

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1 - Diseño de Aplicaciones 2

Evidencia del diseño y especificación de la API

 **GitHub** <https://github.com/IngSoft-DA2-2023-2/266628-255981-271568>

Federico Rodriguez - 255981

Santiago Salinas - 266628

Manuel Morandi - 271568

Tutores: Daniel Acevedo

Rafael Alonso

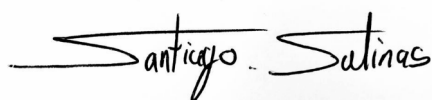
Franco Galeano

Septiembre 2023

Declaración de autoría

Federico Rodriguez, Santiago Salinas y Manuel Morandi declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

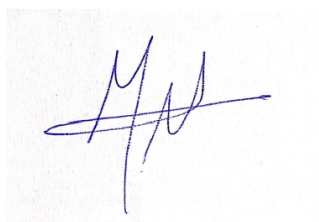
- La obra fue producida en su totalidad mientras construimos el primer obligatorio de Diseño de Aplicaciones 2
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra 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



Santiago Salinas 26/9/2023



Federico Rodriguez 26/9/2023



Manuel Morandi 26/9/2023

Abstract

Documento describiendo la API conteniendo:

- Discusión de los criterios seguidos para asegurar que la API cumple con los criterios REST.
- Descripción del mecanismo de autenticación de requests.
- Descripción general de códigos de error (1xx, 2xx, 4xx, 3xx, 5xx).
- Descripción de los resources de la API.
 - URL base.
 - Para cada resource describir:
 - Resource.
 - Description.
 - Endpoints (Verbo + URI).
 - Parameters.
 - Responses (para todos los códigos de estado).
 - Headers

Abstract	2
Discusión de los criterios REST	4
Mecanismo de autenticación de requests	5
Descripción general de códigos de error	5
Descripción de los resources de la API	7
Admin	7
Get Users	7
Get User by Id	7
Get Purchases	8
Register User	8
Update User	9
Delete User	10
Brand	11
Get Brands	11
Cart	11
Get Cart Price and Info	11
Buy Cart	13
Category	14
Get Categories	14
Colour	14
Get Colours	15
LogIn	15
LogIn	15
Product	16
Get Products	16
Get Specific Product	16
Filter Products	17
Add Product	17
Modify Product	18
Delete Product	20
Purchase	20
Get Specific Purchase	20
Get History	21
User	21
Get Specific User	22
Register User	22
Modify Existing User	23
Delete Existing User	24

Discusión de los criterios REST

Antes de hablar sobre nuestros criterios, debemos definir que es una API REST. Una API es un intermediario, una interfaz que conecta 2 aplicaciones. Expone información y funcionalidades para permitir la comunicación entre ambas. Para que una API sea RESTful, debe cumplir los siguientes criterios:

- 1) Cliente-Servidor: El sistema debe seguir esta arquitectura, dividiendo el sistema en dos grandes roles. Los clientes son todos aquellos que quieren acceder a la información que almacena el servidor. El servidor tiene la capacidad de almacenamiento y procesamiento para poder atender a los request de los clientes.
- 2) Stateless: El servidor no tiene estados, ya que guardar los estados de todos los clientes lo saturaría. En su lugar, las solicitudes del cliente al servidor contienen toda la información necesaria para que el servidor pueda comprenderla y procesarla.
- 3) Cacheable: El servidor puede indicarle al cliente que almacene información que necesitará con frecuencia para fácil acceso.
- 4) Dividido en capas: El sistema se divide en capas, logrando así independencia entre cliente y servidor. Un cliente no debe saber si está conectado al servidor final o a un intermediario.
- 5) Interfaz uniforme: Se siguen un conjunto de convenciones para simplificar el diseño y uso, haciendo que sea más interoperable. Por ejemplo, el sistema debe estar basado en recursos, a los que se accede a través de URIs, utilizando verbos estandarizados (get, post, put, delete).
- 6) Code on Demand: El servidor puede transferir lógica para que sea ejecutada por un cliente, expandiendo funcionalidades.

Nuestra API busca seguir estos criterios, consiguiendo así que sea ordenada y cumpla ciertos estándares. Todas las acciones que se pueden llevar a cabo se hacen con los 4 verbos básicos, de tal manera que su funcionamiento sea lo más

claro y evidente posible, contribuyendo así a obtener una interfaz uniforme. Se implementa un servidor para que sea usado por clientes (nota: los clientes serán aquellas aplicaciones que hagan uso de nuestra API), y esta aplicación se divide en capas para mayor independencia. Toda información es almacenada en la base de datos, absolutamente nada la aplicación servidor. No obstante, el cliente puede almacenar ciertos datos relevantes y frecuentes, como pueden ser los tokens.

Mecanismo de autenticación de requests

Las requests son realizadas por usuarios. Estos usuarios deben ser autenticados para demostrar que son quien dicen ser. Para lograr esto, cada usuario cuenta con un token propio, generado a partir de su mail. Se trata de un string único, que ningún otro usuario podrá tener. Es verdad que la generación de tokens es simple e insegura, ya que se genera directamente con los datos del usuario. Se podría buscar un método más seguro para generarlos, o utilizar una librería que de ello se ocupe, pero escapa a los intereses del curso.

Al loguearse, este token se comenzará a enviar en los headers de las requests que el usuario haga (y que puedan llegar a necesitarlo). Antes de llevar a cabo las operaciones solicitadas, un filtro de autenticación se ocupará de verificar el token. En caso de encontrar incongruencias (el token no es válido, el usuario no está logueado, etc.), la acción no se llevará a cabo y se le informará al usuario. Este filtro se explica en “Descripción de diseño”.

Descripción general de códigos de error

Tras procesar cada request, el servidor responde con un código. El objetivo de este es informar sobre el resultado de la operación. Dependiendo del código devuelto podemos saber si la operación fue exitosa, si hubo un error o si no se pudo llevar a cabo por alguna razón particular.

Se puede entender que hay 5 tipos de códigos, y se dividen por la letra con la que empiezan. Los que empiezan por 1 cumplen un rol informativo, mientras que los que comienzan con 2 denotan una operación exitosa. Por su parte, los que inician con un 3 informan una redirección. Finalmente, los últimos dos grupos expresan errores, siendo los 4xx aquellos errores del cliente y 5xx los del servidor.

En nuestro caso, los códigos que usamos fueron:

- 200: Es un código general de éxito. Fue usado para aquellas operaciones que, tras salir bien, no brindan ninguna información extra que pueda resultar relevante.
- 201: Indica que algo se creó correctamente. Se utiliza para indicar que los objetos solicitados fueron agregados al sistema.
- 204: Representa NoContent. Esto implica que la acción se llevó con éxito y que el usuario no necesita redirigirse y puede seguir operando en la página actual. Lo usamos cuando borramos algo.
- 400: BadRequest, el servidor no puede procesar la solicitud por algo que parece ser error del cliente.
- 401: Unauthorized, el usuario no cumple los requisitos para llevar a cabo esta acción.
- 403: Forbidden, el usuario no tiene el rol pertinente para llevar a cabo la acción.
- 404: Implica que no se encontró el recurso solicitado.
- 500: Internal Server Error, es un error que puede saltar con cualquier request hecha al servidor. Salta si ocurre un error inesperado que no es culpa del usuario de la Api, y es respuesta del Exceptionfilter que atrapa lo que los try y catch dedicados no cubren.

Descripción de los resources de la API

Admin

Este endpoint se utiliza para acceder a todas las operaciones que son exclusivas para administradores. Todas las acciones que pueden llevarse a cabo están protegidas por filtro de autorización y de autenticación, por lo que en los headers se deberá enviar el token bajo la clave de Auth. Si se intenta ejecutar una operación de admin, y el token proveído no corresponde a un usuario con el rol de Admin, la respuesta será un código de estado 403, o 401 si no se recibe ningún token.

Get Users

- Descripción: Retorna una lista que contiene todos los usuarios registrados.
- Verbo: Get
- URI: /Admin/Users
- Headers: Auth → string
- Body:
- Ejemplo: Get/Admin/Users
- Response:
 - 200: Success, se retorna la lista de usuarios deseada.

Get User by Id

- Descripción: Retorna el User que posee el id indicado.
- Verbo: Get
- URI: /Admin/Users/{id}
- Headers: Auth → string
- Body:
- Ejemplo: Get/Admin/Users/1
- Response:
 - 200: Success, se retorna el User solicitado.

- 404: NotFound, el User solicitado no existe o no fue encontrado.

Get Purchases

- Descripción: Retorna la lista de todas las compras realizadas por todos los usuarios.
- Verbo: Get
- URI: /Admin/Purchases
- Headers: Auth → string
- Body:
- Ejemplo: Get/Admin/Purchases
- Response:
 - 200: Success, se retorna la lista solicitada.

Register User

- Descripción: Crea un nuevo User y lo registra en el sistema.
- Verbo: Post
- URI: /Admin/Users
- Headers: Auth → string
- Body:

```
{
  "email": "string",
  "address": "string",
  "password": "string",
  "roles": [
    {
      "name": "string"
    }
  ]
}
```
- Ejemplo: Post/Admin/Users

```
{
  "email": "example@gmail.com",
```

```
"address": "Cuareim 1451",
"password": "password",
"roles": [
  {
    "name": "Admin"
  },
  {
    "name": "Customer"
  }
]
```

- Response:
 - 200: Ok, el User fue creado correctamente.
 - 400: BadRequest, el User no ha podido ser registrado.

Update User

- Descripción: Modifica los datos del usuario seleccionado.
- Verbo: Put
- URI: /Admin/Users/{id}
- Headers: Auth → string
- Body:

```
{
  "id": int,
  "email": "string",
  "address": "string",
  "password": "string",
  "roles": [
    {
      "name": "string"
    }
  ]
}
```

- Ejemplo: Put/Admin/Users/1

```
{  
  "id": 1,  
  "email": "example@gmail.com",  
  "address": "Cuareim 1451",  
  "password": "password",  
  "roles": [  
    {  
      "name": "Admin"  
    },  
    {  
      "name": "Customer"  
    }  
  ]  
}
```

- Response:
 - 204: NoContent, la info del User se actualizó correctamente.
 - 400: BadRequest, la id del User a modificar no coincide con la registrada.
 - 404: NotFound, el User no existe o no se encontró.

Delete User

- Descripción: Elimina un User del sistema.
- Verbo: Delete
- URI: /Admin/Users/{id}
- Headers: Auth → string
- Body:
- Ejemplo: Delete/Admin/Users/1
- Response:
 - 204: NoContent, el User fue eliminado.
 - 404: NotFound, el User no existe o no se encontró.

Brand

Este endpoint maneja todas las marcas que fueron agregadas a la base de datos del sistema. Todos los productos ofrecidos son de una de estas marcas. A día de hoy, no se contempla agregar, modificar ni eliminar marcas si no es por base de datos, por lo que solo tendremos operación Get.

Get Brands

- Descripción: Retorna la lista de las marcas disponibles.
- Verbo: Get
- URI: /Brand
- Headers:
- Body:
- Ejemplo: Get/Brand
- Response:
 - 200: Success, se retorna la lista de marcas disponibles.

Cart

Permite conocer el valor y descuento aplicado, si corresponde, de una sarta de productos. Para enviar el carrito a procesar hacemos uso del verbo POST, para poder enviar un body json. Posteriormente, leemos la respuesta.

Get Cart Price and Info

- Descripción: Devuelve datos de los productos en el carrito, el precio sin descuento, y el precio con descuento si aplica, y qué descuento.

- Verbo: Post
- URI: /Cart
- Headers:
- Body:

```
{
  "products": [
    {
      "id": 0,
      "quantity": 0
    }
  ]
}
```

- Ejemplo: Post/User

```
{
  "products": [
    {
      "id": 1,
      "quantity": 3
    },
    {
      "id": 2,
      "quantity": 1
    }
  ]
}
```

- Response:
 - 201: Created, se retorna la información del carrito.
 - 400: BadRequest, en caso de que el carro esté vacío, el id es incorrecto, o cantidades menores o iguales a cero.

Buy Cart

- Descripción: Toma un carrito y lo guarda como una compra a nombre del usuario.
- Verbo: Post
- URI: /Cart/buy
- Headers: Auth → string
- Body:

```
{  
  "products": [  
    {  
      "id": 0,  
      "quantity": 0  
    }  
  ]  
}
```

- Ejemplo: Post/User

```
{  
  "products": [  
    {  
      "id": 1,  
      "quantity": 3  
    },  
    {  
      "id": 2,  
      "quantity": 1  
    }  
  ]  
}
```

- Response:
 - 200: Success, la compra fue realizada con éxito.
 - 400: BadRequest, en caso de que el carro esté vacío, el id es incorrecto, o cantidades menores o iguales a cero.

- 403: Forbidden, el usuario no tiene el rol Customer.

Category

Se contemplan prendas de ciertas categorías, las cuales no pueden ser modificadas ni eliminadas, tampoco se pueden crear nuevas. Se trabaja solo con Get.

Get Categories

- Descripción: Retorna la lista de las categorías registradas.
- Verbo: Get
- URI: /Category
- Headers:
- Body:
- Ejemplo: Get/Category
- Response:
 - 200: Success, se retorna la lista de categorías disponibles.

Colour

El sistema acepta usar colores de una lista de algunos de ellos. Este endpoint referencia esa lista. Como en el caso de los colores, no se contempla agregar más marcas, así como eliminar o editar las ya existentes.

Get Colours

- Descripción: Retorna la lista de los colores disponibles.
- Verbo: Get
- URI: /Colour
- Headers:
- Body:
- Ejemplo: Get/Colour
- Response:
 - 200: Success, se retorna la lista de colores deseada.

Login

Este endpoint tiene como único objetivo permitir a los usuarios loguearse. Para cumplir este cometido, se hace uso de una acción de Post, que recibe las credenciales del usuario y las verifica.

Login

- Descripción: Verifica las credenciales del usuario para loguearse.
- Verbo: Post
- URI: /Login
- Headers:
- Body:

```
{  
  "email": "string",  
  "password": "string",  
}
```
- Ejemplo: Post/Login

```
{
```



```
"email": "example@gmail.com",  
"password": "password",  
}
```

- Response:
 - 200: Success, el usuario inició sesión.
 - 400: BadRequest, las credenciales son incorrectas.

Product

La aplicación se basa en ofrecer productos de ropa. Este recurso es manejado por este endpoint. Estos productos se pueden obtener (y filtrar), crear, actualizar y eliminar.

Get Products

- Descripción: Se retornan todos los productos registrados.
- Verbo: Get
- URI: /Product
- Headers:
- Body:
- Ejemplo: Get/Product
- Response:
 - 200: Success, se retornan los productos.

Get Specific Product

- Descripción: Retorna el producto con ese id específico.
- Verbo: Get
- URI: /Product/{id}
- Headers:
- Body:

- Ejemplo: Get/Product/1
- Response:
 - 200: Success, se retorna el Product solicitado.
 - 404: NotFound, el Product solicitado no existe o no fue encontrado.

Filter Products

- Descripción: Retorna el o los Product que cumplan los criterios de filtrado.
- Verbo: Get
- URI: /Product/filtered?category={category}&brand={brand}&name={name}
- Headers:
- Body:
- Ejemplo: Get/Product/filtered?category="Caps"&brand="Nike"
- Response:
 - 200: Success, se retornan todos los productos que cumplen las condiciones.

Add Product

- Descripción: Crea un nuevo producto y lo registra en el sistema.
- Verbo: Post
- URI: /Product
- Headers: Auth → string
- Body:

```
{
  "id": 0,
  "name": "string",
  "priceUYU": 0,
  "description": "string",
  "brand": {
    "name": "string"
  },
  "category": {
    "name": "string"
  }
}
```

```
    },  
    "colours": [  
      {  
        "name": "string"  
      }  
    ]  
  }  
}
```

- Ejemplo: Post/Product

```
{  
  "id": 1,  
  "name": "example",  
  "priceUYU": 150,  
  "description": "This is an Example",  
  "brand": {  
    "name": "Nike"  
  },  
  "category": {  
    "name": "Shorts"  
  },  
  "colours": [  
    {  
      "name": "Black"  
    }  
  ]  
}
```

- Response:

- 200: OK, el Product fue creado correctamente.
- 400: BadRequest, el Product no ha podido ser registrado.

Modify Product

- Descripción: Actualiza el producto solicitado.
- Verbo: Put

- URI: /Product/{id}
- Headers: Auth → string
- Body:

```
{
  "id": 0,
  "name": "string",
  "priceUYU": 0,
  "description": "string",
  "brand": {
    "name": "string"
  },
  "category": {
    "name": "string"
  },
  "colours": [
    {
      "name": "string"
    }
  ]
}
```
- Ejemplo: Put/Product/1

```
{
  "id": 1,
  "name": "example",
  "priceUYU": 150,
  "description": "This is an Example",
  "brand": {
    "name": "Nike"
  },
  "category": {
    "name": "Shorts"
  },
  "colours": [
```

```
{  
  "name": "Black"  
}  
]  
}
```

- Response:
 - 204: Ok, la info del Product se actualizó correctamente.
 - 400: BadRequest, la id del Product a modificar no coincide con la registrada. O los nuevos datos son inválidos.
 - 404: NotFound, el Product no existe o no se encontró.

Delete Product

- Descripción: Elimina el producto dado.
- Verbo: Delete
- URI: /Product/{id}
- Headers: Auth → string
- Body:
- Ejemplo: Delete/Product/20
- Response:
 - 204: Ok, el Product fue eliminado.
 - 404: NotFound, el Product no existe o no se encontró.

Purchase

Cuando un usuario confirma la compra de un carrito, esta es almacenada. Este endpoint refiere a esta lista de compras

Get Specific Purchase

- Descripción: Se retorna la compra de ese id.

- Verbo: Get
- URI: /Purchase/{id}
- Headers: Auth → string
- Body:
- Ejemplo: Get/Purchase/4
- Response:
 - 200: Success, se retornan los productos.
 - 403: Forbidden, no se puede acceder a esa compra ya que fue realizada por otro usuario.
 - 404: NotFound, no se encontró una compra con es id.

Get History

- Descripción: Retorna las compras realizadas por el usuario que la solicita.
- Verbo: Get
- URI: /Purchase/history
- Headers: Auth → string
- Body:
- Ejemplo: Get/Purchase/history?email="example@gmail.com"
- Response:
 - 200: Success, se retorna la lista en cuestión.

User

Users son todos aquellos individuos que estén registrados en el sistema. Sobre ellos se pueden hacer todas las operaciones usuales, pero este endpoint abarca sólo aquellas que pueden realizar los que tienen el rol Customer. El resto de operaciones, las que son restringidas, se llevan a cabo desde endpoint Admin. Lo que resulta en un Customer solo pudiendo acceder y modificar a sus propios datos. Puede obtener su id haciendo un get/user genérico con el token auth correspondiente.

Get Specific User

- Descripción: Retorna la información del .
- Verbo: Get
- URI: /User
- Headers: Auth → string
- Body:
- Ejemplo: Get/User
- Response:
 - 200: Success, se retorna el User solicitado.
 - 404: NotFound, el User solicitado no existe o no fue encontrado.

Register User

- Descripción: Crea un nuevo User con el rol Customer y lo registra en el sistema.
- Verbo: Post
- URI: /User
- Headers:
- Body:

```
{  
  "email": "string",  
  "password": "string",  
  "token": "string",  
  "address": "string"  
}
```
- Ejemplo: Post/User

```
{  
  "email": "example@gmail.com",  
  "password": "password",  
  "token": "tokenexample@gmail.comsecure",  
  "address": "Cuareim 1541"  
}
```

- Response:
 - 200: OK, el User fue creado correctamente.
 - 400: BadRequest, el User no ha podido ser registrado.

Modify Existing User

- Descripción: Actualiza los datos del usuario seleccionado.
- Verbo: Put
- URI: /User/{id}
- Headers: Auth → string
- Body:

```
{  
  "id": 0,  
  "email": "string",  
  "password": "string",  
  "token": "string",  
  "address": "string"  
}
```
- Ejemplo: Put/User/20

```
{  
  "id": 20,  
  "email": "example@gmail.com",  
  "password": "password",  
  "token": "tokenexample@gmail.comsecure",  
  "address": "Cuareim 1541"  
}
```
- Response:
 - 204: NoContent, la info del User se actualizó correctamente.
 - 400: BadRequest, la id del User a modificar no coincide con la registrada.
 - 404: NotFound, el User no existe o no se encontró.

Delete Existing User

- Descripción: Elimina el User que realiza la operación de la base de datos.
- Verbo: Delete
- URI: /User
- Headers: Auth → string
- Body:
- Ejemplo: Delete/User
- Response:
 - 204: NoContent, la info del User se actualizó correctamente.
 - 404: NotFound, el User no existe o no se encontró.