiempo y recursos

La documentación de la API REST ayuda a los desarrolladores y usuarios a encontrar rápidamente lo que necesitan, sin tener que buscar en la documentación de la plataforma o pedir ayuda al equipo de soporte. Esto ahorra tiempo y recursos, y permite que los desarrolladores y usuarios trabajen de manera más eficiente.

Mejora la calidad del código

Una buena documentación de la API REST permite que los desarrolladores utilicen de manera efectiva las funciones de la API, lo que a su vez mejora la calidad del código y reduce la cantidad de errores.

Fomenta la colaboración

La documentación de la API REST fomenta la colaboración entre desarrolladores y usuarios, lo que puede llevar a mejoras y nuevas funcionalidades. Además, la documentación puede ser utilizada para la creación de pruebas automatizadas que ayudan a detectar problemas antes de que lleguen a producción.

Simplifica la integración

La documentación de la API REST proporciona información detallada sobre cómo deben ser los parámetros y las solicitudes para interactuar con la API. Esto simplifica la integración, ya que los desarrolladores y usuarios pueden utilizar la información de la documentación para crear solicitudes efectivas.

Métodos comunes para la **documentación de una API REST**

Swagger/OpenAPI

Es una herramienta de documentación de API muy popular que permite describir la estructura y operaciones de una API REST. La especificación OpenAPI, que antes era conocida como Swagger, es un estándar de documentación de API que utiliza YAML o JSON para describir la estructura de una API REST, incluyendo la definición de recursos, operaciones, parámetros y respuestas. Es muy utilizada por desarrolladores y empresas debido a su facilidad de uso y compatibilidad con muchas plataformas.

RAML (RESTful API Modeling Language)

Es otro lenguaje de especificación de API REST que permite describir la estructura y funcionalidad de una API. Se utiliza para documentar y diseñar la API de forma colaborativa, permitiendo que todos los interesados tengan una vista clara del funcionamiento de la API. Permite describir los recursos, métodos HTTP, parámetros y respuestas de la API.

Swagger y RAML

Shell ▾

```
swagger: "2.0"
info:
  description: "This is a sample server Petstore server. You can find out more ;
  version: "1.0.0"
  title: "Swagger Petstore 2.0"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "apiteam@swagger.io"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
host: "petstore.swagger.io"
basePath: "/v2"
tags:
- name: "pet"
  description: "Everything about your Pets"
  externalDocs:
    description: "Find out more"
    url: "http://swagger.io"
schemes:
- "https"
- "http"
paths:
  /pet:
    post:
      tags:
      - "pet"
```

Shell ▾

```
##RAML 1.0
title: Mobile Order API
baseUri: http://localhost:8081/api
version: 1.0

uses:
  assets: assets.lib.raml

annotationTypes:
  monitoringInterval:
    type: integer

/orders:
  displayName: Orders
  get:
    is: [ assets.paging ]
    (monitoringInterval): 30
    description: Lists all orders of a specific user
    queryParameters:|
      userId:
        type: string
        description: use to query all orders of a user
  post:
    /{orderId}:
      get:
        responses:
          200:
            body:
              application/json:
                type: assets.Order
              application/xml:
```

Ejercicio práctico

Labs 08

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-08>



Almuerzo

45 min





Optimización de la **performance** y **escalabilidad** de la **API**



¿Por qué es importante la optimización de la **performance y la escalabilidad de una API REST?**

Experiencia de usuario

Una API REST que es lenta y poco fiable puede afectar negativamente la experiencia del usuario. Los usuarios esperan una respuesta rápida y eficiente de una API REST, por lo que la optimización de la performance es esencial para garantizar que la experiencia del usuario sea satisfactoria.

Carga de trabajo

A medida que una aplicación o sistema crece y se vuelve más popular, la carga de trabajo en la API REST también aumenta. Si la API REST no está diseñada para manejar una carga de trabajo mayor, puede generar tiempos de respuesta lentos, errores y tiempo de inactividad, lo que puede afectar la calidad del servicio y la satisfacción del usuario.

Costos

Si una API REST no está optimizada para la performance y la escalabilidad, puede ser necesario invertir en hardware y software adicional para manejar la carga de trabajo, lo que puede aumentar los costos operativos y afectar la rentabilidad de la aplicación o sistema.

Competencia

En un mercado competitivo, la performance y escalabilidad de una API REST pueden ser un factor importante en la elección de los usuarios y clientes. Si una API REST no cumple con los requisitos de performance y escalabilidad, es probable que los usuarios y clientes elijan alternativas que sí lo hagan.



Métodos comunes para la optimización de la **performance y escalabilidad de una API REST**

Caching: La caché es una técnica que permite almacenar temporalmente los datos en memoria para mejorar la performance de la API REST. Esto significa que las solicitudes futuras para los mismos datos pueden ser atendidas desde la caché en lugar de tener que buscarlos en la base de datos o en otra fuente de datos. Esto reduce el tiempo de respuesta y la carga en la base de datos o en la fuente de datos.

Compresión: La compresión es una técnica que permite reducir el tamaño de las respuestas de la API REST, lo que a su vez reduce el tiempo de transferencia de datos. Esto se logra mediante la eliminación de datos redundantes y la codificación de los datos en un formato más compacto.

Indexación: La indexación es una técnica que se utiliza para mejorar la velocidad de búsqueda de datos en una base de datos. Al indexar las columnas de la base de datos utilizadas con mayor frecuencia en las consultas de la API REST, se puede mejorar significativamente la velocidad de búsqueda y reducir el tiempo de respuesta.

Métodos comunes para la optimización de la **performance y escalabilidad de una API REST**

Escalabilidad horizontal

La escalabilidad horizontal es una técnica que se utiliza para mejorar la escalabilidad de la API REST al agregar más servidores a la infraestructura en lugar de agregar recursos adicionales a un solo servidor. Esto permite a la API REST manejar una carga de trabajo mayor y reducir la posibilidad de tiempo de inactividad.

Paginación

La paginación es una técnica que se utiliza para limitar el número de resultados devueltos por la API REST en una sola solicitud. Esto reduce la carga en la base de datos y mejora la velocidad de respuesta de la API REST. Además, la paginación permite a los usuarios recuperar grandes conjuntos de datos de manera eficiente mediante la recuperación de datos en pequeñas cantidades de forma secuencial.



Ejemplos de la implementación de optimización de la performance y escalabilidad en aplicaciones reales

Facebook utiliza técnicas de caché para reducir la carga en sus servidores y mejorar la performance de su API REST. Además, Facebook ha implementado técnicas de escalabilidad horizontal mediante el uso de tecnologías como Cassandra y Hadoop para distribuir la carga entre múltiples servidores.

Netflix utiliza técnicas de caché y compresión para mejorar la performance de su API REST. Además, Netflix ha implementado técnicas de escalabilidad horizontal mediante el uso de tecnologías como Amazon Web Services (AWS) para distribuir la carga entre múltiples servidores.

Twitter utiliza técnicas de caché y compresión para mejorar la performance de su API REST. Además, Twitter ha implementado técnicas de escalabilidad horizontal mediante el uso de tecnologías como Apache Mesos y Kubernetes para distribuir la carga entre múltiples servidores.

Uber utiliza técnicas de caché y paginación para mejorar la performance de su API REST. Además, Uber ha implementado técnicas de escalabilidad horizontal mediante el uso de tecnologías como Apache Cassandra y Docker para distribuir la carga entre múltiples servidores.

Receso

15 min





Ejemplos prácticos de **diseño de APIs REST**

Diseño de API REST utilizando diferentes tecnologías y plataformas

.NET es un marco de aplicación de Microsoft que se utiliza para desarrollar aplicaciones web y API REST. .NET proporciona una amplia variedad de bibliotecas y herramientas que hacen que el desarrollo de API REST sea más rápido y eficiente. Además, .NET es altamente escalable y se puede usar para crear API REST para proyectos de cualquier tamaño.

Go es un lenguaje de programación de código abierto que se utiliza para desarrollar aplicaciones web y API REST. Go es conocido por ser rápido y eficiente, lo que lo hace ideal para crear API REST escalables. Go también proporciona características como la concurrencia y la ejecución paralela, lo que permite que las API REST de Go manejen grandes cargas de trabajo.

Spring Boot es un marco de aplicación web de Java que se utiliza para desarrollar API REST. Spring Boot proporciona una estructura sólida para el desarrollo de API REST y facilita la creación de API RESTful. Spring Boot también proporciona características como configuración automática y una gran cantidad de bibliotecas que hacen que el desarrollo de API REST sea más eficiente.

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos, con una sintaxis simple y fácil de aprender. Es muy popular en el mundo de la programación debido a su capacidad para manejar tareas complejas con una sintaxis concisa, lo que lo convierte en una opción popular para la creación de aplicaciones, incluyendo APIs REST.

Consideraciones importantes al diseñar APIs REST en diferentes tecnologías y plataformas

Diseño de recursos:

Independientemente de la tecnología o plataforma utilizada, es fundamental que el diseño de recursos sea consistente y coherente en toda la API REST. Los recursos deben ser fáciles de identificar y deben representar de manera precisa los datos o las acciones que se están exponiendo a través de la API REST.

Escalabilidad:

Es importante diseñar la API REST con escalabilidad en mente, especialmente si se espera un gran volumen de solicitudes. La escalabilidad puede lograrse a través de la implementación de un diseño de API REST sin estado, la utilización de caché y la implementación de técnicas de partición de datos.

Seguridad:

La seguridad es fundamental en cualquier API REST. Es importante asegurarse de que la API REST esté protegida contra ataques como la inyección de SQL, la falsificación de solicitudes en sitios cruzados (XSS) y la falsificación de solicitudes de sitios entre sitios (CSRF). Además, es importante autenticar y autorizar a los usuarios de la API REST para garantizar que solo se acceda a los recursos adecuados.



Consideraciones importantes al diseñar APIs REST en diferentes tecnologías y plataformas

Rendimiento: El rendimiento es crucial en cualquier API REST. Para lograr un alto rendimiento, es importante optimizar la velocidad de respuesta de la API REST, minimizar el tamaño de las respuestas y utilizar técnicas de almacenamiento en caché. Además, es importante optimizar la consulta y manipulación de datos y minimizar el tiempo de procesamiento del servidor.

Documentación: La documentación es esencial para cualquier API REST, independientemente de la tecnología o plataforma utilizada. La documentación debe incluir información sobre los recursos, los métodos admitidos, los parámetros de entrada, las respuestas de salida y cualquier autenticación o autorización necesarias.



Buenas prácticas para el diseño de APIs REST en diferentes tecnologías y plataformas

Utilice nombres de recursos claros y consistentes:

Es importante utilizar nombres de recursos claros y coherentes en toda la API REST. Los nombres de recursos deben representar con precisión los datos o las acciones que se están exponiendo a través de la API REST.

Utilice verbos HTTP para indicar la acción:

Los verbos HTTP como GET, POST, PUT, DELETE y PATCH deben utilizarse para indicar la acción que se está realizando en un recurso determinado. Por ejemplo, la solicitud GET se utiliza para recuperar información, mientras que la solicitud POST se utiliza para enviar datos a un recurso.

Utilice códigos de estado HTTP para indicar el resultado de la operación:

Los códigos de estado HTTP deben utilizarse para indicar el resultado de una operación. Por ejemplo, el código de estado 200 indica que la operación se ha completado correctamente, mientras que el código de estado 404 indica que el recurso no se ha encontrado.

Utilice formatos de datos estandarizados:

Es importante utilizar formatos de datos estandarizados como JSON o XML para representar los datos. Esto asegura que los clientes y servidores puedan comunicarse con facilidad y reducir la posibilidad de errores.



Buenas prácticas para el diseño de APIs REST en diferentes tecnologías y plataformas

Use versiones de API:

Si se espera que la API cambie en el futuro, es importante utilizar versiones para la API. Esto permite a los desarrolladores y clientes actualizar sus aplicaciones y asegura que no se rompan cuando se produzcan cambios en la API.

Implemente la seguridad adecuada:

Es importante implementar la seguridad adecuada en la API REST para proteger los recursos y datos subyacentes. Esto puede incluir la autenticación y autorización, la encriptación de datos y la protección contra ataques de inyección de SQL, XSS y CSRF.

Documente la API REST:

Es importante documentar la API REST para que los clientes puedan comprender cómo interactuar con ella. La documentación debe incluir información sobre los recursos, los métodos admitidos, los parámetros de entrada, las respuestas de salida y cualquier autenticación o autorización necesarias.

Ejercicio práctico

Labs 10

URL

<https://github.com/academia-consultec/api-design/blob/develop/lab-10>



Evaluación teórica del módulo



Contacto

E. ilopez@consultec-ti.com

T. +507 60618404

Panamá



info@consultec-ti.com



@consulteclatam



@consultec-ti



consultec-ti.com



Gracias

¡Nos vemos pronto!