EXERCISES QUE TRABAJAREMOS EN LA CUE

- EXERCISE 1: DISEÑO DE UNA API RESTFULL

EXERCISE 1: DISEÑO DE UNA API RESTFULL

El objetivo del presente ejercicio es el diseño de API HTTP y RESTful, actualmente puede ser complicado, ya que no existe un estándar oficial y obligatorio. Básicamente, hay muchas formas de implementar una API, pero algunas de ellas se han probado en la práctica y se han adoptado ampliamente. Se aplicara las mejores prácticas para construir API HTTP y RESTful que contendrá:

- La estructura de URL
- Los métodos HTTP
- La creación y actualización de recursos
- El diseño de relaciones
- Los formatos de carga útil
- La paginación
- El control de versiones

Consideremos dos URL por el recurso de empleado, entonces tenemos:

1. URL que representa la colección de recursos

```
/employees
```

2. URL que representa un simple recurso

```
/employees/25
```

Para ello se utiliza sustantivos consistentemente plurales. Por ejemplo, en vez de usar `/empleado` podemos usar

```
/empleado/25
```

En general se evitá mezclar sustantivos en plural y en singular, lo cual es confuso y propenso a errores.

Use sustantivos en lugar de verbos para recursos. Esto mantendrá el API simple y la cantidad de URL baja. No se recomienda realizar lo siguiente:

```
/getAllEmployees
/getAllExternalEmployees
/createEmployee
/updateEmployee
```

DEFINICIÓN DE MÉTODOS HTTP QUE OPERAN EL RECURSO EMPLEADO

```
GET /employees
GET /employees?estado=externo
POST /employees
PUT /employees/15
```

Utilice direcciones URL para especificar el recurso empleado con los que se desea trabajar. Utilice los métodos HTTP para especificar ***qué*** hacer con este recurso. Con los cinco métodos HTTP `GET`, `POST`, `PUT`, `PATCH` y `DELETE`, puede proporcionar la funcionalidad CRUD (Crear, Leer, Actualizar, Eliminar) y más.

Lectura : use `GET` para leer recursos.

Crear : use `POST` o `PUT` para crear nuevos recursos.

Actualizar : use `PUT` y `PATCH` para actualizar los recursos existentes.

Eliminar : use `DELETE` para eliminar los recursos existentes.

COMPRENDER LA SEMÁNTICA DE LOS MÉTODOS HTTP

Definición de idempotencia : un método HTTP es idempotente cuando podemos ejecutar la solicitud de forma segura una y otra vez y todas las solicitudes conducen al mismo estado.

GET

- Idempotente
- Solo lectura. GET nunca cambia el estado del recurso en el lado del servidor. No debe tener efectos secundarios.
- La respuesta se puede almacenar en caché de forma segura.

  Ejemplos:

  `GET /employees` - Listas de todos los empleados

  `GET /employees/1`- Muestra los detalles del empleado 1

PUT

- Idempotente
- Se puede utilizar tanto para crear como para actualizar
- Comúnmente utilizado para actualizar (actualizaciones completas).

  Ejemplo:

  `PUT /employees/1- actualiza el empleado 1` (poco común: crea el empleado 1)

  Para usar `PUT` para crear, el cliente debe conocer la URL completa (incluida la ID) por adelantado. Eso es poco común ya que el servidor generalmente genera la identificación. Por lo tanto, `PUT` para crear se usa normalmente cuando solo hay un elemento y la URL no es ambigua.

  `PUT /employees/1/avatar` - Crea o actualiza el avatar del empleado 1.

  Solo hay un avatar para cada empleado.

  Incluya siempre la carga útil completa en la solicitud. Es todo o nada. `PUT` no está destinado a ser utilizado para actualizaciones parciales (ver `PATCH`).

POST

- No idempotente
- Utilizado para crear

Ejemplo:

`POST /employees` crean un nuevo empleado. La nueva URL devuelve al cliente la localización en el encabezado (por ejemplo, `Location: /employees/12`). Múltiples solicitudes `POST /employees` conducen a muchos empleados nuevos diferentes (es por eso que `POST` no es idempotente).

PATCH

- idempotente
- Se utiliza para actualizaciones parciales.

  Ejemplo:

  `PATCH /employees/1`- actualiza el empleado 1 con los campos contenidos en la carga útil. Los Los demás campos del empleado 1 no se modifican.

DELETE

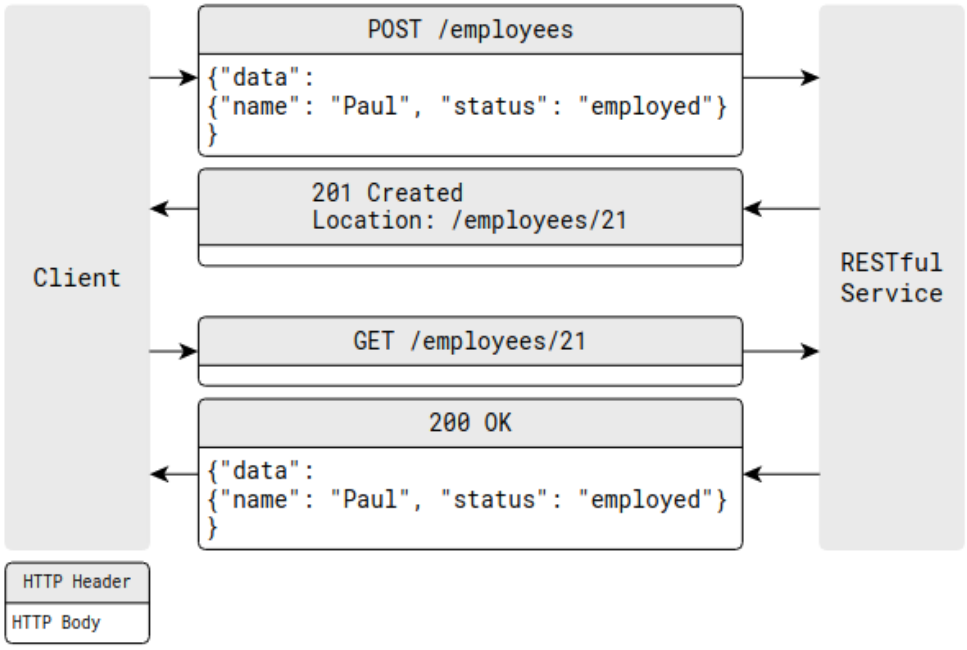
- idempotente
- Se utiliza para la eliminación.

  Ejemplo:

  `DELETE /employees/1` – Elimina al empleado 1

POST EN LA URL DE LA COLECCIÓN DE RECURSOS PARA CREAR UN NUEVO RECURSO

¿Cómo podría ser una interacción cliente-servidor para crear un nuevo recurso?

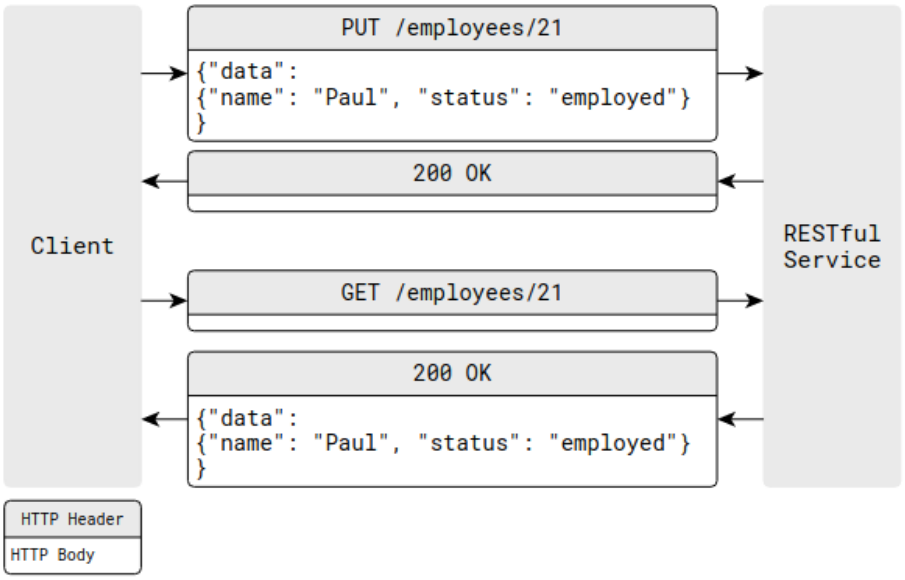


Use POST para crear un nuevo recurso

El cliente envía una solicitud `POST` a la URL de recopilación del recurso /employees. El cuerpo HTTP contiene los atributos del nuevo recurso "Paul".

El servicio web RESTful genera una `ID` para el nuevo empleado, crea el empleado en su modelo interno y envía una respuesta al cliente. Esta respuesta contiene el código de estado `201` (Creado) y un Location en el encabezado HTTP que indica la URL bajo la cual se puede acceder al recurso creado.

PUT EN LA URL DE RECURSO ÚNICO PARA ACTUALIZAR UN RECURSO



Use PUT para actualizar un recurso existente.

El cliente envía una solicitud `PUT` a la única URL del recurso `/employee/21`. El cuerpo HTTP de la solicitud PUT contiene todos los campos del empleado y cada campo se actualizará en el lado del servidor.

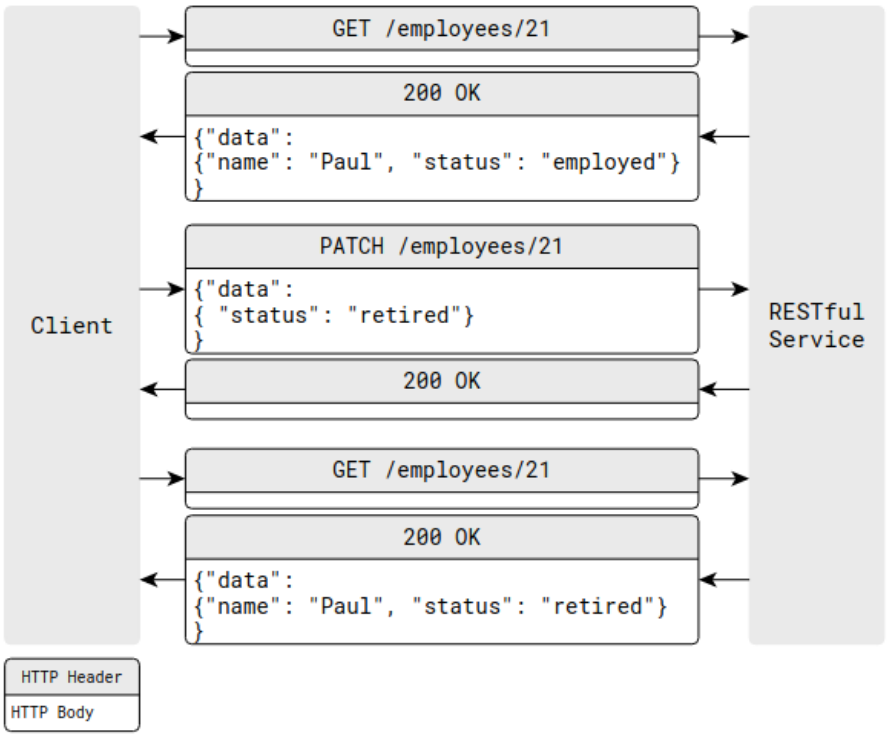
El servicio REST actualiza el name y status del empleado con el `ID 21` y confirma los cambios con el código de estado HTTP `200`.

USAR PATCH PARA ACTUALIZACIONES PARCIALES DE UN RECURSO

`PUT` no se utiliza para actualizaciones parciales. `PUT` solo debe usarse para reemplazos completos de un recurso. Enviar todos los campos cada vez (aunque sólo desee actualizar un solo campo) puede provocar sobreescrituras accidentales en caso de actualizaciones paralelas. Además, la implementación de la validación es difícil ya que debe admitir ambos casos de uso: crear (algunos campos no deben ser null) y actualizar (null valores para marcar campos que no deben actualizarse) al mismo tiempo. Por lo tanto, no se utiliza `PUT` y envía solo los campos que deben actualizarse. Los campos faltantes en la solicitud `PUT` deben tratarse como valores null y vaciar los campos de la base de datos o desencadenar errores de validación.

Para esto se utiliza `PATCH` para actualizaciones parciales. Envíe solo los campos que deben actualizarse. De esta manera, la carga útil de la solicitud es bastante sencilla, las actualizaciones paralelas de diferentes campos no anulan los campos no relacionados, la validación se vuelve más fácil, la semántica de los valores null no es ambigua (tanto para `PUT` como para `PATCH`) y ahorra ancho de banda.

Por ejemplo, la siguiente solicitud `PATCH` actualiza solo el campo ***status*** pero no el ***name***



Use PATCH y envíe solo los campos que desea actualizar.

OBTENCIÓN DE LOS DATOS REALES EN EL CAMPO *DATA*

Para recuperar datos de nuestra API, debemos realizar una petición GET a la ruta deseada. Usando una petición GET podemos buscar y filtrar resultados en base a nuestros criterios de búsqueda. Por ejemplo, para encontrar a todos los empleados en nuestra API, debemos hacer lo siguiente: `GET /employees`. Esto devuelve una lista de objetos en el campo ***data***.

```
{
   "data":[
      {
         "id":1,
         "name":"Larry"
```

```
        },
        {
            "id":2,
            "name":"Peter"
        }
    ]
}
```

¿Y cómo podemos filtrar nuestros resultados para que solo obtengamos la información que necesitamos y no todo el catálogo de empleados? Para filtrar los resultados para que solo muestre 1 resultado, debemos agregar otra barra y la identificación, o "id" del objeto que deseamos ver. Por ejemplo, para ver la información relacionada solo al primer empleado, debemos realizar la siguiente petitición `GET /employees/1`. Esto devuelve un único objeto en el campo data:

```
{
    "data":{
        "id":1,
        "name":"Larry"
    }
}
```

La carga útil de las solicitudes `PUT`, `POST` y `PATCH` también debe contener el campo data con el objeto real.

ventajas:

- Queda espacio para agregar metadatos (por ejemplo, para paginación, enlaces, advertencias de obsolescencia, mensajes de error)
- Consistencia
- Compatible con el estándar JSON:API

SE UTILIZA LA CADENA DE CONSULTA (?) PARA PARÁMETROS OPCIONALES Y COMPLEJOS

No hagas esto:

```
GET /employees
GET /externalEmployees
GET /internalEmployees
GET /internalAndSeniorEmployees
```

Mantenga sus URL simples y establezca una URL pequeña. Elija una URL base para su recurso y apeguese a ella. Mueva la complejidad o los parámetros opcionales a la cadena de consulta.

```
GET /employees?state=internal&title=senior
GET /employees?id=1,2
```

La forma de filtrado JSON:API es:

```
GET /employees?filter[state]=internal&filter[title]=senior
GET /employees?filter[id]=1,2
```

UTILIZAR CÓDIGOS DE ESTADO HTTP

El servicio web RESTful debe responder a la solicitud de un cliente con un código de respuesta de estado HTTP adecuado.

- 2xx – éxito – todo funcionó bien.
- 4xx – error del cliente – si el cliente hizo algo mal (por ejemplo, el cliente envía una solicitud no válida o no está autorizado)
- 5xx – error del servidor: fallas en el lado del servidor (errores al intentar procesar la solicitud, como fallas en la base de datos, los servicios dependientes no están disponibles, errores de programación o estados que no deberían ocurrir)

Considere los códigos de estado HTTP disponibles . Sin embargo, tenga en cuenta que usarlos todos podría resultar confuso para los usuarios de su API. Mantenga pequeño el conjunto de códigos de estado HTTP utilizados. Es común usar los siguientes códigos:

- 2xx: Éxito
  - 200 Bien
  - 201 Creado
- 3xx: redirigir
  - 301 Movido Permanentemente
  - 304 No modificado

- 4xx: error del cliente
  - 400 Petición Incorrecta
  - 401 No autorizado
  - 403 Prohibido
  - 404 No encontrado
  - 410 Ido
- 5xx: error del servidor
  - 500 Error interno de servidor

No abuses del 404 . Intenta ser más preciso. Si el recurso está disponible, pero el usuario no puede verlo, devuelva un 403 Prohibido. Si el recurso existió una vez pero ahora se elimina o desactiva, use 410.

PROPORCIONAR MENSAJES DE ERROR ÚTILES

Además de un código de estado apropiado, debe proporcionar una descripción útil y detallada del error en el cuerpo de su respuesta HTTP.

```
GET /employees?state=super
```

Respuesta:

```
// 400 Bad Request

{
   "errors":[
      {
         "status":400,
         "detail":"Invalid state. Valid values are 'internal' or
'external'",
         "code":352,
         "links":{
            "about":"http://www.domain.com/rest/errorcode/352"
         }
      }
   ]
}
```

USE CAMELCASE PARA NOMBRES DE ATRIBUTOS

No utilice guiones bajos (`year_of_birth`) ni mayúsculas (`YearOfBirth`). A menudo, su servicio web RESTful será consumido por un cliente escrito en JavaScript. Por lo general, el cliente convertirá la respuesta JSON en un objeto JavaScript (llamando a `var person = JSON.parse(response)`) y llamará a sus atributos. Por lo tanto, es una buena idea ceñirse a la convención de JavaScript, que hace que el código de JavaScript sea más legible e intuitivo

PROPORCIONAR PAGINACIÓN

Casi nunca es una buena idea devolver todos los recursos de su base de datos a la vez. En consecuencia, debe proporcionar un mecanismo de paginación. Dos enfoques populares son:

- Paginación basada en desplazamiento
- Paginación basada en conjunto de claves, también conocido como Token de continuación, también conocido como Cursor (recomendado)

CONTROL DE VERSIONES DE LA API RESTFUL

El control de versiones le permite lanzar cambios incompatibles y de última hora de su API en una nueva versión sin romper los clientes. Pueden seguir consumiendo la versión anterior. Los clientes pueden migrar a la nueva versión a su propia velocidad.

Este tema es muy discutido en la comunidad. Debe tener en cuenta que puede terminar construyendo y manteniendo diferentes versiones de una API durante mucho tiempo, lo cual es costoso.

Si está creando API internas, lo más probable es que conozca a todos sus clientes. Por lo tanto, realizar cambios importantes puede volver a ser una opción. Pero requerirá más comunicación y un despliegue coordinado.

Sin embargo, estos son los dos enfoques más populares para el control de versiones:

- Versionado a través de URL: `/v1/`
- Control de versiones a través del `Accept` encabezado HTTP:

  `Accept: application/vnd.myapi.v1+json` (Negociación de contenido)