Universitat Oberta de Catalunya

uoc.edu

Desarrollo de arquitectura .NET Core 3.1 + Angular 9 en entorno macOS

Borja Montes Prado
Grado de Ingeniería Informática
TFG Desarrollo web

Profesor colaborador: Gregorio Robles Martínez
Profesor/a responsable: Santi Caballé Llobet

12/06/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Desarrollo de arquitectura .NET Core 3.1 + Angular 9 en entorno macOS*
Nombre del autor:	*Borja Montes Prado*
Nombre del consultor/a:	*Gregorio Robles Martínez*
Nombre del PRA:	*Santi Caballé Llobet*
Fecha de entrega (mm/aaaa):	06/2020
Titulación::	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Desarrollo Web*
Idioma del trabajo:	Español
Palabras clave	*APIWeb, Angular, .NET Core*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

La finalidad de este proyecto es construir una API REST utilizando la última versión de .NET conocida (.NET Core 3.1) y que su información sea consumida por una aplicación web desarrollada en Angular 9. Una de las novedades más significativas de .NET Core frente a versiones anteriores de los *frameworks web* de Microsoft, es su capacidad real de funcionar en múltiples plataformas, incluyendo Linux o macOS. En este caso, partiendo de una arquitectura genérica, pero actual, la programación de la totalidad del proyecto se ha realizado en un equipo que corre bajo el sistema operativo macOS Mojave.

Por otro lado, se ha seleccionado Angular 9, con la finalidad de adquirir conocimientos de una tecnología ampliamente demandada en el mercado laboral y en la que el alumno carecía de experiencia previa.

Finalmente, destacar que la intención del proyecto ha sido tratar de forma transversal el mayor número posible de disciplinas, lenguajes y herramientas, frente al desarrollo de una aplicación convencional que cubra unos determinados requisitos.

Abstract (in English, 250 words or less):

The purpose of this project is to build a REST API using the latest known version of .NET (.NET Core 3.1) with its information consumed by a web application developed in Angular 9. One of the most significant novelties of .NET Core compared to previous versions of Microsoft web frameworks, is the real ability to work on multiple platforms, including Linux or macOS. In this case, based on a generic but current architecture, the entire project has been programmed on a computer running under the macOS Mojave operating system.

On the other hand, Angular 9 has been selected, in order to acquire the knowledge of a technology widely demanded in the labor market and in which the student lacked previous experience, stops.

Finally, it should be noted that the intention of the project has been to deal transversally with as many disciplines, languages and tools as possible, as opposed to developing a conventional application that covers certain requirements.

Índice

1. Introducción

1.1 Contexto y justificación del Trabajo

En la actualidad, con el fin de ahorrar en costes, resolver problemas de implementación y mejorar las operaciones de DevOps y producción, las empresas están optando por implementaciones con arquitectura de microservicios mediante el uso de contenedores. Este enfoque está basado en una colección de servicios que permiten ser desarrollados, probados, implementados y versionados independientemente.

En el caso de Microsoft, la implementación de esta arquitectura se realiza con Azure Kubernetes Service y Azure Service Fabric, productos recién creados en colaboración con líderes del sector cómo Docker, Mesosphere y Kubernetes, que permiten la utilización de contenedores Windows y Linux. El principal objetivo de estos productos es ayudar a las empresas a implementar aplicaciones a velocidad y escala de nube, independientemente de la plataforma o herramientas elegidas.

Este enfoque, está alineado tanto con la necesidad actual de consumo de servicios de forma instantánea e independiente del lugar, como con la flexibilidad y escalabilidad necesarias para el desarrollo o actualización de aplicaciones.

1.2 Objetivos del Trabajo

El objetivo final de este trabajo es implementar desde cero operaciones CRUD con una arquitectura basada en .NET Core para el *back end* y en Angular 9 para el *front end*. El enfoque, por cuestiones de tiempo y espacio, se realizará de manera horizontal, con el fin de presentar todos los elementos necesarios de la arquitectura de una manera accesible y visual.

Asimismo, se pretende profundizar en distintas competencias transversales adquiridas durante la realización del Grado de Ingeniería Informática (programación orientada a objetos, diseño y creación de base de datos, gestión de proyectos, ingeniería del *software*, etc.) y ampliar el conocimiento de nuevas tecnologías no utilizadas con anterioridad (p.e. Angular 9).

Se definen los siguientes subobjetivos:

- Generar documentación en API propia
- Consumir a APIs ajenas
- Integrar base de datos en API
- Desarrollar operaciones CRUD en aplicación web consumiendo los servicios de la API
- Crear contenedor/es Docker.
- Adquirir experiencia en desarrollos .NET bajo el sistema operativo macOS

1.3 Enfoque y método seguido

Para la consecución de los objetivos planteados es necesario:

- Crear de documentación en API propia

Para la documentación de la API se utilizará Swagger UI, por ser un estándar actual de la industria que permite consumir su documentación de forma sencilla a través de un interfaz web.

- Acceder y consumir APIs ajenas

Se pretende consumir información de APIs ajenas con las que nutrir de datos a nuestra aplicación.

- Integrar base de datos en API REST

Se valorarán opciones de bases de datos tanto relacionales como no relacionales.

- Definir tecnología web para mostrar información de los servicios ofrecidos por la API REST

Durante la etapa de análisis y diseño se valorarán las tecnologías a utilizar para el desarrollo de la interfaz de usuario. No es objetivo de este proyecto profundizar en este aspecto, aunque sí parece necesario presentar una introducción de la interacción entre *backend* y *frontend* de la arquitectura elegida.

- Crear de contenedores Docker

Se creará uno o varios contenedores, en función de la necesidad, con el fin de mostrar su utilidad en la fase de implantación y pruebas.

- Elegir Framework .NET

Se seleccionará el framework más apropiado. Las opciones iniciales son :

.NET Framework 4.x
.NET Core

1.4 Planificación del Trabajo

La planificación se ha definido en base a las fechas de los diferentes entregables en lugar de a las fases del proyecto, por ser fechas de obligado cumplimiento. De este modo, han quedado definidas cuatro etapas:

- PEC 1
- PEC 2
- PEC 3
- ENTREGA FINAL

A pesar de esto, en el diagrama de Gantt se ha indicado las fases del proyecto que se realizan en cada etapa. De este modo, la relación entre etapas y fases es la siguiente:

ETAPA	FASE	FECHA
PEC 1	REQUERIMIENTOS	13/03/2020
PEC 2	ANÁLISIS-DISEÑO-DESARROLLO	10/04/2020
PEC 3	IMPLEMENTACIÓN	29/05/2020
ENTREGA FINAL		12/06/2020

Diagrama de Gantt

Para asegurar las entregas y la realización de todas las tareas correspondientes a este proyecto se ha definido el siguiente diagrama de Gantt:

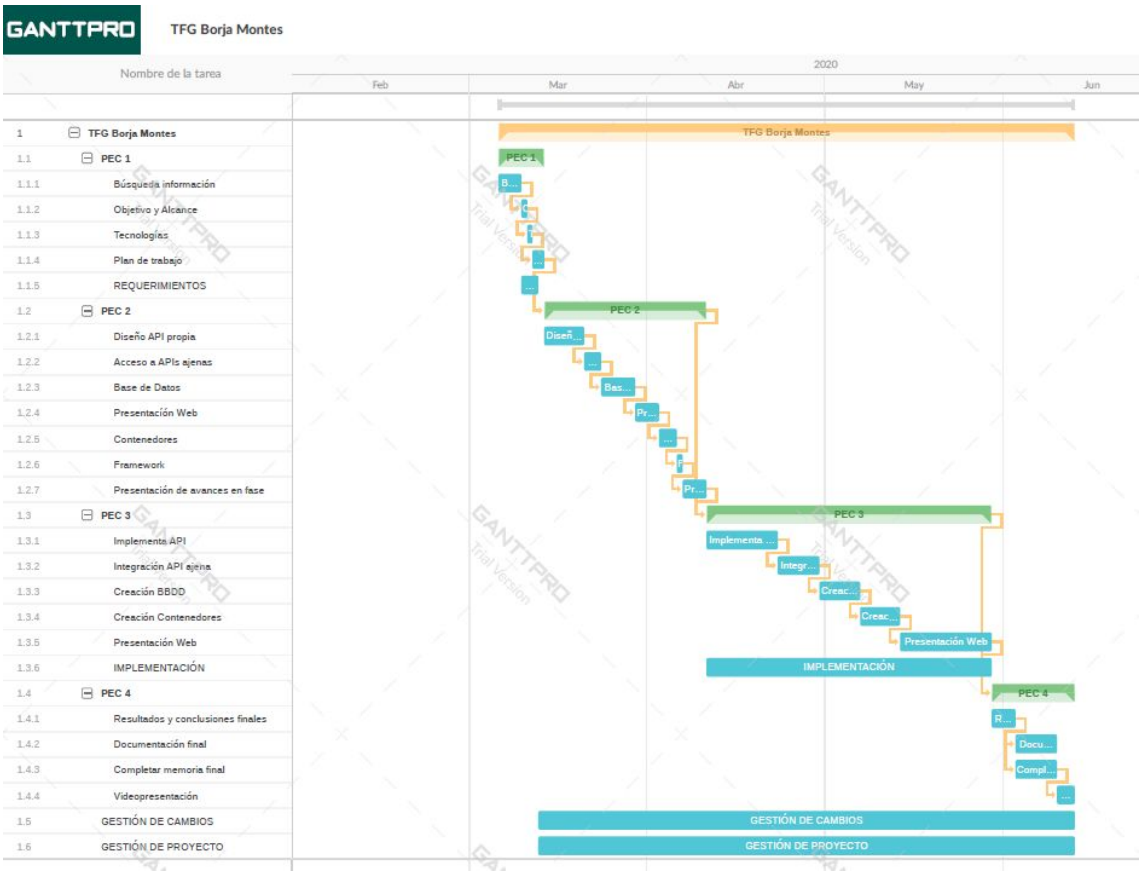


Figura 1: Diagrama de Gantt con planificación temporal del proyecto a realizar

Seguimiento y control

Además de los puntos de control establecidos en los entregables, se ha establecido el email como canal de comunicación con el tutor así como herramienta de seguimiento y control.

Evaluación de riesgos

Con el fin de registrar todos los problemas e inconvenientes que vayan apareciendo durante la realización del proyecto se realizará una gestión documental utilizando Google Drive como herramienta que contenga todos los documentos propios del proyecto. El tutor podrá tener acceso permanente a dichos archivos para una comunicación más fluida y directa.

Asimismo, se considera prioritario que en el primer hito quede definido de manera mucho más concisa el alcance del proyecto y las tecnologías a utilizar, para evitar retrasos que imposibiliten la entrega en tiempo del proyecto.

El alumno posee experiencia laboral en entornos .NET y programación de APIs en .NET Framework, pero muy poco conocimiento de tecnologías web de *front end* y .NET Core, por lo que es difícil precisar el alcance de funcionalidades que pueden ser desarrolladas durante el proyecto, ya que en el transcurso del desarrollo del proyecto es necesario adquirir los conocimientos necesarios para su programación.

1.5 Breve sumario de productos obtenidos

A la finalización del proyecto se han obtenido dos productos:

- API REST que permite gestionar operaciones CRUD en el *back end*
- Aplicación web que consume los servicios de la API y permite ejecutar operaciones CRUD, mostrar listados y realizar búsquedas de productos.

1.6 Breve descripción de los otros capítulos de la memoria

En los siguientes capítulos se han definido:

- Objetivo del proyecto
- Diseño del proyecto
- Instalación de lenguajes, tecnologías y herramientas
- Creación de API REST
- Creación de aplicación web Angular 9

2. Objetivo, diseño e instalación

Objetivo del proyecto

El objetivo del proyecto es construir una aplicación que cubra los aspectos básicos de una arquitectura web Angular + ASP .NET Core Universal [1]:

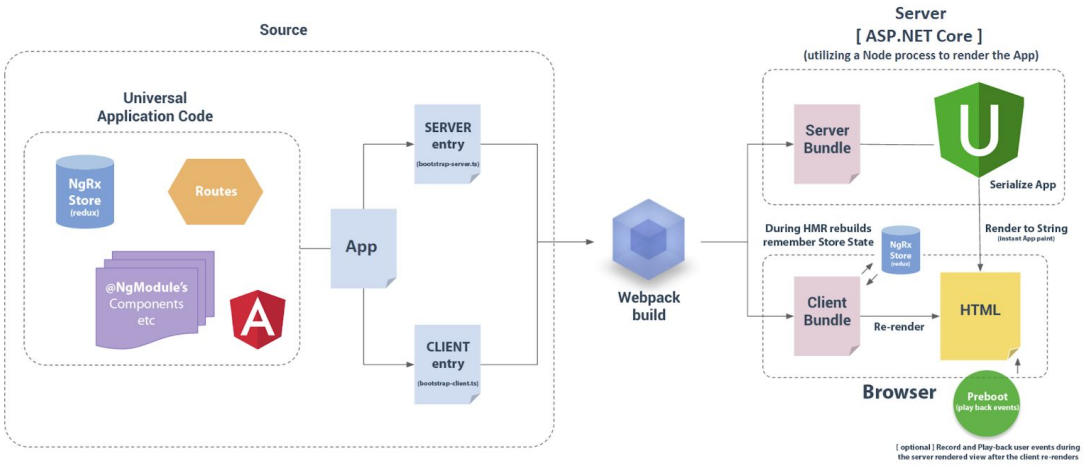


Figura 2: Arquitectura web Angular + ASP .Net Core Universal
Fuente: https://awesomeopensource.com/project/TrilonIO/aspnetcore-angular-universal Consulta 11/04/2020

Asimismo, y de una forma más sencilla, la aplicación construida debe satisfacer la relación entre *front end* y *back end* representada en el siguiente gráfico:

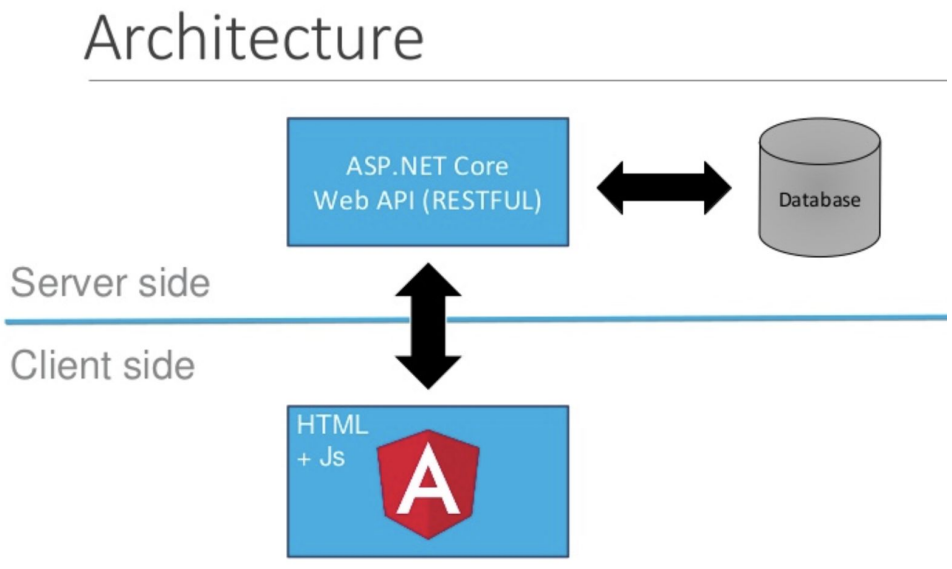


Figura 3: Visión general de arquitectura del proyecto. Fuente: Ottawa IT Community 2018

Diseño del proyecto

Las herramientas que se han utilizado durante el proyecto son las siguientes:

- Docker Desktop. (https://www.docker.com/products/docker-desktop). Es una aplicación que se puede utilizar en máquinas MacOS y Windows y que permite para construir y compartir aplicaciones en forma de contenedor y microservicio. Es un estándar de la industria que destaca por la velocidad y seguridad que aporta al diseñar y mostrar aplicaciones en contenedores en el escritorio. Docker Desktop incluye la aplicación Docker, herramientas para desarrolladores, Kubernetes y sincronización de versiones para motores Docker de producción. Por último, los flujos de trabajo de desarrollo se sirven de Docker Hub para extender su entorno de desarrollo a un repositorio seguro para una rápida construcción automática, integración continua y colaboración segura.

- Postman. Es una herramienta que surgió originalmente como una extensión para el navegador Google Chrome, pero que en la actualidad dispone de aplicaciones nativas para MAC y Windows y permite realizar test de forma sencilla de las operaciones implementadas en la API Web.

- Swagger UI (https://swagger.io/). Es una plataforma que permite diseñar la API y generar la documentación necesaria para su consumo. Del mismo modo, posee un editor intuitivo y potente que facilita el mantenimiento y escalado de la aplicación.
  Según Chakray, es la plataforma más utilizada para documentar API Web ya que a su extremada sencillez hay que sumarle la ventaja de que puede ser leída y comprendida por otras máquinas y permite automatizar directamente procesos dependientes de la API.
  Del mismo modo, Swagger UI es utilizado para organizar los métodos implementados y realizar pruebas durante el desarrollo y mostrar su funcionamiento en posteriores etapas.

- Sqlite3 (https://www.sqlite.org/). Es una base de datos relacional *open source* que también posee integración directa y simple con Docker.

- SQliteFlow. (https://www.sqliteflow.com/). Es un editor para macOS que permite una consulta visual e intuitiva de las tablas contenidas en una base de datos SQLite3, así como realizar consultas SQL.

- Visual Studio 2019 para Mac. (https://visualstudio.microsoft.com/es/vs/mac/). Es un IDE de Microsoft en el que se define el desarrollo de la aplicación web mediante .NET Core, ya que permite una implementación .NET

multiplataforma. Asimismo, posee compatibilidad con Docker de forma integrada. El lenguaje de programación utilizado es C#.

- Angular 9 CLI. (https://angular.io/) Para la parte del *front end* se ha elegido este framework mantenido por Google cuyo objetivo es aumentar las aplicaciones basadas en navegador con Modelo Vista Controlador (MVC), al mismo tiempo que facilita el desarrollo y las pruebas.

- Visual Studio Code para Mac. Es un IDE de Microsoft multiplataforma en el que se define el desarrollo del *front end* utilizando Angular 9. Ha ganado popularidad en los últimos meses entre los desarrolladores por su sencillez de uso, una vasta propuesta de paquetes en su marketplace y continuas actualizaciones mensuales.

- Bootstrap 4. (https://getbootstrap.com/docs/4.0/). Es un *framework front end* que facilita el desarrollo de código web. Incluye HTML y CSS [2] basado en plantillas con tipografías, formularios, botones, tablas, etc. Otra cualidad es su habilidad para crear diseños *responsive* que pueden ser consumidos en distintos tipos de dispositivo. [3]

- JQuery (https://jquery.com/). Es una biblioteca de JavaScript rápida, pequeña y rica en funciones. Hace que cosas como el desplazamiento y la manipulación de documentos HTML, el manejo de eventos, la animación y Ajax sean mucho más simples con una API. Del mismo modo, es fácil de usar y funciona prácticamente en la totalidad de navegadores. Su instalación es necesaria, ya que es utilizada y requerida por Bootstrap 4.

Instalación de lenguajes, herramientas y tecnologías

Para la creación del proyecto, es necesario instalar las herramientas y paquetes software definidos a continuación. Asimismo, se adjuntan capturas que muestran un correcto funcionamiento en el entorno en que son testadas.

- **Docker Desktop**:

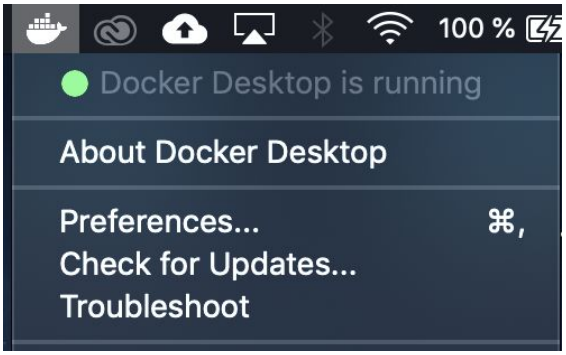


Figura 4: Captura de pantalla con Docker Desktop

Desde la aplicación Terminal de macOs se puede ver que la instalación es correcta, la versión que ha sido instalada y la ejecución de una muestra de una imagen de prueba:

```
MacBook-Pro-de-Borja:~ borjamontesprado$ docker -v
Docker version 19.03.8, build afacb8b
MacBook-Pro-de-Borja:~ borjamontesprado$ docker image list
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              fce289e99eb9        15 months ago       1.84kB
MacBook-Pro-de-Borja:~ borjamontesprado$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

MacBook-Pro-de-Borja:~ borjamontesprado$
```

Figura 5: Captura de pantalla que muestra la versión de Docker Desktop instalada

Para la adquisición de conocimientos necesarios para la interacción de Docker con proyectos que utilizan tecnología .NET Core en sistemas operativos macOS se ha tomado como referencia de consulta el enlace Introducción a Docker en Visual Studio para Mac:
Consulta 12/04/2020
https://docs.microsoft.com/es-es/visualstudio/mac/docker-quickstart?view=vsmac-2019

- **.NET Core para macOS.**

Es necesaria la instalación del SDK de .NET Core 3.1 para macOS, que es obtenido a través del enlace https://dotnet.microsoft.com/download, y que es necesario en la creación del proyecto que crea la API Web .NET Core.

Su correcta instalación se puede comprobar por línea de comandos.

Figura 6: Captura de pantalla que muestra la versión de .NET Core instalada

- **Postman**:

Durante la etapa inicial de la programación de la API se ha utilizado Postman para comprobar el correcto funcionamiento de todas las operaciones GET, PUT, POST y DELETE implementadas en los controladores correspondientes a cada entidad. Para su uso, solamente es necesario seleccionar la operación que se desea consultar e introducir la URL de las rutas de los métodos que se han de consumir. Ver ejemplo de captura:

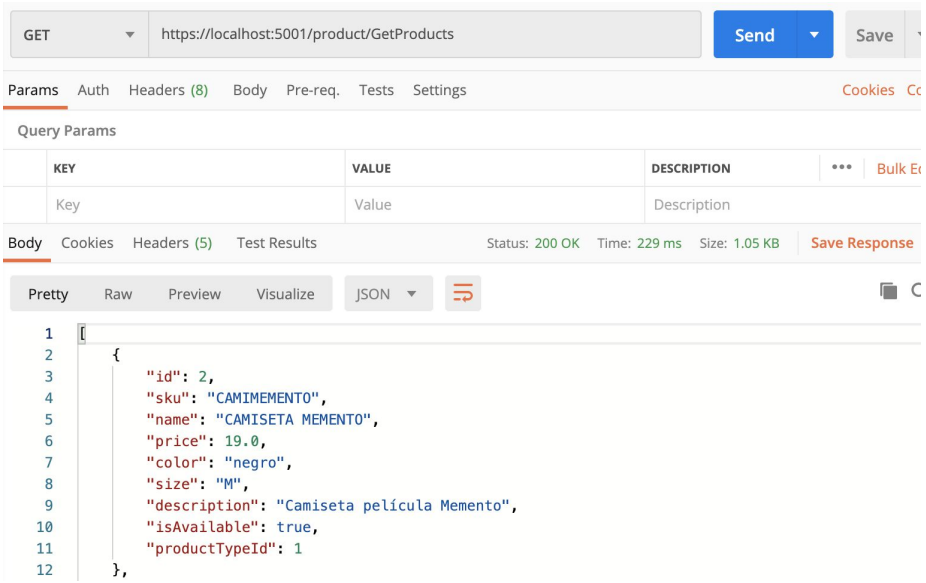


Figura 7: Captura de pantalla que muestra json con la devolución de una petición a la API

Asimismo, la herramienta permite crear entornos e introducir variables que automatizan y ahorran tiempo en los test

- **SQlite3 y SQLiteFlow**:

La instalación habitual de SQLite3 permite visualizar en macOS el contenido de las bases de datos mediante la línea de comandos de la aplicación Terminal, pero se ha utilizado y consumido la versión trial de SQLiteFlow durante el desarrollo de las pruebas de la base de datos utilizada. No ha sido necesario recurrir a versiones más avanzadas, ya que como se podrá comprobar más adelante, la creación de tablas se realiza automáticamente en el *scaffolding* facilitado por Entity Framework Core.

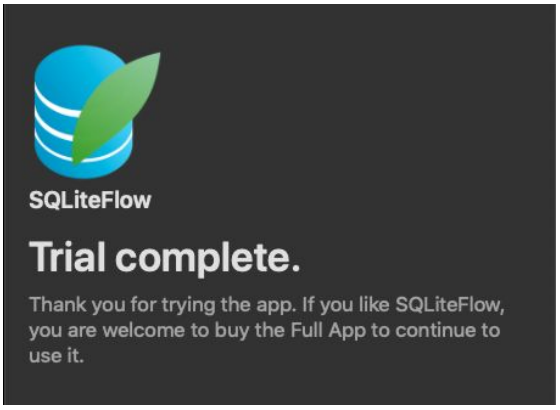


Figura 8: Captura de pantalla que muestra la versión trial finalizada de SQLiteFlow para macOS

- **JQuery 4.51 y Bootstrap 4**

Su instalación se realiza por línea de comandos mediante siguientes comandos:

```
$ npm install --save bootstrap
```

```
$ npm install --save jquery
```

- **Angular 9 CLI**:

Del mismo modo que los anteriores paquetes, utilizando npm se realiza la instalación del CLI de Angular 9 de manera global (opción -g).

```
$ npm install -g @angular/cli
```

3. Creación y programación API REST

Back end.

En primer lugar, cabe recordar que las aplicaciones REST (acrónimo de *Representional State Transfer)* permiten organizar interacciones entre sistemas independientes, por lo que representan la solución actual más adecuada para su comunicación con la aplicación Angular que será construida posteriormente. Las APIs REST tienen un diseño predefinido y se construyen encima del protocolo HTTP, y utilizan URIs para tener acceso a los recursos y verbos HTTP para las operaciones.

Para la creación del proyecto, se ha partido de la plantilla genérica propuesta por la aplicación Visual Studio 2019 y la última versión de .NET Core (v 3.1) como plataforma de destino.

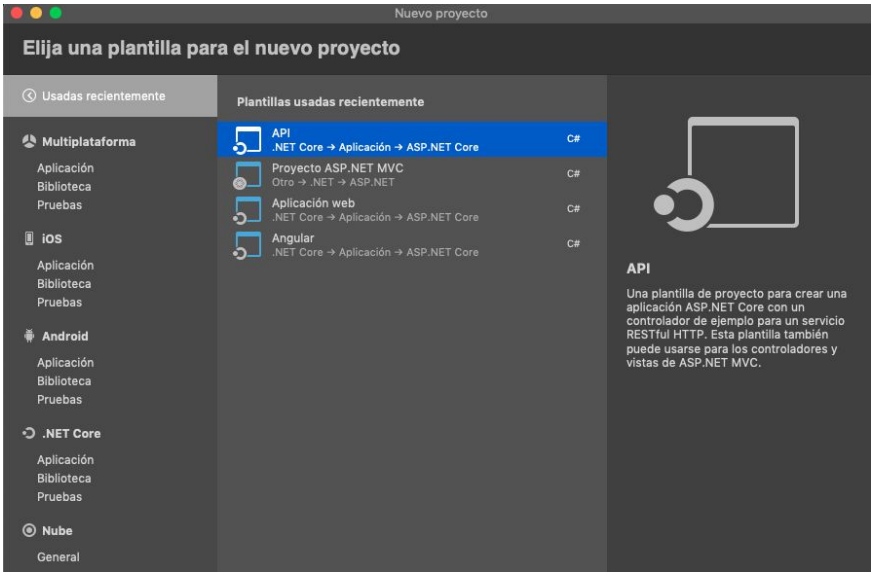


Figura 9: Captura de pantalla que muestra la plantilla utilizada para la creación de la API REST

Asimismo, se ha creado un proyecto en con los archivos necesarios para realizar el vínculo con Docker Desktop:

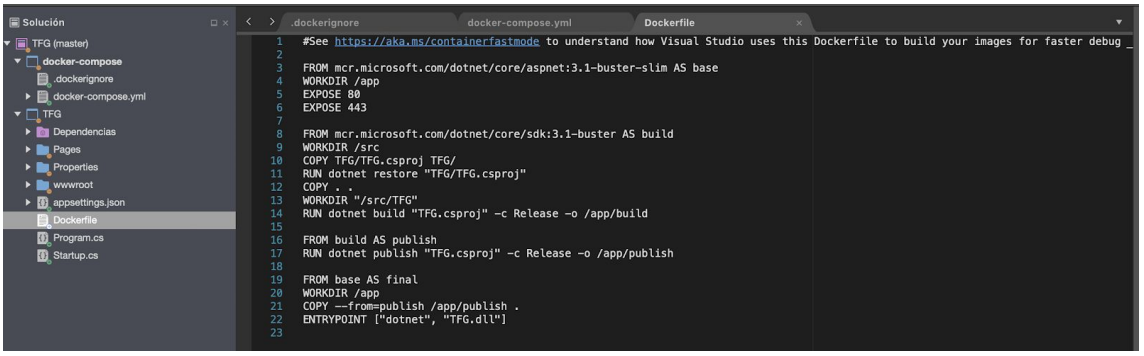


Figura 10: Captura de pantalla que muestra los archivos de configuración de Docker

Nota. En la fase inicial de creación de la API, se ha utilizado como consulta el curso Building Web APIs with ASP.Net Core de Christian Wenz, y reutilizado código en los ficheros Program.cs, Startup.cs, IQueryableExtensions.cs y ProductController.cs. Esta fuente también ha permitido conocer y comprender los procesos necesarios para añadir de forma sencilla a una API mecanismos de autenticación por *tokens* (https://jwt.io/introduction/), así como versionado de API y que serán explicados más adelante .
Del mismo modo, se ha utilizado código propuesto por Microsoft en el fichero ConfigureSwaggerOptions, en el que aparece mencionada su fuente https://github.com/microsoft/aspnet-api-versioning/blob/master/samples/aspnetcore/SwaggerSample/ConfigureSwaggerOptions.cs

La estructura básica de la API REST queda compuesta del siguiente modo:

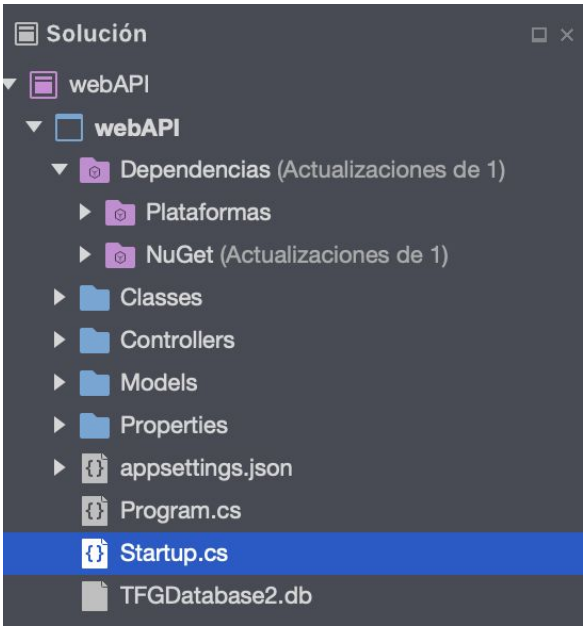


Figura 11: Captura de pantalla que muestra la estructura de carpetas de la API REST

Modelo de datos

Para la realización de la base de datos, se ha optado por elegir un modelo consistente y testado previamente que ha sido desarrollado por Barry Williams (LinkedIn) y está disponible para consumo público en la web http://www.databaseanswers.org/

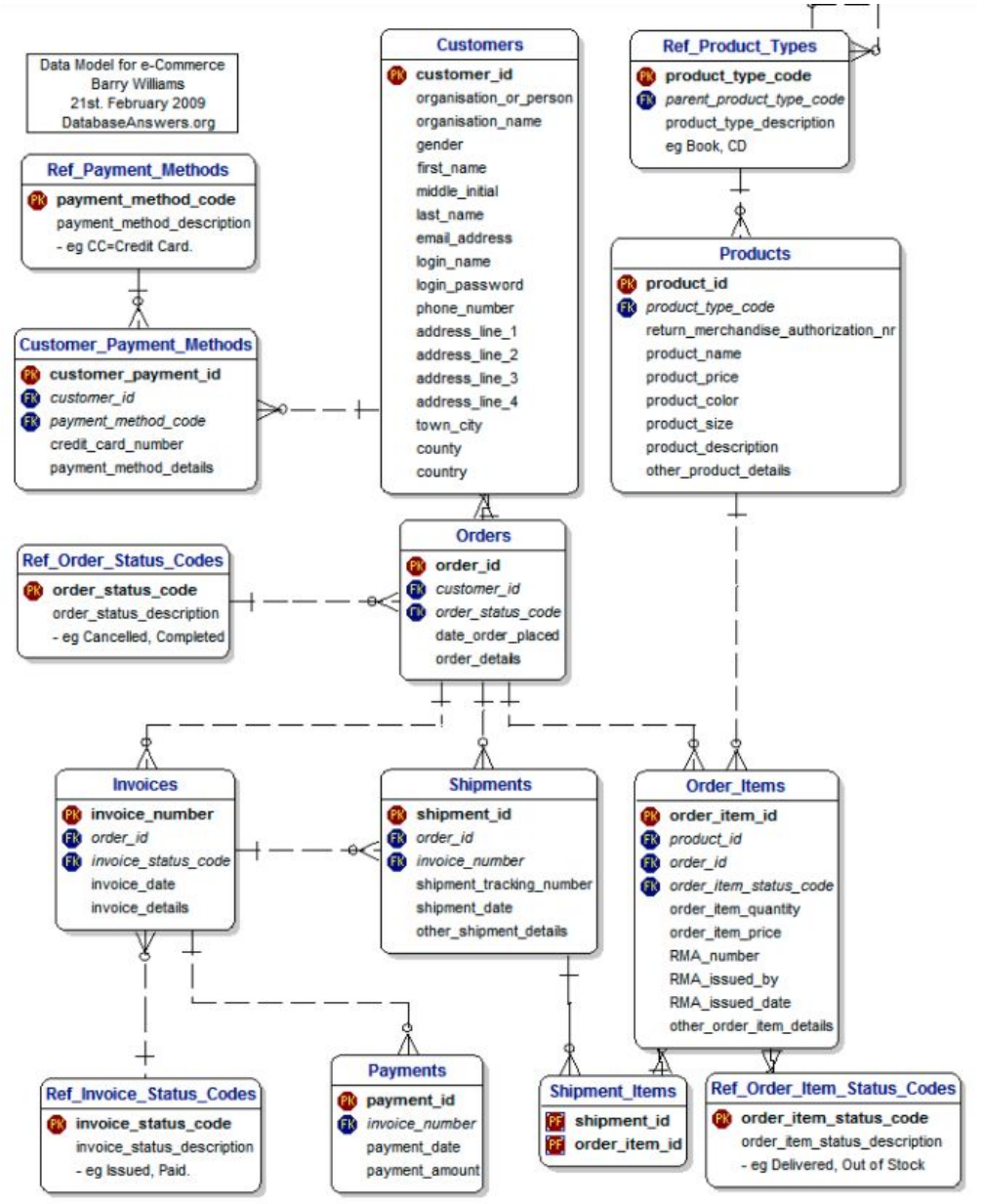


Figura 12: Diseño de base de datos

Por no ser objetivo principal de este proyecto, algunas de las entidades y atributos no han sido definidas en el código, pero se han respetado las relaciones propuestas en el modelo. Estas entidades se almacenan en la carpeta Models de la webAPI. En este casom se han implementado las entidades Customer, RefProductType, Product, Orders, OrderItems, Shipment e Invoice y se ha obviado el resto de entidades.

A continuación, se muestra un ejemplo de la entidad Order que se crea en la carpeta Models con sus respectivos atributos y relaciones. El resto de entidades siguen las mismas convenciones y estructuras.

```
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;

namespace webAPI.Models
{
    public class Order
    {
        public int Id { get; set; }
        public int CustomerId { get; set; }
        public DateTime DateOrderPlaced { get; set; }
        public string Details { get; set; }
        [JsonIgnore]
        public virtual Customer Customer { get; set; }
        public virtual List<OrderItem> OrderItems { get; set; }
        public virtual List<Shipment> Shipments { get; set; }
        public virtual List<Invoice> Invoices { get; set; }
    }
}
```

Figura 13: Implementación de la clase Order en API

Nota. Se utiliza la etiqueta Json Ignore sobre el atributo Customer para evitar referencias cíclicas dentro de la clase.

Entity Framework Core

Una vez definidos los modelos de las clases, y antes de comenzar a implementar la API es necesario realizar la configuración del modelo de datos. En este caso se ha escogido Entity Framework Core para relacionar una base de datos relacional con el modelo.

El primer paso, es instalar el paquete NuGet, Microsoft.EntityFrameworkCore

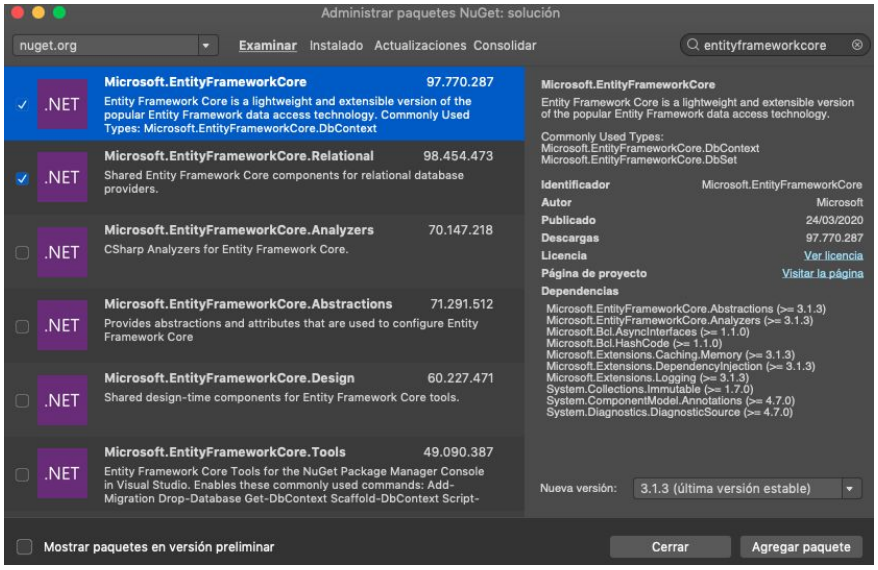


Figura 14: Captura de pantalla que muestra la instalación del paquete Microsoft. EntityFrameworkCore

Una vez instalado, dentro de los modelos se define una clase MyDbContext, que hereda de DbContext que permite relacionar los modelos ya creados con las tablas de bases de datos en las que se gestiona la información que consume la API.

```
namespace webAPI.Models
{
        public class MyDbContext : DbContext
        {
        public MyDbContext(DbContextOptions<MyDbContext> options) : base(options)
        { }

        public DbSet<Product> Products { get; set; }
        public DbSet<RefProductType> RefProductTypes { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Shipment> Shipments { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<Invoice> Invoices { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
        optionsBuilder.UseSqlite("Filename=TFGDatabase2.db");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
        modelBuilder.Entity<Customer>().ToTable("Customers");
        modelBuilder.Entity<Invoice>().ToTable("Invoice");
        modelBuilder.Entity<Order>().ToTable("Order");
```

```
        modelBuilder.Entity<OrderItem>().ToTable("OrderItem");
        modelBuilder.Entity<Shipment>().ToTable("Shipment");
        modelBuilder.Entity<Product>().ToTable("Product");
        modelBuilder.Entity<RefProductType>().ToTable("RefProducttType");

        }
        }
}
```

En el fragmento anterior, se ve cómo de manera muy sencilla se realiza el vínculo entre la base de datos SqlLite, las entidades y las tablas. [4] La posterior ejecución de este código, generará un *scaffolding* automático que evita crear manualmente las tablas implicadas.

Una vez realizado el anterior paso, y mediante inyección de dependencias se agrega la clase MyDbContext a los servicios del interfaz Configuracion contenido en el fichero Startup.cs

```
public void ConfigureServices(IServiceCollection services)
        {
        services.AddDbContext<MyDbContext>(options =>
                options.UseSqlite("MyDbContext"));
        services.AddControllers()
…….
```

URIs

El paso siguiente consiste en definir todas las URIs que serán utilizadas posteriormente por el controlador correspondiente de cada entidad. Del mismo modo que en el caso anterior, a modo de resumen se muestran únicamente las URIs definidas para la entidad Customer. El resto de entidades realizan acciones semejantes.

Verbo HTTP	**URI**	**Acción**
GET	https://localhost:5001/customer	Devuelve todos los customers
PUT	https://localhost:5001/customer/1	Inserta un nuevo customer con id=1
POST	https://localhost:5001/customer/1	Actualiza los datos del customer con id=1
DELETE	https://localhost:5001/customer/1	Elimina los datos del customer con id=1

Como se ha mencionado previamente, todas estas URIs han sido cargadas en Postman para el testado de la API Web durante su programación.

Controladores y acciones

Una vez definidos los modelos, la base de datos y las URIs, ya es posible comenzar la programación de los controladores y sus correspondientes acciones. A modo de ejemplo, se muestra el código del controlador del modelo de Customer.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using webAPI.Classes;
using webAPI.Models;

namespace webAPI.Controllers
{
        [ApiVersion("1.0")]
        [Route("customer")]
        [ApiController]
        public class CustomerController : ControllerBase
        {
        private readonly MyDbContext _context;

        public CustomerController(MyDbContext context)
        {
        _context = context;
        _context.Database.EnsureCreated();
        }

        [HttpGet]
        public async Task<IActionResult> GetAllCustomers([FromQuery]
CustomerQueryParameters query)

        {
        IQueryable<Customer> customers = _context.Customers;

        var c = customers.Count();
```

```
        if (!string.IsNullOrEmpty(query.LoginName))
        {
                customers = customers.Where(p => p.LoginName == query.LoginName);
        }

        if (!string.IsNullOrEmpty(query.SortBy))
        {
                if (typeof(Customer).GetProperty(query.SortBy) != null)
                {
                customers = customers.OrderByCustom(query.SortBy, query.SortOrder);
                }
        }

        customers = customers.Skip(query.Size * (query.Page - 1)).Take(query.Size);
        return Ok(await customers.ToArrayAsync());
        }

        [HttpGet]
        [Route("{id}")]
        public async Task<IActionResult> GetCustomer(int id)
        {
        var customer = await _context.Customers.FindAsync(id);
        if (customer == null)
        {
                return NotFound();
        }
        return Ok(customer);
        }

        [HttpPost]
        public async Task<ActionResult<Customer>> InsertCustomer([FromBody] Customer customer)
        {
        _context.Customers.Add(customer);
        await _context.SaveChangesAsync();
        return CreatedAtAction(
                "GetCustomer",
                new { id = customer.Id },
                customer
                );
        }

        [HttpPut("{id}")]
```

```
        public async Task<ActionResult<Customer>> UpdateCustomer(int id,
[FromBody] Customer customer)
        {

        if (id != customer.Id)
        {
                return BadRequest();
        }

        _context.Entry(customer).State = EntityState.Modified;

        try { await _context.SaveChangesAsync(); }
        catch (DbUpdateConcurrencyException)
        {
                if (_context.Customers.Find(id) == null)
                {
                return NotFound();
                }
        }

        return NoContent();
        }

        [HttpDelete("{id}")]
        public async Task<ActionResult<Customer>> DeleteCustomer(int id)
        {
        var customer = await _context.Customers.FindAsync(id);
        if (customer == null)
        {
                return NotFound();
        }

        _context.Customers.Remove(customer);
        await _context.SaveChangesAsync();
        return customer;
        }
        }
}
```

En el controlador, se puede ver la relación entre las URIs anteriormente definidas y las operaciones planteadas. Se puede observar que el controlador CustomerController es una clase heredada de ControllerBase (enlace) que ha sido creada para su uso en ASP NET.Core, en lugar de la clase ApiController que era utilizada en .NET Framework.

Por otro lado, justo antes de definir la clase CustomerController se define la ruta del controlador utilizando la etiqueta Route, en este caso [Route("customer")], valor que se puede observar en las URIs de Customer.

Las acciones GET, POST, PUT y DELETE dan acceso a los correspondientes métodos GetAllCustomers, InsertCustomer, UpdateCustomery DeleteCustomer.

Por último, en la devolución de las peticiones se implementa la interfaz IActionResult que devuelve el estado HTTP del servicio, además de la información solicitada, lo que permite detallar la causa de los posibles errores o el éxito tras la ejecución del método.

Algunos de los distintos códigos numéricos devueltos por el método Ok IActionResult devuelve son:

- 200 para indicar éxito
- 204 para indicar que no hay contenido para los valores solicitados
- 400 para indicar una petición incorrecta
- 404 para una petición inexistente.

Dicho esto, para una información más detallada, se pueden implementar métodos que vinculen las respuesta y permitan gestionar las excepciones. En este proyecto se utilizan sólamente algunos métodos, como NotFound, Conflict, BadRequest, no obstante, se pueden consultar información más detallada de las respuestas no exitosas en el *RFC Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* (https://tools.ietf.org/html/rfc7231), texto al que están vinculadas las respuestas de los métodos descritos.

En el siguiente ejemplo se puede observar la consulta de un producto no existente en base de datos, Y cómo se puede utilizar la respuesta que Postman obtiene para obtener más información.

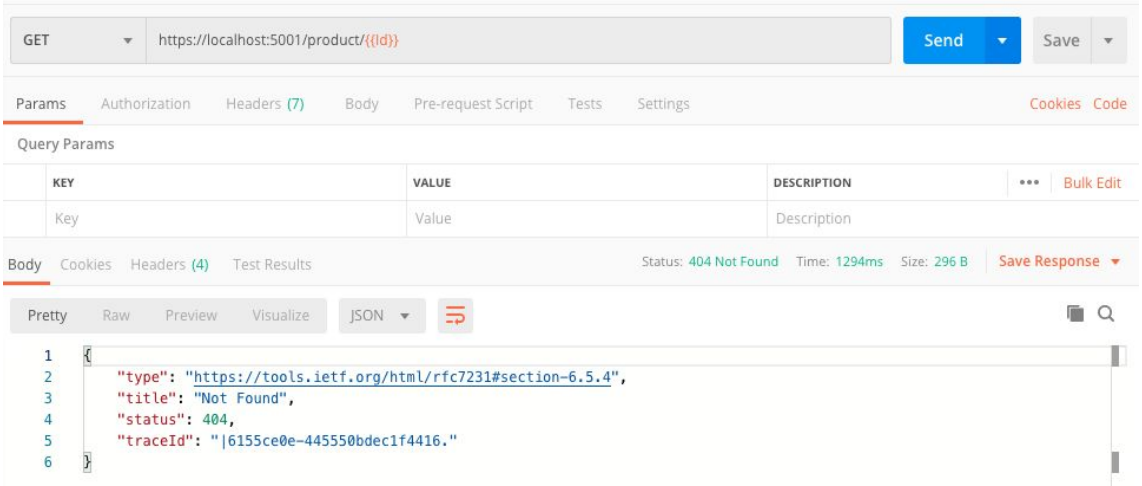


Figura 15: Captura de pantalla que muestra error devuelto por API

En el Json devuelto por el servicio, se puede observar cómo se devuelve un valor '*Not found*' para el producto solicitado (se ha seleccionado una id diferente a las existentes para recibir la excepción. La url indicada en el atributo *'type'* muestra información detallada al consumidor de la API con la causa del fallo en la devolución del método.

Del mismo modo, la API también gestiona las peticiones incorrectas y maneja el error alertando al usuario del problema detectado en la petición. Por ejemplo, la introducción de un id con un formato incorrecto. Asimismo, se puede observar que los atributos 'type' y 'status' han cambiado.

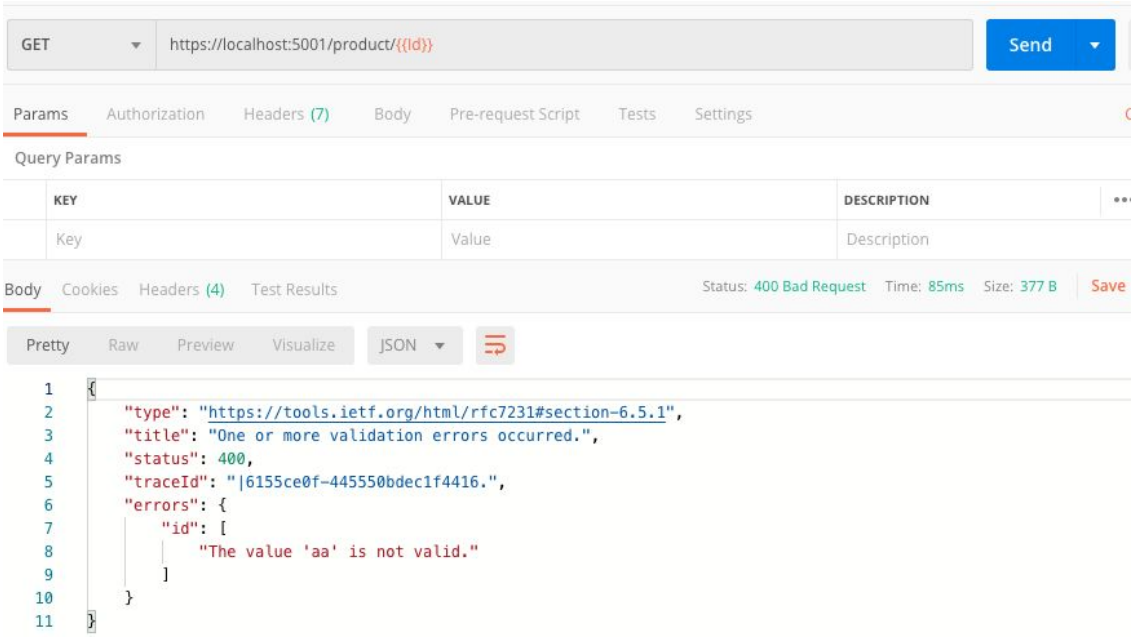


Figura 16: Captura de pantalla que muestra error en petición incorrecta a la API

Query y QueryParameters

En la carpeta Classes se han definido clases que permiten utilizar diferentes parámetros dentro de las consultas realizadas en cada controlador correspondiente. Básicamente su función es proporcionar consultas más elaboradas, precisas y ordenadas que las que pueden generar por defecto los métodos implementados por defecto.

Versionado API

Es lógico que las APIs vayan evolucionando y cambiando en función de las necesidades de las aplicaciones que las consumen, por lo que resulta muy útil poder

gestionar las distintas versiones generadas en el tiempo. Para ello, es necesario instalar el paquete NuGet Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer.

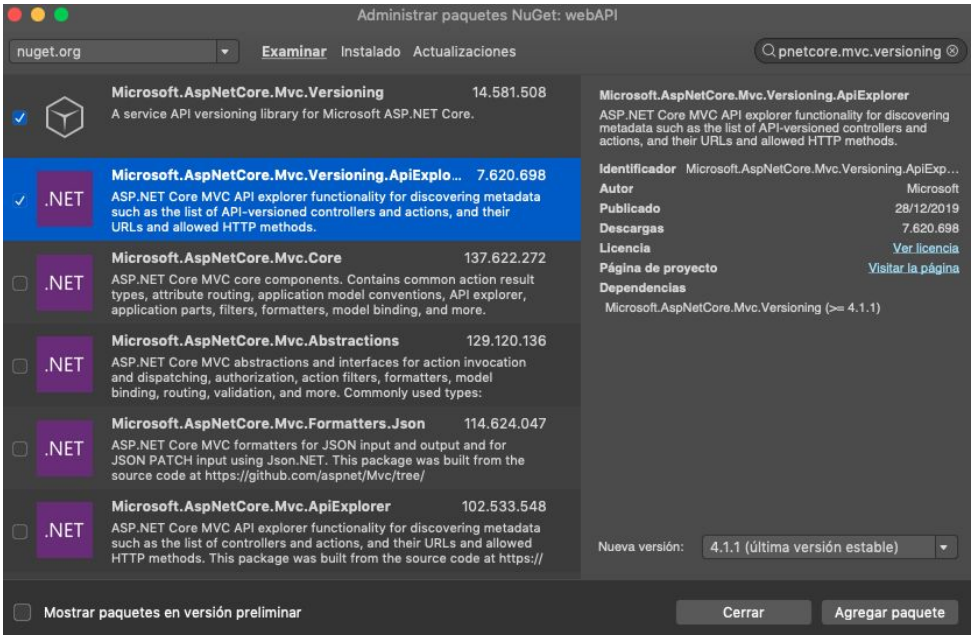


Figura 17: Captura de pantalla que muestra la instalación del paquete Microsoft.AspNetCore.Mvc.Versioning.ApiExplore

De manera muy sencilla, en los servicios de configuración se ejecuta el método AddAPiVersioning que incluirá los parámetros deseados de versionado. En el caso de nuestra API

```
services.AddApiVersioning(options => {
        options.ReportApiVersions = true;
        options.DefaultApiVersion = new ApiVersion(1, 0);
        options.AssumeDefaultVersionWhenUnspecified = true;
        options.ApiVersionReader =
        new HeaderApiVersionReader("X-API-Version");
```

Swagger UI:

Para poder tener acceso al interfaz de usuario de Swagger, en primer lugar, se han instalado los paquetes Swashbuckle.AspNetCore y NSwag.NetCore, provistos por nuget.org.

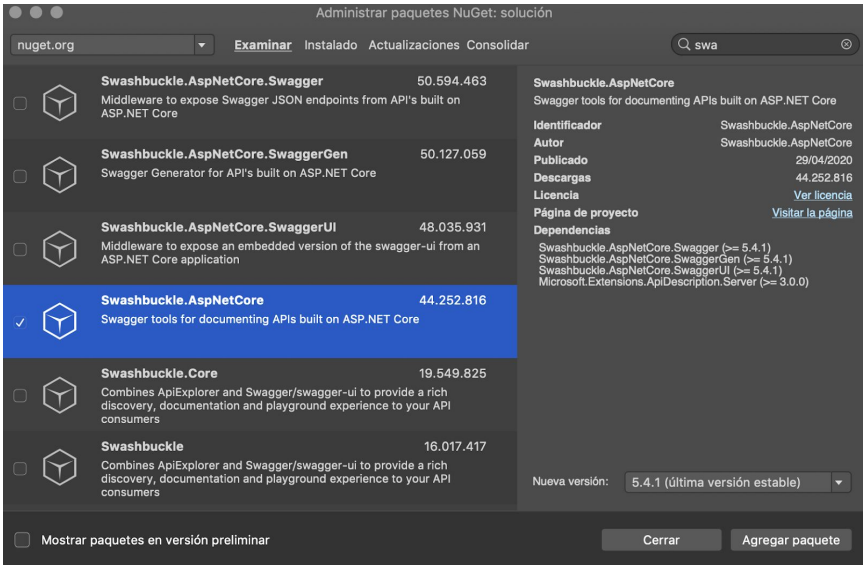


Figura 18: Captura de pantalla que muestra la instalación del paquete Swashbuckle.AspNetCore

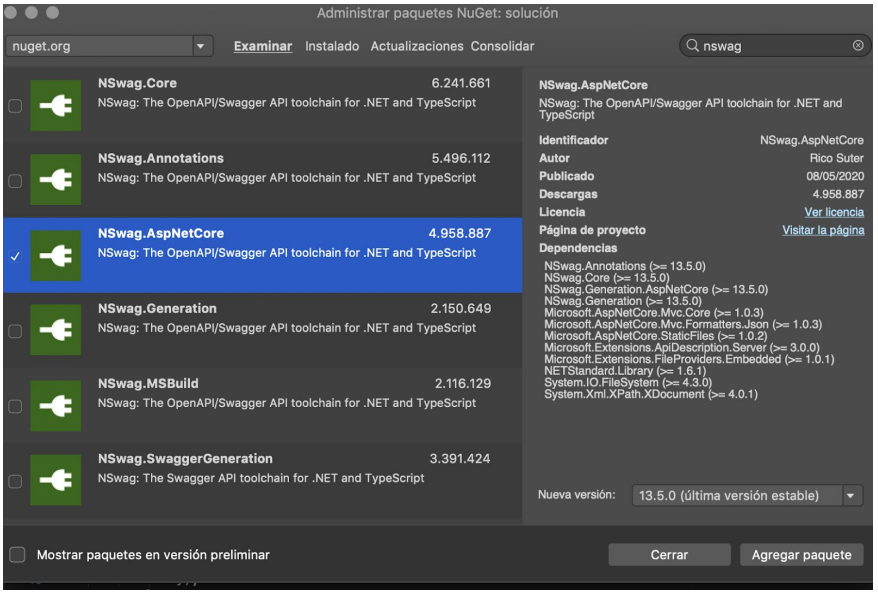


Figura 19: Captura de pantalla que muestra la instalación del paquete NSwag.AspNetCore

Una vez instalados, y de manera similar al versionado API, se utiliza la clase ConfigureSwaggerOptions propuesta por Microsoft y se inserta en los servicios de configuración.

Una vez programada esta acción, se podrán realizar consultas y test con Swagger UI, así cómo tener acceso a la documentación de la API. Para acceder a ella, se ejecuta la aplicación y se accede desde un navegador web a la URL https://localhost:5001/swagger que muestra:

Figura 20: Captura de pantalla que muestra resultado en formato JSON del método GET Customer utilizando UI Swagger

Cross-Origin Resource Sharing

En la arquitectura propuesta, existe el problema de que Javascript no puede hacer solicitudes de dominio cruzado debido a determinados riesgos inherentes de seguridad. Por ejemplo, la falsificación de solicitudes entre aplicaciones podría permitir ataques potencialmente peligrosos. Por lo tanto, por defecto están bloqueados, pero lógicamente se puede configurar su desbloqueo entre aplicaciones que cumplan determinados requisitos. El uso compartido de recursos Cross-Origin [5] permite que cuando se realiza una petición POST o PUT, automáticamente se agrega un encabezado HTTP Origin, y si el origen es conocido, la API permite devolver la respuesta solicitada. En el caso de los GET, si la solicitud no tiene permiso de acceso de dominio cruzado, simplemente no devolverá información alguna.

Habilitar CORS en .NET Core y establecer las políticas necesarias, se puede hacer de manera relativamente sencilla, ya que los servicios de configuración pueden agregarlo con una llamada que contenga los parámetros de validación que se consideren oportunos. En este caso:

```
services.AddCors(options =>
{
        options.AddDefaultPolicy(builder =>
        {
        builder.WithOrigins("http://localhost:4123")
        .AllowAnyHeader()
        .AllowAnyMethod();
```

```
        });
});
```

Se autoriza el uso de una aplicación que se comunique por mediación de la dirección http://localhost:4123, y permitirá todas sus cabeceras y métodos. También pueden aplicarse políticas individuales mediante el uso de etiquetas para cada método de cada controlador o individualmente para todas las operaciones de controlador, pero para evitar problemas derivados, básicamente se permitirán todas las solicitudes provenientes de la aplicación Angular 9 que será ejecutada en la URL http://localhost:4123.

JWTBearer

Por último, a pesar de no estar implementada la autenticación de la API, se ha añadido a los servicios existentes, el método de autenticación de JWT

```
services.AddAuthentication("Bearer")
        .AddJwtBearer("Bearer", options =>
        {
        options.Authority = "http://localhost:54229";
        options.RequireHttpsMetadata = false;

        options.Audience = "webapi";
        });
```

4. Cliente web Angular 9

Se crea el proyecto Angular 9 desde un terminal unix, encargado de controlar el *front end* y al que se añade por defecto el routing y CSS para la gestión de estilos en la presentación web:

```
[MacBook-Pro-de-Borja:TFG borjamontesprado$ ng new TFG-Angular-App
[? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
› CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
  Stylus [ http://stylus-lang.com                                         ]
```

Figura 21: Captura de pantalla que muestra las opciones de *routing* y estilo seleccionados

La estructura del proyecto desde Visual Studio Code es la siguiente:

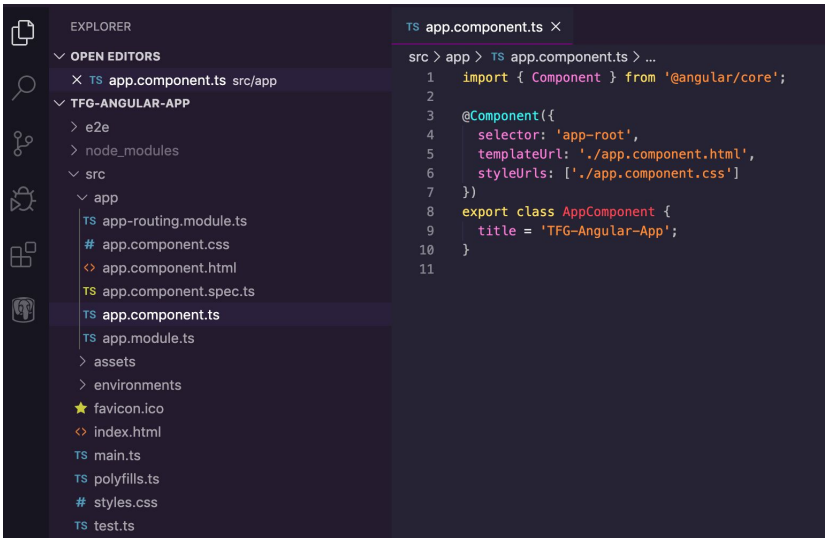


Figura 22: Captura de pantalla que muestra la estructura de la aplicación Angular 9

Y la funcionalidad programada se limita a ejecutar las operaciones CRUD de la entidad Producto.

En este momento, se añaden las carpetas:

- Components
- Interface
- Services

En la primera, se añaden los componentes de navegación y todas y cada una de las operaciones que serán realizadas desde Angular 9.

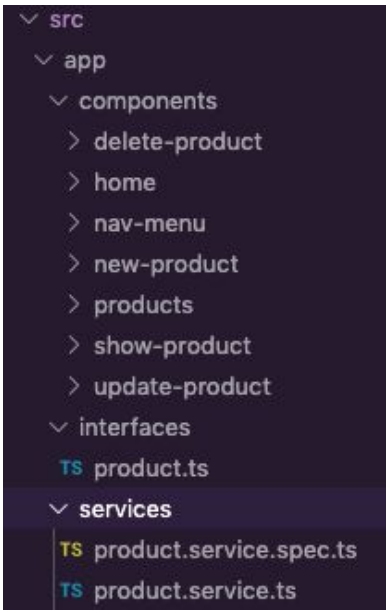


Figura 23: Captura de pantalla que muestra la estructura de la carpetas de componentes

En la carpeta Interfaces se guarda la clase que almacena el interface product y en futuras ampliaciones, es la carpeta en la que deben de almacenarse los nuevos interfaces.

```
export interface Product {
 id: number;
 sku: string;
 name: string;
 description: string;
 color: string;
 size: string;
 isAvailable: boolean;
 price: number;
 productTypeId: number;
}
```
Figura 24: Captura de pantalla que muestra la implementación de la interfaz Product

Finalmente, en services se configuran las peticiones que han de ser enviadas a la API REST.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Product} from 'src/app/interfaces/product';
```

```
@Injectable({
 providedIn: 'root'
})
export class ProductService {

 _baseURL: string = "https://localhost:5001/product";

 constructor(private http: HttpClient) { }

 getAllProducts(){
   return this.http.get<Product[]>(this._baseURL+"/GetProducts");
 }

 addProduct(product: Product){
   return this.http.post(this._baseURL, product);
 }

 getProductById(id: number){
   return this.http.get<Product>(this._baseURL+"/"+id);
 }

 getProductByName(name: string){
   return this.http.get<Product[]>(this._baseURL+"/name/"+name);
 }

 updateProduct(product: Product){
   return this.http.put(this._baseURL+"/"+product.id, product);
 }

 deleteProduct(id: number){
   return this.http.delete(this._baseURL+"/"+id);
 }

}
```

Figura 25: Operaciones que pueden ser solicitadas por la aplicación web

Como se puede observar, se establece una URL Base en la que está la dirección de ejecución de la api y posteriormente son añadidas las variables necesarias.

Asimismo, dentro del fichero de configuración app.module.ts, se definen las rutas y componentes que son utilizadas por la aplicación Angular 9

```
import { AppComponent } from './app.component';
import { NavMenuComponent } from './components/nav-menu/nav-menu.component';
import { HomeComponent } from './components/home/home.component';
import { ProductsComponent } from './components/products/products.component';
import { DeletePrductComponent } from
'./components/delete-product/delete-product.component';
import { NewProductComponent } from
'./components/new-product/new-product.component';
import { ShowProductComponent } from
'./components/show-product/show-product.component';
import { UpdateProductComponent } from
'./components/update-product/update-product.component';
import { ProductService } from './services/product.service';



@NgModule({
 declarations: [
   AppComponent,
   NavMenuComponent,
   HomeComponent,
   ProductsComponent,
   DeleteProductComponent,
   NewProductComponent,
   ShowProductComponent,
   UpdateProductComponent
 ],
 imports: [
   BrowserModule.withServerTransition({ appId: 'ng-cli-universal' }),
   HttpClientModule,
   FormsModule,
   ReactiveFormsModule,
   RouterModule.forRoot([
     { path: '', component: HomeComponent, pathMatch: 'full' },
     { path: 'products', component: ProductsComponent},
```

```
    { path: 'new-product', component: NewProductComponent},
    { path: 'update-product/:id', component: UpdateProductComponent},
    { path: 'delete-product/:id', component: DeleteProductComponent},
    { path: 'show-product/:id', component: ShowProductComponent}
  ]),

],
providers: [ProductService],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Figura 25: Archivo app.module.ts

Desde Visual Studio Code o un terminal se ejecuta la aplicación utilizando el comando ng serve --port 4123 (Recordemos los problemas comentados de CORS)

La ejecución de la aplicación muestra una sencilla web de página única que permite realizar las operaciones de consulta, modificación y borrado de los productos almacenados en base de datos.

5. Resultados

A continuación, se presentan capturas de pantalla de la aplicación en tiempo de ejecución:

Página de Inicio

Figura 26: Captura de página inicial

Nuevo producto

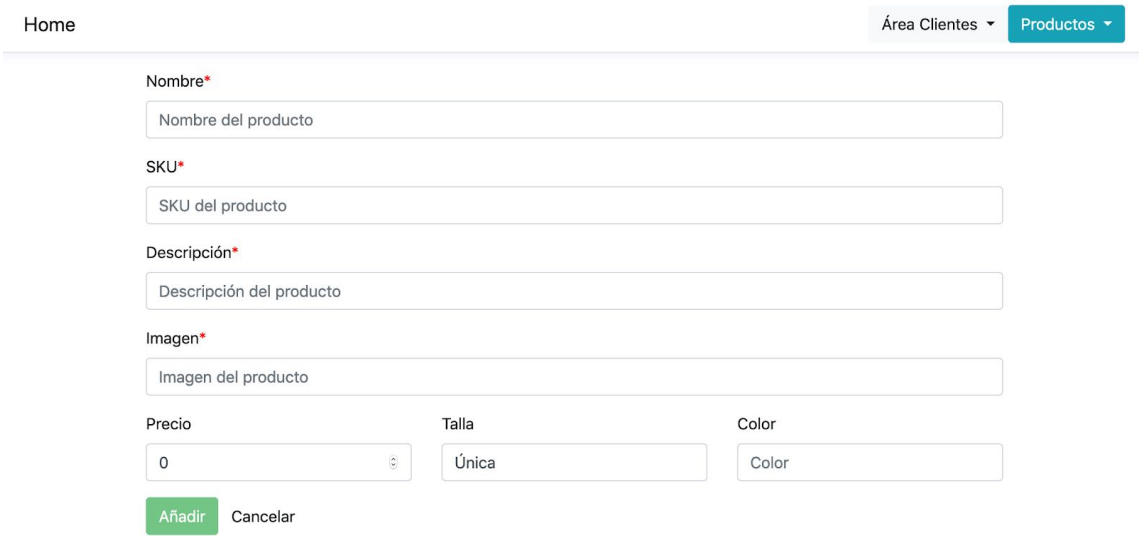
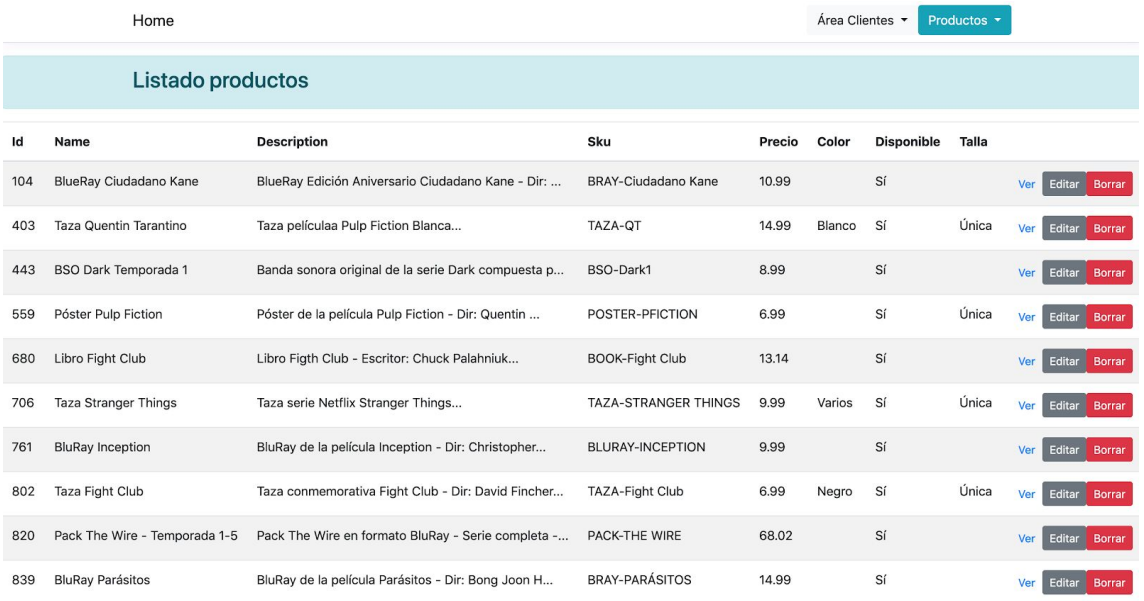


Figura 28: Captura de página de la funcionalidad que permite añadir un nuevo producto

Listado de productos para el administrador de la aplicación web



Id	Name	Description	Sku	Precio	Color	Disponible	Talla	
104	BlueRay Ciudadano Kane	BlueRay Edición Aniversario Ciudadano Kane - Dir: ...	BRAY-Ciudadano Kane	10.99		Sí		Ver Editar Borrar
403	Taza Quentin Tarantino	Taza películaa Pulp Fiction Blanca...	TAZA-QT	14.99	Blanco	Sí	Única	Ver Editar Borrar
443	BSO Dark Temporada 1	Banda sonora original de la serie Dark compuesta p...	BSO-Dark1	8.99		Sí		Ver Editar Borrar
559	Póster Pulp Fiction	Póster de la película Pulp Fiction - Dir: Quentin ...	POSTER-PFICTION	6.99		Sí	Única	Ver Editar Borrar
680	Libro Fight Club	Libro Figth Club - Escritor: Chuck Palahniuk...	BOOK-Fight Club	13.14		Sí		Ver Editar Borrar
706	Taza Stranger Things	Taza serie Netflix Stranger Things...	TAZA-STRANGER THINGS	9.99	Varios	Sí	Única	Ver Editar Borrar
761	BluRay Inception	BluRay de la película Inception - Dir: Christopher...	BLURAY-INCEPTION	9.99		Sí		Ver Editar Borrar
802	Taza Fight Club	Taza conmemorativa Fight Club - Dir: David Fincher...	TAZA-Fight Club	6.99	Negro	Sí	Única	Ver Editar Borrar
820	Pack The Wire - Temporada 1-5	Pack The Wire en formato BluRay - Serie completa -...	PACK-THE WIRE	68.02		Sí		Ver Editar Borrar
839	BluRay Parásitos	BluRay de la película Parásitos - Dir: Bong Joon H...	BRAY-PARÁSITOS	14.99		Sí		Ver Editar Borrar

Figura 29: Captura de página que muestra a un usuario administrador el listado de productos disponibles

Vista individual de producto

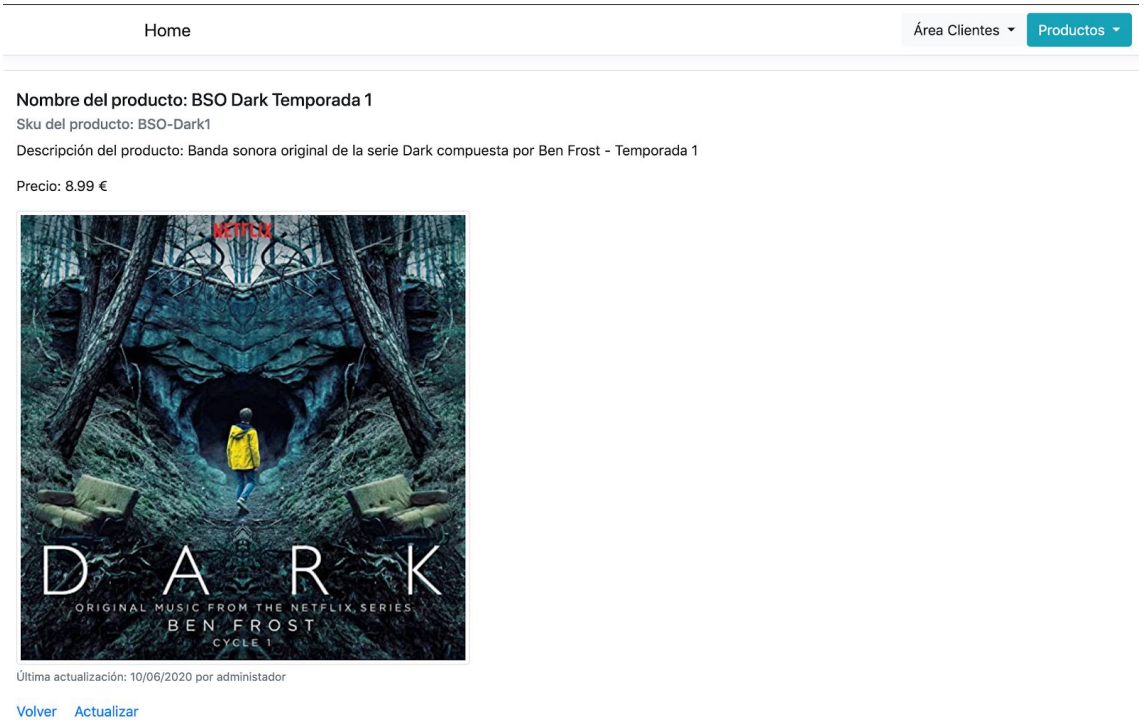


Figura 30: Captura de página que muestra el contenido de un producto concreto

Edición de producto

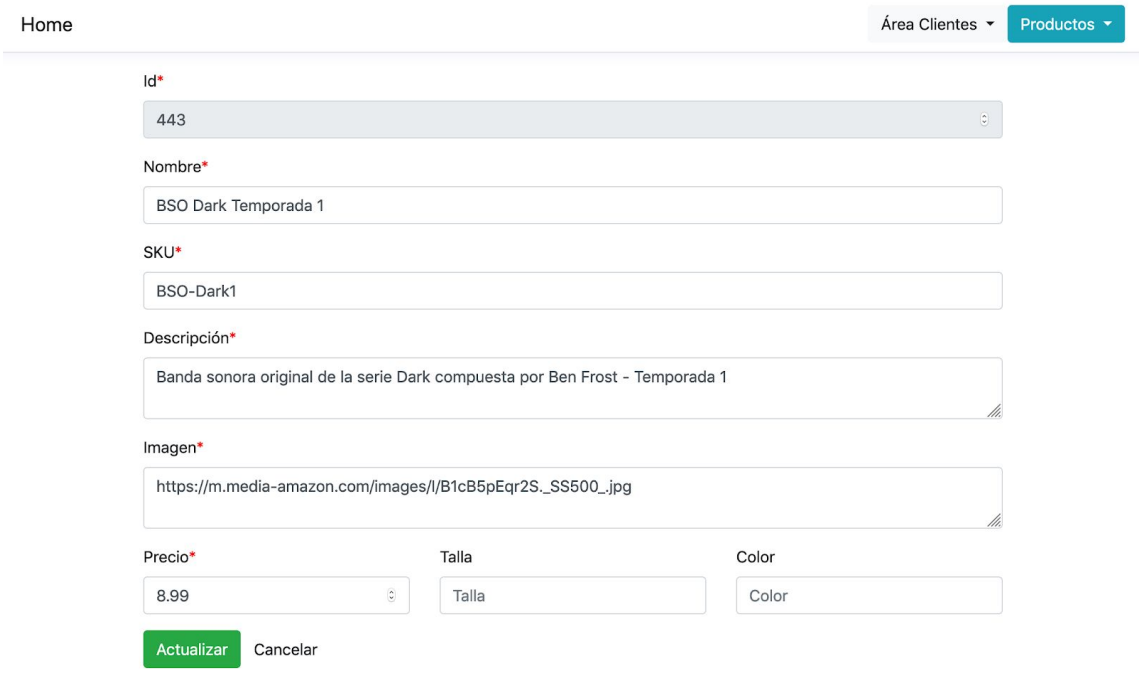


Figura 31: Captura de página de edición de producto

Confirmación de borrado de producto.

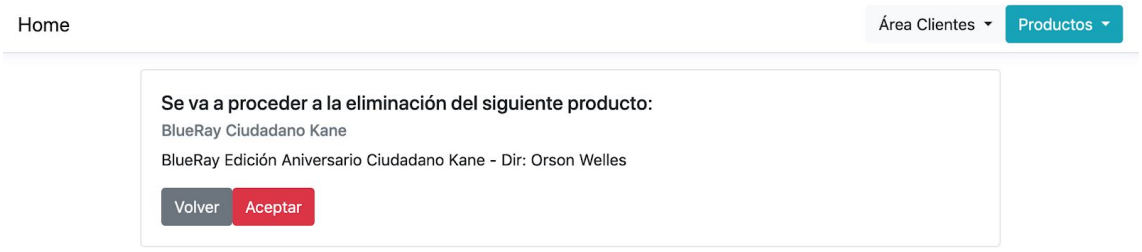


Figura 32: Captura de página de confirmación de borrado de un producto

Listado de producto en área de clientes

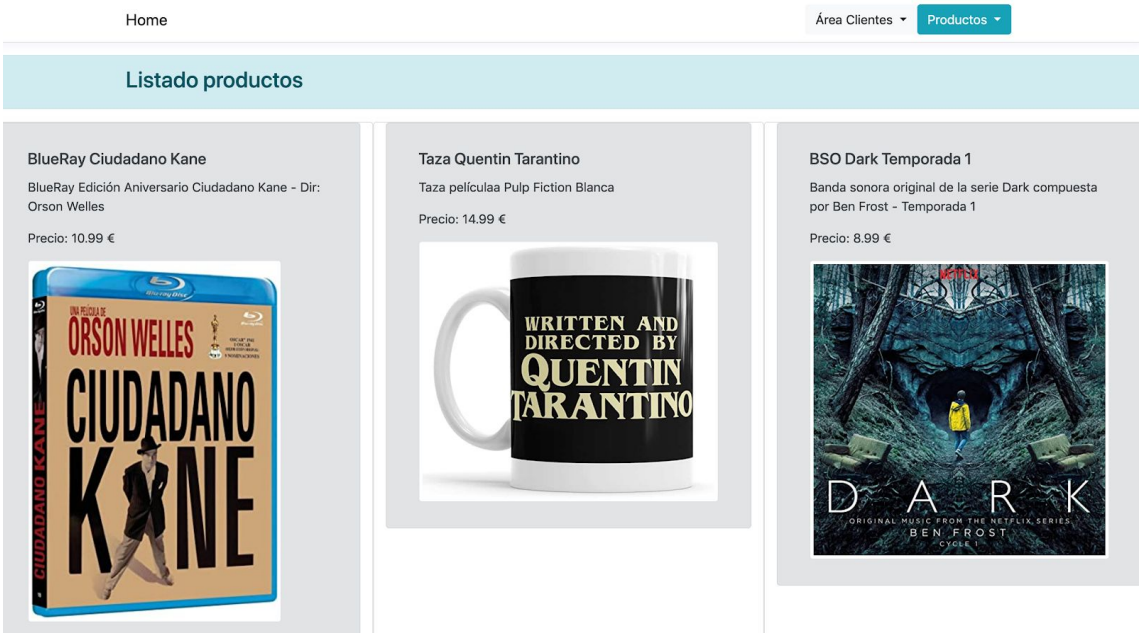


Figura 33: Captura de página que muestra listado de productos disponibles para los clientes

Buscador de productos

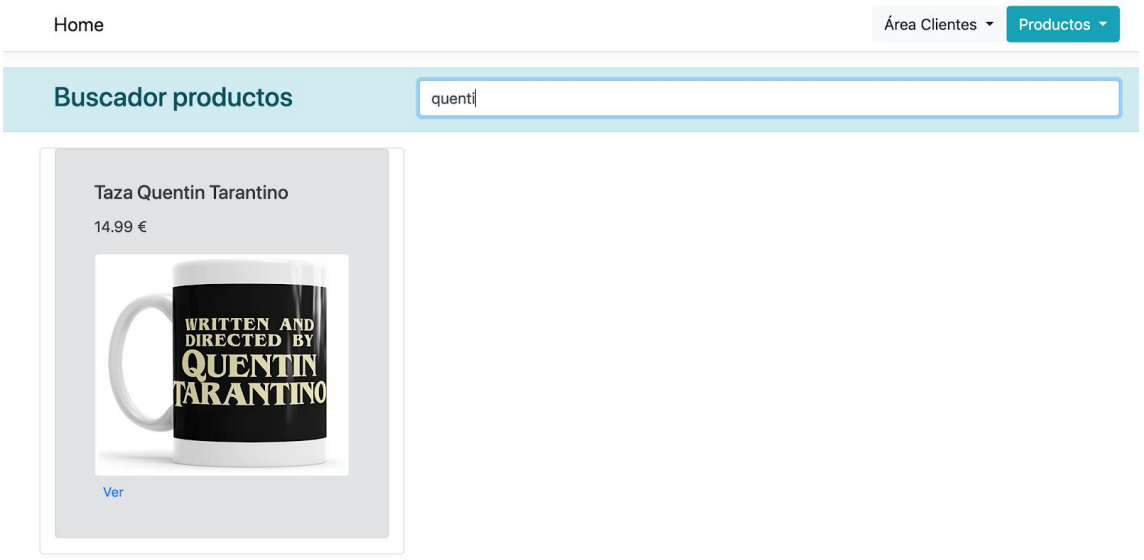


Figura 34: Captura de buscador de productos

6. Conclusiones

La conclusión principal de este proyecto ha sido que desarrollar una arquitectura web utilizando la plataforma .NET Core fuera del ecosistema del sistema operativo Windows no presenta problemas adicionales significativos. Más allá de pequeñas funcionalidades que no están presentes en el IDE Visual Studio 2019 para Mac, la experiencia de programación ha sido satisfactoria y el resultado final puede considerarse idéntico al que se hubiera conseguido utilizando el sistema operativo Windows 10.

Asimismo, se ha podido comprobar que la curva de aprendizaje de Angular para un usuario sin experiencia previa puede ser lenta y ha influido de manera definitiva en las funcionalidades completadas en el *front end.*

Del mismo modo, la idea inicial de realizar dos aplicaciones con el mayor número de herramientas y lenguajes posibles, pero con unos requisitos y funcionalidades vagamente definidos, han afectado significativamente al resultado final.

En cuanto a la planificación y metodología, se puede considerar que han sido correctas y adecuadas, si bien las circunstancias especiales que se han vivido en los últimos meses con la crisis del COVID-19 han evitado que por primera vez en mi tiempo de estudio en la UOC haya podido presentar las PECs en las fechas propuestas.

Respecto a las líneas de trabajo futuro quedaría pendiente:

- Securizar API REST
- Securizar aplicación web
- Finalizar panel de control de administrador
- Implementar carro de compra y pasarela de pago en el área del cliente

7. Glosario

Back end: Parte del desarrollo web que se encarga de que toda la lógica de una página web funcione

Front end: Parte del desarrollo web consumida por el usuario y con la cual interacciona.

API REST: Interfaz de programación de aplicaciones que se apoya en la arquitectura REST para el desarrollo de aplicaciones en red

Arquitectura REST: es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web

Framework: Conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

URI: Acrónimo de Uniform Resource Identifier (en español identificador uniforme de recursos), que sirve para identificar recursos en Internet, precisamente lo que el nombre indica.

8. Bibliografía y webgrafía

[1] De Sanctis V. (2020) *ASP.NET Core and Angular 9.* Packt, Birmingham UK

[2] Aubry c. (2014) *HTML 5 y CSS para sitios con Diseño Web Responsive.* Eni Ediciones, Barcelona

[3] Krause J. (2016) *Introducing Bootstrap 4.* Apress, New York, NY

[4] Joshi B. (2019) *Beginning Database Programming Using ASP.NET Core 3*. Apress, Berkeley, CA

[5] Hobbs, S. (2019) *CORS Tutorial: A Guide to Cross-Origin Resource Sharing.* Recuperado de https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/ en 20/05/2020

Bouchefra A.(2019) 3+ *Ways to Add Bootstrap 4 to Angular 9/8 With Example & Tutorial.* Recuperado el 19/05/2020 de https://www.techiediaries.com/angular-bootstrap/

Trupja E. (2019) *Building Angular and ASP.NET Core Applications.* LinkedIn Learning.

Zanfir A (2019) B*uilding Applications with Angular, ASP.NET Core, and Entity Framework Core.* LinkedIn Learning

Wenz C (2020) *Building Web APIs with ASP.NET Core.* LinkedIn Learning

9. Anexos

Repositorio de aplicaciones:

- API REST: https://gitlab.com/borjaamontes/angular-app.git
- ANGULAR APP: https://gitlab.com/borjaamontes/angular-app.git