

Universidad ORT Uruguay

Facultad de Ingeniería

# **Diseño de aplicaciones 2**

## **Descripción del diseño**

### **Obligatorio 1**

Emiliano Barboza (147067)  
Mauricio Dalgalarondo (189280)

Docentes:  
Ignacio Valle  
Daniel Acevedo

Entregado como requisito de la materia Diseño de  
aplicaciones 2  
<https://github.com/ORT-DA2/barboza-dalgalarondo>

15 de octubre de 2020

# Declaraciones de autoría

Nosotros, Emiliano Barboza y Mauricio Dalgarrondo, 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 la materia
- 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.

## **Resumen**

Este documento del obligatorio 1 plantea poner en práctica los conocimientos vistos hasta el momento en el curso del punto de vista del diseño de la api:

1. Arquitecturas web
2. Rest Api
3. Diseño de web apis

# Palabras claves

- Api rest - es un estilo de arquitectura de software que se utiliza para describir cualquier interfaz entre diferentes sistemas que utilice HTTP para comunicarse.
- HTTP - Es un protocolo de comunicación que permite el envío de información.
- Bearer - es un tipo de string ilegible al que se le llama token, que tiene la particularidad de ser formado de la siguiente forma "Bearer token".

# Índice general

<b>1. Descripción del Proyecto</b>	<b>6</b>
1.1. Introducción y Objetivos . . . . .	6
1.2. Alcance de la Aplicación . . . . .	6
<b>2. Criterios seguidos para asegurar la API</b>	<b>8</b>
2.1. Buenas prácticas en el uso de la semántica . . . . .	8
2.1.1. Pensar en sustantivos y no en verbos . . . . .	8
2.1.2. Estabilidad de endpoints . . . . .	8
2.1.3. Estructura HTTP . . . . .	8
2.1.4. Versionado de apis . . . . .	9
2.1.5. Principio KISS . . . . .	11
2.1.6. Uso de modelos para comunicar datos . . . . .	11
2.2. Buenas prácticas del lado técnico . . . . .	11
2.2.1. Uso de modelos para comunicar datos . . . . .	11
2.2.2. Validar todo . . . . .	12
2.2.3. Mantener los controllers livianos . . . . .	12
2.2.4. Uso de async/await siempre que sea posible . . . . .	13
2.3. Mecanismo de autenticación de requests . . . . .	13
<b>3. Descripción de los recursos de la API</b>	<b>15</b>
3.1. Resumen api . . . . .	15
3.2. Bookings api . . . . .	16
3.2.1. UpdateBookingStatus . . . . .	16
3.2.2. GetBookingStatus . . . . .	16
3.3. Lodgments . . . . .	17
3.3.1. ActivateLodgment . . . . .	17
3.3.2. DeactivateLodgment . . . . .	17
3.3.3. AddBookingAtLodgment . . . . .	18
3.4. Regions . . . . .	18
3.4.1. GetRegions . . . . .	18
3.4.2. AddSpotToRegion . . . . .	19
3.4.3. GetSpotsByRegion . . . . .	19
3.5. Sessions . . . . .	20
3.5.1. Login . . . . .	20
3.5.2. Logout . . . . .	20
3.6. Spots . . . . .	21

3.6.1.	AddLodgmentToSpot . . . . .	21
3.6.2.	DeleteLodgmentInSpot . . . . .	21
3.6.3.	GetLodgmentsInSpot . . . . .	22
3.7.	Users . . . . .	23
3.7.1.	GetUsers . . . . .	23
3.7.2.	AddUser . . . . .	23
3.7.3.	DeleteUser . . . . .	24
3.7.4.	UpdateUser . . . . .	24

# 1. Descripción del Proyecto

## 1.1. Introducción y Objetivos

El objetivo de este proyecto es crear una api rest para dar soporte a la marca Uruguay Natural.

Las tecnologías involucradas son:

- Web Services (REST API) - Se expondrá las apis en un IIS.
- Microsoft Visual Studio Code (lenguaje C)
- Microsoft SQL Server Express 2017
- Postman
- NET Core SDK 3.1 / ASP.NET Core 3
- Entity Framework Core 3.1.3

## 1.2. Alcance de la Aplicación

El nuevo sistema, deberá dar acceso a 3 tipos de usuarios posibles en la plataforma (turistas, administradores y super administradores). Para los cuales se listan los requerimientos funcionales para los mismos.

Requerimientos funcionales:

- RF1 - Búsqueda de puntos turísticos por región y por categoría:.
- RF2 - Elegir un punto turístico y realizar una búsqueda de hospedajes.
- RF3 - Dado un hospedaje, realizar una reserva.

Requerimientos no funcionales

- RNF1 - Independencia de librerías.

Se debe diseñar la solución que al modificar el código fuente minimice el impacto del cambio en los componentes físicos de la solución.

Cada paquete lógico debe ser implementado en un assembly independiente.

- RNF2 - Acceso a las funcionalidades mediante HTTP

Acceso mediante web service.

- RNF3 - Persistencia en base de datos.

El diseño debe contemplar el modelado de una solución de persistencia adecuada para el problema utilizando Entity Framework (Code First).

Se espera que como parte de la entrega se incluya dos respaldos de la base de datos: uno vacío y otro con datos de prueba.

Se debe entregar el archivo .bak y también el script .sql para ambas bases de datos.

- RNF4 - Mantenibilidad

Estar en un repositorio Git.

Haber sido escrito utilizando TDD

Se debe utilizar el framework Moq para realizar los Mocking.

- RNF5 - Control de versiones

GitFlow



## 2. Criterios seguidos para asegurar la API

### 2.1. Buenas prácticas en el uso de la semántica

#### 2.1.1. Pensar en sustantivos y no en verbos

La idea general de las restful apis está basada en el concepto de usar sustantivos a la hora de darle nombre a nuestros endpoints. Estas uando se leen, deben generar el entendimiento que son acciones. Ejmplos

- GET /spots, obtener la lista de spots
- GET /spots/10, obtener el spot 10
- POST /spots, crear un spot

No es aceptable algo de este estilo, debido a que al leerlo son verbos:

- GET /getAllSpots, obtener la lista de spots
- GET /getSpotById/10, obtener el spot 10
- POST /addSpot, crear un spot

#### 2.1.2. Estabilidad de endpoints

Dada la naturaleza de los endpoints, estos comunican nuestro sistema con el mundo, es por eso que una vez publicados un cambio en ellos puede romper funcionalidad de los clientes que consumen nuestra api, por lo cual es importante definir las rutas que se expondrán para no alterarlas.

#### 2.1.3. Estructura HTTP

Es importante seguir buenas prácticas a la hora de crear nuestras apis, por lo cual se siguieron las recomendaciones de Mozilla developers como guía.

## Uso de verbos

- GET, para obtener datos, listas, elementos específicos.
- PUT/POST, para cambiar o crear datos.

## Respuestas HTTP

Siguiendo las guías se manejan los códigos de estado de HTTP. En particular, se decidió usar la mínima cantidad posible de forma de agregar nuevos, solamente de ser necesarios.

- 200 - Ok - es usado para devolver que la petición fue exitosa.
- 500 - Internal Server Error - estos son utilizados para evidencia errores del lado del servidor.
- 401 - Unauthorized - estos son usados para mostrar que un usuario necesita credenciales para el uso de un recurso.
- 400 - Bad Request - sirven para mostrar que los datos que envió el cliente no son correctos.

### 2.1.4. Versionado de apis

Para el versionado se tomó como base el artículo y se eligió soportar dos tipos de versionado.

- **Query String-Based Versioning**, el mismo nos permite pasar el versionado de la api que deseamos utilizar utilizando parámetros en la query.

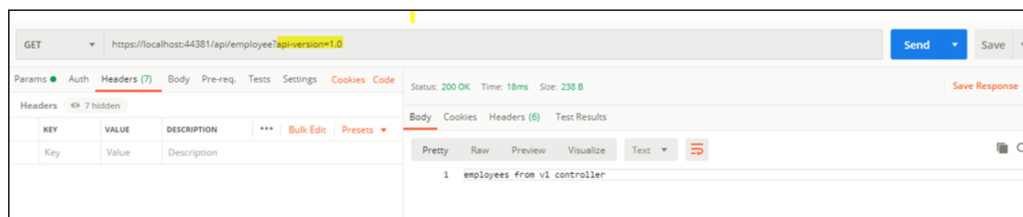


Figura 2.1: Versionado por parámetros en la query

- **HTTP Header-Based Versioning**, el mismo nos permite pasar el versionado de la api que deseamos utilizar utilizando los headers. Para su correcto uso es necesario pasar el valor **Api-Version**.

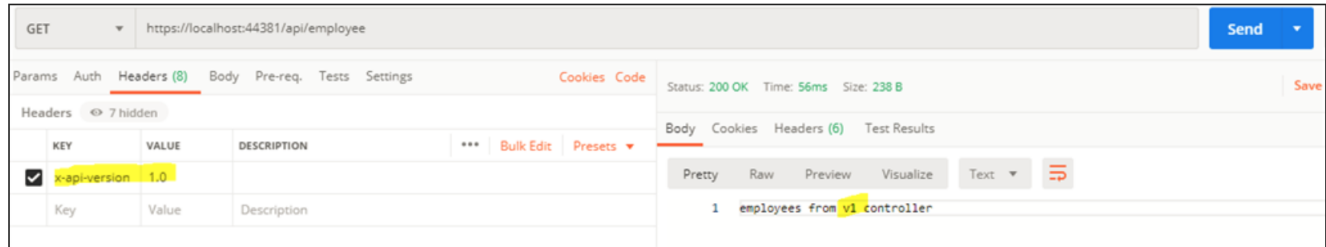


Figura 2.2: Versionado por header

- En caso que no se provea una versión, se toma como default la versión 1.0.
- En caso que una versión solicitada no sea compatible con la versión soportada por el endpoint, este devolverá en los headers de respuesta las versiones que soporta.

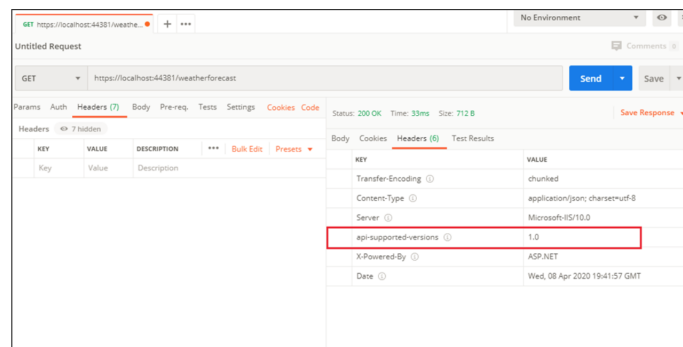


Figura 2.3: Versiones disponibles para la api

- Llevando esto a código, el versionado en sí, se colocó en una clase base (`BaseApiController`) la cual permite que todos los controllers tengan esa versión por defecto.

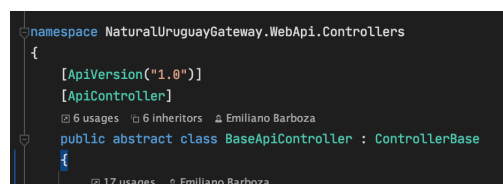


Figura 2.4: Versionado en controller

A su vez las versiones se pueden extender, deprecarse y otras funciones más las cuales sugerimos leer el siguiente artículo.

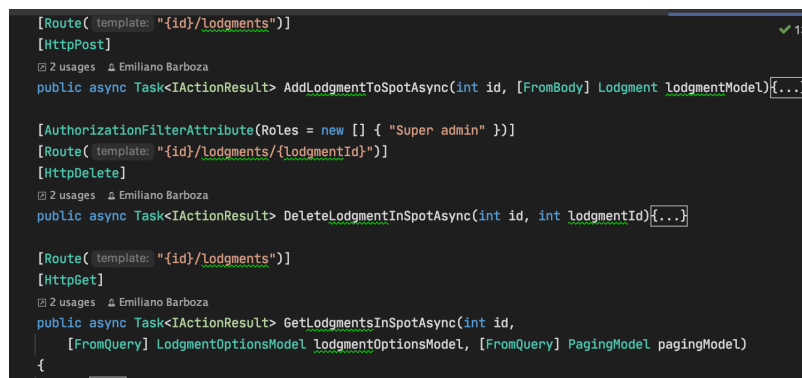
### 2.1.5. Principio KISS

Este principio nos guió junto a TDD a exponer solamente los recursos/endpoints que necesitábamos. El principio explica que solamente se expone lo que vamos a necesitar y nada más, lo cual nos ayudó también a la hora de ir exponiendo los códigos HTTP previamente descriptos.

### 2.1.6. Uso de modelos para comunicar datos

La comunicación con la api se puede dar tanto con modelos, entidades o datos primitivos.

La idea aquí es utilizar los distintos mecanismos para poder enviar datos desde el cliente al servidor.

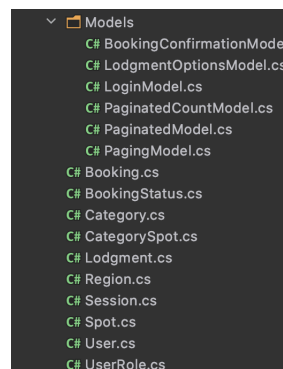


```
[Route( template: "{id}/Lodgments")]
[HttpPost]
public async Task<IActionResult> AddLodgmentToSpotAsync(int id, [FromBody] Lodgment lodgmentModel){...}

[AuthorizationFilterAttribute(Roles = new [] { "Super admin" })]
[Route( template: "{id}/Lodgments/{lodgmentId}")]
[HttpDelete]
public async Task<IActionResult> DeleteLodgmentInSpotAsync(int id, int lodgmentId){...}

[Route( template: "{id}/Lodgments")]
[HttpGet]
public async Task<IActionResult> GetLodgmentsInSpotAsync(int id,
    [FromQuery] LodgmentOptionsModel lodgmentOptionsModel, [FromQuery] PagingModel pagingModel)
{
    ...
}
```

Figura 2.5: Comunicación de modelos con el controller



- Models
  - C# BookingConfirmationModel
  - C# LodgmentOptionsModel.cs
  - C# LoginModel.cs
  - C# PaginatedCountModel.cs
  - C# PaginatedModel.cs
  - C# PagingModel.cs
  - C# Booking.cs
  - C# BookingStatus.cs
  - C# Category.cs
  - C# CategorySpot.cs
  - C# Lodgment.cs
  - C# Region.cs
  - C# Session.cs
  - C# Spot.cs
  - C# User.cs
  - C# UserRole.cs

Figura 2.6: Modelos utilizados

## 2.2. Buenas prácticas del lado técnico

### 2.2.1. Uso de modelos para comunicar datos

Como se comentó anteriormente, el uso de modelos u entidades para la comunicación es esencial para la correcta comunicación.

### 2.2.2. Validar todo

Por defecto se asume todo dato debe ser validado. Es por eso que se exponen distintos mecanismos de validación.

1. DataAnnotations, estos los usamos a nivel de entidades/modelos para aprovechar al máximo los beneficios que nos brinda el Framework.  
Aquí aprovechamos que el mismo controle si nos envían los mails con formato correcto, o simplemente datos requeridos.
2. A nivel de servicios, se hacen las validaciones lógicas, por ejemplo saber si un id de una región existe.

### 2.2.3. Mantener los controllers livianos

Los controllers deben ser lo más livianos posibles, estos deben hacer una sola cosa y es la de ser un pasa manos al servicio que será el que sabe cómo realizar la acción solicitada.

Los controllers deben únicamente pasar los datos a los servicios y no tener lógica en el mismo.

Lo único aceptable es el try catch para poder devolver correctamente una excepción.

```
[Route("template: \"{id}/lodgments")]
[HttpGet]
@ 2 usages  Emiliano Barboza
public async Task<IActionResult> GetLodgmentsInSpotAsync(int id,
    [FromQuery] LodgmentOptionsModel lodgmentOptionsModel, [FromQuery] PagingModel pagingModel)
{
    try
    {
        PaginatedModel<Lodgment> paginatedLodgments = await service.GetLodgmentsInSpotAsync(id, lodgmentOptionsModel, pagingModel);
        return Ok(paginatedLodgments);
    }
    catch (Exception ex)
    {
        var exceptionResponse :ObjectResult = GetExceptionResponse(ex, Request);
        return exceptionResponse;
    }
}
```

Figura 2.7: Controller liviano

#### 2.2.4. Uso de async/await siempre que sea posible

El uso de este tipo de programación favorece el uso de recursos de hardware, permitiendo liberar threads del thread pool. De no usarlos un thread permanecerá en uso hasta el fin de la acción, tomemos en cuenta que determinadas acciones pueden no estar ocurriendo en el servidor, por lo que genera un delay y un desperdicio de recursos del sistema. Para tomar esta decisión nos basamos en el artículo donde se describen las diferencias entra async vs sync en ASP.NET.

### 2.3. Mecanismo de autenticación de requests

El mecanismo de autenticación consta de dos partes:

1. El SessionsController, es el punto de entrada para poder identificarnos como usuarios del sistema (login), este se comunica con la base de datos utilizando los paquetes que se muestran en la imagen a continuación.

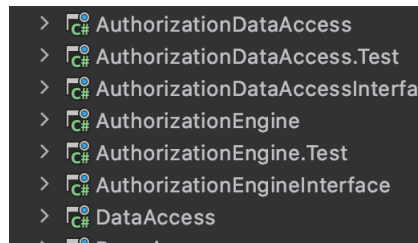


Figura 2.8: Paquetes para autenticación

Y finalmente se guardará una sesión en la base de datos la cuál identifica que somos usuarios del sistema y que nos hemos logueado para hacer uso del mismo.

2. Luego tenemos un filtro (AuthorizationFilterAttribute) que será el encargado de asegurar cada controller o simplemente endpoint que necesitemos asegurar.

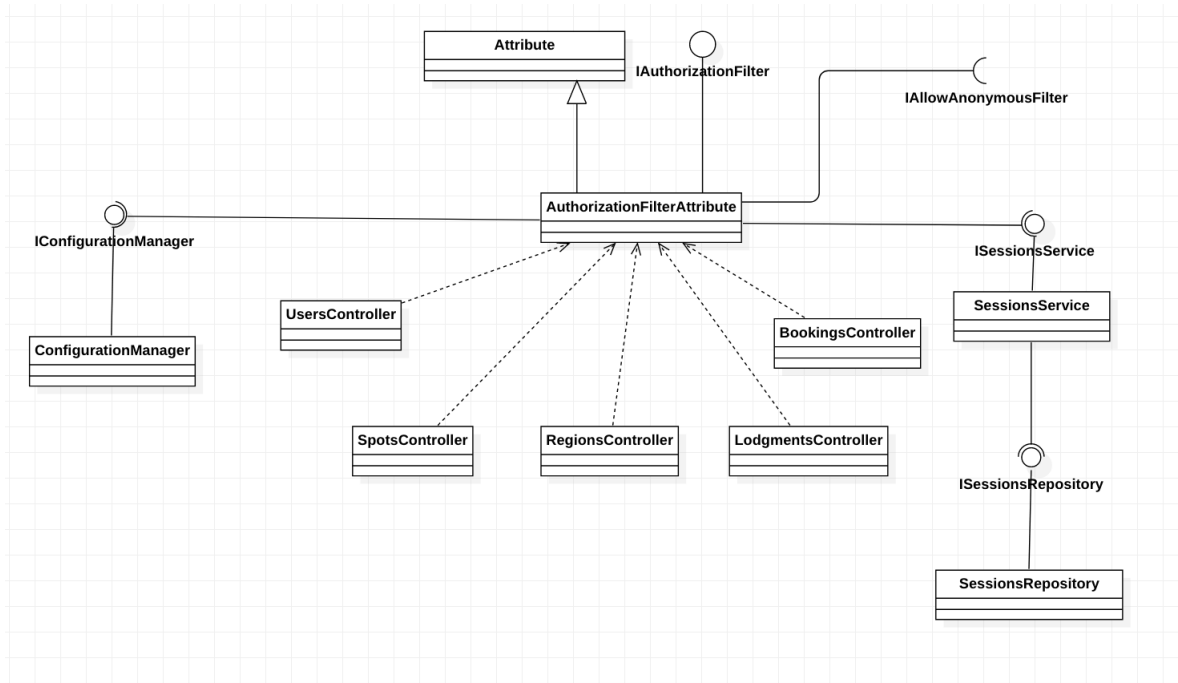


Figura 2.9: Filtro AuthorizationFilterAttribute

El filtro intercepta los requests y mediante un proceso valida lo siguiente:

- Authorization header este presente.
- El schema utilizado sea del tipo Bearer.
- Que haya una sesión disponible para ese token.
- Que el role del usuario tenga acceso al recurso.

En caso de fallar en alguno de estos, se devolverá una excepción según los criterios tomados en secciones anteriores.

## 3. Descripción de los recursos de la API

### 3.1. Resumen api

**NaturalUruguayGateway Api** 1 **OAS3**  
/swagger/natural-uruguay-gateway/swagger.json

<b>Bookings</b>		▼
PUT	/bookings/{id}/status	
GET	/bookings/{confirmationCode}/status	
<b>Lodgments</b>		▼
POST	/lodgments/{id}/activate	
DELETE	/lodgments/{id}/deactivate	
POST	/lodgments/{id}/bookings	
<b>Regions</b>		▼
GET	/regions	
POST	/regions/{id}/spots	
GET	/regions/{id}/spots	
<b>Sessions</b>		▼
POST	/sessions	
DELETE	/sessions	
<b>Spots</b>		▼
POST	/spots/{id}/lodgments	
GET	/spots/{id}/lodgments	
DELETE	/spots/{id}/lodgments/{lodgmentId}	
<b>Users</b>		▼
GET	/users	
POST	/users	
DELETE	/users/{id}	
PUT	/users/{id}	

Figura 3.1: Resumen de apis



## 3.2. Bookings api

### 3.2.1. UpdateBookingStatus

PUT

/bookings/{id}/status

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small>	
integer(int32)	id
(path)	

Request body

application/json-patch+json

Example Value | Schema

```
{
  "id": 0,
  "name": "string",
  "description": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.2: UpdateBookingStatus

### 3.2.2. GetBookingStatus

GET

/bookings/{confirmationCode}/status

Try it out

Parameters

Name	Description
<b>confirmationCode</b> <small>required</small>	
string	confirmationCode
(path)	

Responses

Code	Description	Links
200	Success	No links

Figura 3.3: GetBookingStatus

# 3.3. Lodgments

## 3.3.1. ActivateLodgment

POST

/lodgments/{id}/activate

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="id"/>

Responses

Code	Description	Links
200	Success	No links

Figura 3.4: ActivateLodgment

## 3.3.2. DeactivateLodgment

DELETE

/lodgments/{id}/deactivate

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="id"/>

Responses

Code	Description	Links
200	Success	No links

Figura 3.5: DeactivateLodgment

### 3.3.3. AddBookingAtLodgment

POST

/lodgments/{id}/bookings

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small>	
integer(int32)	id
(path)	

Request body

application/json-patch+json

Example Value

Schema

```
{
  "name": "string",
  "lastName": "string",
  "email": "user@example.com",
  "lodgmentOptions": {
    "checkIn": 0,
    "checkOut": 0,
    "amountOfAdults": 0,
    "amountOfUnderAge": 0,
    "amountOfBabies": 0
  }
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.6: AddBookingAtLodgment

## 3.4. Regions

### 3.4.1. GetRegions

GET

/regions

Try it out

Parameters

Name	Description
Direction	Direction
string	
(query)	
Limit	Limit
integer(int32)	
(query)	
Offset	Offset
integer(int32)	
(query)	
Order	Order
string	
(query)	
FilterBy	FilterBy
string	
(query)	

Responses

Code	Description	Links
200	Success	No links

Figura 3.7: GetRegions

### 3.4.2. AddSpotToRegion

POST

/regions/{id}/spots

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(int32) (path)	id

Request body

application/json-patch+json

Example Value

Schema

```
{
  "id": 0,
  "name": "string",
  "thumbnail": "string",
  "description": "string",
  "lodgments": [
    {
      "id": 0,
      "name": "string",
      "amountOfStays": 0,
      "address": "string",
      "images": [
        "string"
      ],
      "price": 0,
      "totalPrice": 0,
      "description": "string",
      "capacity": 0,
      "phoneNumber": "string",
      "contactInformation": "string",
      "isDeleted": true,
      "spotId": 0
    }
  ],
  "regionId": 0
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.8: AddSpotToRegion

### 3.4.3. GetSpotsByRegion

GET

/regions/{id}/spots

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(int32) (path)	id
Direction string (query)	Direction
Limit integer(int32) (query)	Limit
Offset integer(int32) (query)	Offset
Order string (query)	Order
FilterBy string (query)	FilterBy

Responses

Code	Description	Links
200	Success	No links

Figura 3.9: GetSpotsByRegion

# 3.5. Sessions

## 3.5.1. Login

POST /sessions

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value | Schema

```
{  "email": "user@example.com",  "password": "string"}  
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.10: Login

## 3.5.2. Logout

DELETE /sessions

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	Success	No links

Figura 3.11: Logout

# 3.6. Spots

## 3.6.1. AddLodgmentToSpot

POST

/spots/{id}/lodgments

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="id"/>

Request body

application/json-patch+json

Example Value

Schema

```
{
  "id": 0,
  "name": "string",
  "amountOfStairs": 0,
  "address": "string",
  "images": [
    "string"
  ],
  "price": 0,
  "totalPrice": 0,
  "description": "string",
  "capacity": 0,
  "phoneNumber": "string",
  "contactInformation": "string",
  "isDeleted": true,
  "spotId": 0
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.12: AddLodgmentToSpot

## 3.6.2. DeleteLodgmentInSpot

DELETE

/spots/{id}/lodgments/{lodgmentId}

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="id"/>
<b>lodgmentId</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="lodgmentId"/>

Responses

Code	Description	Links
200	Success	No links

Figura 3.13: DeleteLodgmentInSpot

### 3.6.3. GetLodgmentsInSpot

GET

/spots/{id}/lodgments

Try it out

Parameters

Name	Description
<b>id</b> <small>required</small> integer(\$int32) (path)	<input type="text" value="id"/>
CheckIn integer(\$int64) (query)	<input type="text" value="CheckIn"/>
CheckOut integer(\$int64) (query)	<input type="text" value="CheckOut"/>
AmountOfAdults integer(\$int32) (query)	<input type="text" value="AmountOfAdults"/>
AmountOfUnderAge integer(\$int32) (query)	<input type="text" value="AmountOfUnderAge"/>
AmountOfBabies integer(\$int32) (query)	<input type="text" value="AmountOfBabies"/>
Direction string (query)	<input type="text" value="Direction"/>
Limit integer(\$int32) (query)	<input type="text" value="Limit"/>
Offset integer(\$int32) (query)	<input type="text" value="Offset"/>
Order string (query)	<input type="text" value="Order"/>
FilterBy string	<input type="text" value="FilterBy"/>

Figura 3.14: GetLodgmentsInSpot

# 3.7. Users

## 3.7.1. GetUsers

GET

/users

Try it out

Parameters

Name	Description
Direction string (query)	<input type="text" value="Direction"/>
Limit integer(\$int32) (query)	<input type="text" value="Limit"/>
Offset integer(\$int32) (query)	<input type="text" value="Offset"/>
Order string (query)	<input type="text" value="Order"/>
FilterBy string (query)	<input type="text" value="FilterBy"/>

Responses

Code	Description	Links
200	Success	No links

Figura 3.15: GetUsers

## 3.7.2. AddUser

POST

/users

Try it out

Parameters

No parameters

Request body

application/json-patch+json

Example Value

Schema

```
{
  "id": 0,
  "roleId": 0,
  "name": "string",
  "email": "user@example.com",
  "password": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.16: AddUser



### 3.7.3. DeleteUser

DELETE

/users/{id}

Try it out

Parameters

Name	Description
id * required	
integer(\$int32)	
(path)	

Responses

Code	Description	Links
200	Success	No links

Figura 3.17: DeleteUser

### 3.7.4. UpdateUser

PUT

/users/{id}

Try it out

Parameters

Name	Description
id * required	
integer(\$int32)	
(path)	

Request body

application/json-patch+json

Example Value

Schema

```
{
  "id": 0,
  "roleId": 0,
  "name": "string",
  "email": "user@example.com",
  "password": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

Figura 3.18: UpdateUser

# Bibliografía

- [1] Universidad ORT Uruguay. (2013) Documento 302 - Facultad de Ingeniería. [Online]. Available: <http://www.ort.edu.uy/fi/pdf/documento302facultaddeingenieria.pdf>
- [2] Anónimo. (2012) Conceptos básicos de HTTP. [Online]. Available: [https://developer.mozilla.org/es/docs/Web/HTTP/Basics\\_of\\_HTTP](https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP)
- [3] Joydip Kanjilal. (2020) Versionados en .NET CORE. [Online]. Available: <https://www.infoworld.com/article/3562355/how-to-use-api-versioning-in-aspnet-core.html>
- [4] Sumit Kharche. (2018) Versionados en .NET CORE. [Online]. Available: <https://www.c-sharpcorner.com/article/api-versioning-in-asp-net-core-web-api/>
- [5] Lucia Armentano. (2017) Buenas prácticas para el Diseño de una API RESTful Pragmática. [Online]. Available: <https://elbauldelprogramador.com/buenas-practicas-para-el-diseno-de-una-api-restful-pragmatica/>
- [6] Rubén de Celis Hernández. (2017) Buenas Prácticas para el Buen Diseño de una API RESTful. [Online]. Available: <https://mascandobits.es/programacion/buenas-practicas-para-el-buen-diseno-de-una-api-restful/>
- [7] TutorialTeacher. Web API Filters. [Online]. Available: <https://www.tutorialsteacher.com/webapi/web-api-filters>
- [8] SomosTechies. (2016) Agregando detalles de auditoria automáticamente en ASP.net MVC - ActionFilter. [Online]. Available: <https://somostechies.com/agregando-detalles-de-auditoria-automaticamente-en-asp-net/>
- [9] Javi Giner. (2015) Añadir Summarys como Description en Help Pages de ASP .NET Web API. [Online]. Available: <http://javiginer.com/summary-description-help-pages-asp-net-web-api/>
- [10] Microsoft. (2020) Creating Help Pages for ASP.NET Web API. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/creating-api-help-pages>
- [11] Desconocido. How To Do Custom Validation Using ValidationAttribute of ASP.NET MVC. [Online]. Available: <http://dotnetmentors.com/mvc/how-to-do-custom-validation-using-validationattribute-of-aspnet-mvc.aspx>

- [12] Refactoring GURU. (2020) Proxy. [Online]. Available: <https://refactoring.guru/design-patterns/proxy>
- [13] Kabukicho, Tokyo. (2019) Long Story Short: Async/Await Best Practices in .NET. [Online]. Available: [https://medium.com/@deep\\_blue\\_day/long-story-short-async-await-best-practices-in-net-1f39d7d84050](https://medium.com/@deep_blue_day/long-story-short-async-await-best-practices-in-net-1f39d7d84050)
- [14] Microsoft. (2020) Asynchronous programming with async and await. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/>