OBLIGATORIO 1 - SmartHome

Diseño 2

Evidencia del diseño y especificación de la API

Link al repositorio:

https://github.com/IngSoft-DA2/281542-281835-281535

Integrantes:

- Matias Corvetto (281535)
- Alejo Fraga (281542)
- Sebastian Vega (281835)

Docentes:

- Daniel Acevedo
- Federico Gonzales

Septiembre 2024

Universidad ORT Uruguay

Facultad de Ingeniería

Abstract

Se ha desarrollado una API RESTful basada en los principios de la arquitectura REST, cumpliendo con las restricciones esenciales, como la interfaz uniforme, la separación entre cliente y servidor, la operación sin estado y una arquitectura de múltiples capas. Sin embargo, si bien éramos conscientes de la importancia del almacenamiento en caché para mejorar la eficiencia, no pudimos implementarlo debido a las limitaciones de tiempo. A pesar de ello, se ha implementado un sistema de autenticación basado en tokens para asegurar el acceso a los recursos. Los códigos de estado HTTP se utilizan para indicar los resultados de las solicitudes, y los recursos proporcionan distintos niveles de permisos para su acceso.

Índice

1. Criterios seguidos para asegurar que la API cumpla REST

REST establece cinco reglas arquitectónicas clave que identifican a una API como genuinamente RESTful. Estos pilares son:

1.1. Cacheable

El concepto de "cacheable" en el contexto de una API RESTful se refiere a la capacidad de almacenar las respuestas de las solicitudes en un sistema de caché, lo que permite a los clientes reutilizar esos datos en lugar de realizar una nueva solicitud al servidor. Esto puede mejorar significativamente la eficiencia y la velocidad de la aplicación, ya que reduce la carga en el servidor y minimiza la latencia en la recuperación de datos. En una API cacheable, las respuestas incluyen encabezados que indican si son aptas para el almacenamiento en caché y por cuánto tiempo pueden ser reutilizadas.
Sin embargo, en el desarrollo de nuestra API, no implementamos la funcionalidad de almacenamiento en caché debido a limitaciones de tiempo. A pesar de esto, somos conscientes de la importancia del caching para optimizar el rendimiento y la eficiencia de la API.

1.2. Stateless

El servidor opera bajo un modelo sin estado, lo que implica que no conserva información sobre las solicitudes HTTP previas del cliente; cada solicitud se procesa de manera independiente, como si fuera la primera interacción. En escenarios donde es esencial preservar el estado del usuario, como durante el inicio de sesión o al realizar operaciones que requieren permisos específicos, es crucial que cada solicitud del cliente incluya toda la información necesaria para su procesamiento, incluidas las credenciales de autenticación. En este contexto, utilizamos "tokens" para gestionar el proceso de autenticación.

1.3. Sistema de capas

El cliente se conecta a través de una interfaz, que puede ser directamente al servidor principal o mediante un intermediario. Para el usuario, la arquitectura subyacente es completamente transparente e irrelevante. Su principal preocupación es que la API REST funcione correctamente, sin tener que preocuparse por los detalles técnicos. En este contexto, estamos alineados con esta necesidad, ya que el cliente, representado por la herramienta Postman, no

tiene conocimiento sobre el desarrollo interno de la API REST ni sobre su funcionamiento; su única inquietud es que las solicitudes HTTP se manejen de manera adecuada.

1.4. Interfaz uniforme

La interfaz uniforme es uno de los principios fundamentales de la arquitectura REST, que establece un marco para la interacción entre clientes y servidores. Esta restricción promueve la creación de una interfaz estandarizada que gestione interacciones de manera coherente, asegurando que cada recurso sea accesible a través de una URL única y que se adhiera a un conjunto de buenas prácticas. En el diseño de APIs, se prioriza el uso de sustantivos en las URIs, en lugar de verbos, lo que refleja un enfoque centrado en los recursos en lugar de en las acciones. Esta filosofía se evidencia en las implementaciones donde las URIs identifican los recursos de forma clara y concisa, eliminando la necesidad de verbos y facilitando una comprensión más directa del propósito de cada recurso.
El uso de sustantivos es una buena práctica y altamente recomendada, la cual decidimos implementar

Recurso
/users
/homeowners
/auth
/devices
/deviceTypes
/companies
me/homes
/homes/{id}/members
/homes/{id}/hardwares
me/homes/{memberId}/permissions
/sensors
/sensors/{id}/windowDetection
/camera
/camera/{id}/PersonDetection
/camera/{id}/MovementDetection
/members/{memberId}/notifications

1.5. Cliente-Servidor

En nuestro obligatorio cumplimos con este principio porque hemos diseñado la API de manera que el cliente y el servidor puedan desarrollarse de forma independiente. El cliente solo interactúa con el servidor a través de las URIs para acceder o modificar los recursos, sin necesitar conocer la lógica interna del servidor. Esta separación permite que el cliente sea portátil, ya que puede ser implementado en diferentes entornos sin depender de detalles del servidor.

Por otro lado, el servidor se enfoca en el procesamiento de las solicitudes y el manejo de datos, sin interferir en la interfaz de usuario ni depender del estado del cliente, lo que le permite manejar múltiples solicitudes de manera eficiente. Al usar una comunicación estandarizada y mantener roles bien definidos, ambos componentes pueden evolucionar sin afectar al otro, asegurando escalabilidad y la integración con otros sistemas.

2. Autenticación en las requests

El mecanismo de autenticación en esta API se basa en el uso de tokens, transmitidos a través del header ‘Authorization’ en cada solicitud.
Cuando un usuario necesita autenticarse, realiza una petición POST al endpoint /auth, enviando las credenciales necesarias para identificarse, las cuales son email y contraseña. El backend, tras validar estas credenciales, genera un token en formato GUID. Este GUID, debido a su naturaleza, garantiza ser único para cada sesión del usuario, y se devuelve al cliente en la respuesta.

Este token GUID se convierte en la llave de acceso para el usuario en todas las siguientes solicitudes que requieran autenticación. Cada vez que el usuario quiera realizar una petición a un endpoint que requiera autenticación deberá incluir el token en el header ‘Authorization’ de la request. Este mecanismo permite que el token actúe como prueba de que el usuario ya ha sido autenticado previamente, evitando la necesidad de volver a ingresar las credenciales en cada solicitud.

Cada vez que se necesite validar que el usuario esté autenticado, se consultará el token del header de la request por medio de un filtro de autenticación (AuthenticationFilter). Este filtro tiene la responsabilidad de validar la autenticidad del token incluido en el header. Para ello, el filtro verifica si el token corresponde a un usuario autenticado en el sistema. Si el token es válido, se permite que la petición pase al método correspondiente para ser procesada. Es importante aclarar que en algunos casos, cuando el usuario que realiza la request debe tener permisos especiales, también se debe pasar por el filtro de autorización (AuthorizationFilter), el cual verifica que el usuario tenga los permisos necesarios para realizar las acciones solicitadas.

3. Códigos de estado

3.1. 200:

El código de estado HTTP 200 indica que la solicitud realizada por el cliente ha sido procesada correctamente por el servidor. Este código se utiliza para confirmar que el recurso solicitado está disponible y se ha enviado al cliente sin errores. En términos generales, un código 200 refleja que la operación se llevó a cabo con éxito y que no hubo problemas en la comunicación entre el cliente y el servidor.

```
return new ObjectResult(new
{
    InnerCode = "OK",
    Message = "Operation was successfully performed",
    Data = getUsersResponse
})
{
    StatusCode = (int)HttpStatusCode.OK
};
```

3.2. 201:

El código de estado HTTP 201, conocido como "Created", indica que una solicitud ha sido completada exitosamente y que, como resultado, se ha creado un nuevo recurso en el servidor. Este código se utiliza comúnmente en respuestas a solicitudes POST, donde se envían datos para crear un nuevo recurso, como un nuevo registro en una base de datos.

```
return new ObjectResult(new
{
    InnerCode = "Created",
    Message = "User was successfully created",
    Data = userData
})
{
    StatusCode = (int)HttpStatusCode.Created
};
```

3.3. 400:

El código de estado HTTP 400, denominado "Bad Request", se utiliza para señalar que la solicitud enviada por el cliente al servidor no puede ser procesada debido a

errores en la sintaxis o en los parámetros de la misma. Este código indica que el servidor no pudo interpretar correctamente la solicitud debido a problemas como datos mal formateados, parámetros obligatorios ausentes o valores no válidos.

```
{
    typeof(NullReferenceException),
    (Exception exception) =>
    {
        var concreteException = (NullReferenceException)exception;
        return new ObjectResult(new
        {
            InnerCode = "BadRequest",
            Message = "Request is missing",
        })
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
},
```

```
{
    typeof(ArgumentOutOfRangeException),
    (Exception exception) =>
    {
        var concreteException = (ArgumentOutOfRangeException)exception;
        return new ObjectResult(new
        {
            InnerCode = "BadRequest",
            Message = "Argument is not in a valid range",
            Details = concreteException.Message
        })
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
},
```

```
{
    typeof(ArgumentNullException),
    (Exception exception) =>
    {
        var concreteException = (ArgumentNullException)exception;
        return new ObjectResult(new
        {
            InnerCode = "BadRequest",
            Message = "Argument cannot be null or empty",
            Details = concreteException.Message
        })
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
},
```

```
{
    typeof(ArgumentException),
    (Exception exception) =>
    {
        var concreteException = (ArgumentException)exception;
        return new ObjectResult(new
        {
            InnerCode = "BadRequest",
            Message = "Argument is invalid",
            Details = concreteException.Message
        })
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
},
```

3.4. 401:

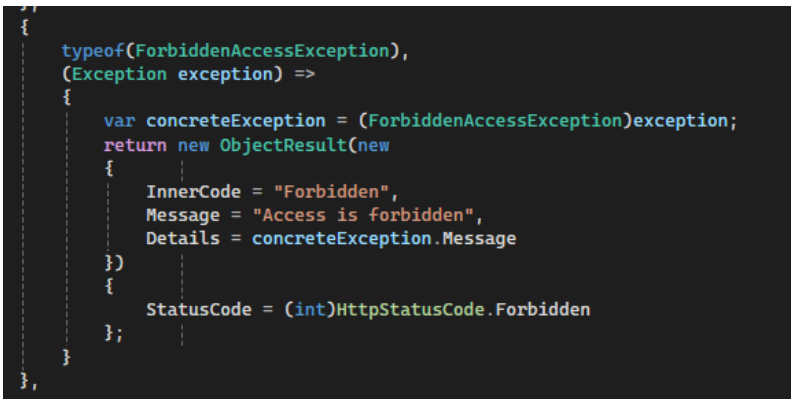
El código de estado HTTP 401, denominado "Unauthorized", indica que la solicitud del cliente no ha sido aplicada porque no se han proporcionado credenciales de autenticación válidas. Este código se utiliza cuando el servidor requiere autenticación para acceder al recurso solicitado, y el cliente no ha enviado las credenciales necesarias o éstas son incorrectas.

```
{
    typeof(UnauthorizedAccessException),
    (Exception exception) =>
    {
        var concreteException = (UnauthorizedAccessException)exception;
        return new ObjectResult(new
        {
            InnerCode = "Unauthenticated",
            Message = "You are not authenticated",
            Details = concreteException.Message
        })
        {
            StatusCode = (int)HttpStatusCode.Unauthorized
        };
    }
},
```

3.5. 403:

El código de estado HTTP 403, conocido como "Forbidden", indica que el servidor ha comprendido la solicitud del cliente, pero se niega a autorizarla. Esto significa que,

aunque las credenciales de autenticación pueden ser correctas (si se requieren), el cliente no tiene permisos suficientes para acceder al recurso solicitado.



3.6. 404:

El código de estado HTTP 404, denominado "Not Found", indica que el servidor no ha podido encontrar el recurso solicitado. Esto ocurre cuando la URL proporcionada por el cliente no corresponde a ningún recurso disponible en el servidor, o cuando el recurso ha sido eliminado o movido a otra ubicación.

3.7. 409:

El código de estado HTTP 409, denominado "Conflict", indica que la solicitud no pudo completarse debido a un conflicto con el estado actual del recurso en el servidor. Este código se utiliza cuando el servidor determina que procesar la solicitud podría generar inconsistencias, colisiones o violaciones de restricciones, como en casos de control de versiones o de reglas de negocio que no permiten el cambio solicitado.

```
{
    typeof(InvalidOperationException),
    (Exception exception) =>
    {
        var concreteException = (InvalidOperationException)exception;
        return new ObjectResult(new
        {
            InnerCode = "Conflict",
            Message = "A conflict occurred with existing resources",
            Details = concreteException.Message
        })
        {
            StatusCode = (int)HttpStatusCode.Conflict
        };
    }
},
```

3.8. 500:

El código de estado HTTP 500, denominado "Internal Server Error", indica que el servidor ha encontrado una condición inesperada que le impide completar la solicitud del cliente. Este código es una respuesta genérica que se utiliza cuando no se puede especificar el problema exacto, lo que significa que hay un error en el servidor que no ha sido manejado adecuadamente.

```
public void OnException(ExceptionContext context)
{
    var response = _errors.GetValueOrDefault(context.Exception.GetType());

    if (response == null)
    {
        context.Result = new ObjectResult(new
        {
            InnerCode = "InternalError",
            Message = "There was an error when processing the request"
        })
        {
            StatusCode = (int)HttpStatusCode.InternalServerError
        };
        return;
    }

    context.Result = response(context.Exception);
}
```

4. Recursos

4.1. URL Base

➢ localhost:5000

4.2. Descripción de cada Recurso

➢ La descripción de cada recurso se encuentra en este Google Sheets: DocuAPI.
Además se encuentra en formato .xlsx en la carpeta Documentación.

5. Referencias bibliográficas

[1] Brian Mulloy, Web API Design Crafting - Interfaces that Developers Love

6. Declaracion de autoria

Nosotros, Sebastian Vega, Alejo Fraga y Matias Corvetto, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

• La obra fue producida en su totalidad mientras realizábamos el Primer Obligatorio de Diseño de Aplicaciones 2;

• Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;

• Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;

• En la obra, hemos acusado recibo de las ayudas recibidas;

• Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;

• Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Matias Corvetto

8/10/24