**NestJs Fundamentals**

**Habilidades y competencias que desarrollarás...**

* Inicialización de proyectos en node.
* Configuración básica de un proyecto.
* **ACTIVIDAD 01**
* Crear un proyecto en Nest JS bajo el nombre ecommerce-<usuario de github>.
* **ACTIVIDAD 02**
* Crear los módulos Products, Users y Auth.
* **ACTIVIDAD 03**
* Crear sus respectivos controllers y services.
* **ACTIVIDAD 04**
* Crear los endpoints GET /products, GET /users y Get /auth.
* **ACTIVIDAD 05**
* Crear un middleware global que loguee la ruta, método y la fecha-hora en que se llamó al endpoint.

**¡Bien hecho!**

**TIPS**

* Utiliza Nest CLI para inicializar el proyecto.
* Recuerda “modularizar” el código para trabajar de forma ordenada.

**NestJS Fundamentals 2**

**INSTRUCCIONES**

**Habilidades y competencias que desarrollarás...**

* Creación y uso de repositorios de NestJS.
* **ACTIVIDAD 01**
* Crear los repositorios para Users y Products.
* **ACTIVIDAD 02**
* Guardar las entidades en un array en memoria.
* **ACTIVIDAD 03**
* Cargar algunas entidades de prueba *hardcodeadas*, con las siguientes propiedades...
* **Users**
* id:number
* email: string
* name: string
* password: string
* address: string
* phone: string
* country?: string | undefined
* city?: string | undefined
* **Products**
* id:number
* name: string
* description: string
* price: number
* stock: boolean
* imgUrl: string
* **ACTIVIDAD 04**
* Modificar los endpoints GET /products y GET /users para que devuelvan el array de entidades.

**¡Bien hecho!**

**TIPS**

* Recuerda que los controllers sólo pueden comunicarse con el repositorio a través de los servicios.
* No olvides actualizar el array de providers .

**NestJS Routing**

**INSTRUCCIONES**

**Habilidades y competencias que desarrollarás...**

* Creación de rutas para el procesamiento de solicitudes correspondientes al CRUD de cada entidad.
* Validación del cuerpo de las solicitudes.
* Implementar la paginación de resultados de acuerdo a la información recibida en la ruta.
* Uso de guardias para la protección de rutas.
* **ACTIVIDAD 01**
* Crear todos los endpoints CRUD para Products y Users (GET, GET{id}, POST, PUT{id}, DELETE{id}).

**ACTIVIDAD 02**

Desarrollar la lógica de creación, listado y eliminación desde el repository.

* GET debe devolver la lista de elementos , y httpStatus = 200.
* GET{id} debe devolver el elemento con id pedido, y httpStatus = 200.
* En el caso de Users, no devolver el password.
* POST debe devolver el id de la entidad creada, y httpStatus = 201.
* PUT{id} y DELETE{id} pueden devolver el id de la entidad editada/eliminada y httpStatus 200.
* **ACTIVIDAD 03**
* Validar en POST y PUT que la estructura de la entidad corresponda a la estructura de cada entidad.

**ACTIVIDAD 04**

El método GET puede recibir como query params los valores page y limit.

* Si no recibe el parámetro page, el valor por defecto es 1.
* Si no recibe el parámetro limit, el valor por defecto es 5.
* Bonus: Implementar la lógica desde el repositorio, para paginar las entradas devuelvas.

**ACTIVIDAD 05**

Crear el endpoint POST /auth/signin, que reciba email y password.

* Para el login se utilizarán las credenciales email / password.
* Inyectar el usersRepository para poder hacer consultas.
* No se procederá al login si faltan alguna de las dos credenciales.
* No se procederá con el login en caso de que no exista un usuario registrado con la dirección de email proporcionada.
* En caso de que el usuario no exista o la contraseña proporcionada no coincida con la registrada, se deberá enviar una única respuesta para cualquiera de los casos. Ej: “Email o password incorrectos”. NOTA: Por seguridad es preferible no especificar cuál de los dos datos ha fallado en su verificación.

**ACTIVIDAD 06**

Dentro de la carpeta Auth, crear una guarda AuthGuard, que debe verificar lo siguiente...

* Debe existir un header Authorization.
* Dicho header, tiene que tener una estructura como la siguiente: Basic: <email>:<password>.
* NO validaremos por ahora que sea un email y un password válido, únicamente verificar si el header es enviado y continente un email y un password.
* Todos los endpoints de Users, salvo el POST, deben utilizar esta guarda.
* Todos los endpoints de Products, salvo el GET y el GET{id} deben utilizar esta guarda.

**¡Bien hecho!**

**TIPS**

* No te preocupes por ahora por el manejo de errores, la ruta de autenticación puede devolver strings únicamente.

**NestJS y TypeORM**

**ACTIVIDAD 01**

Instalar y configurar las librerías necesarias para utilizar TypeORM y postgres.

* Crear un archivo de configuración para la conexión con TypeORM.
* Los datos de conexión a la BD deben ser almacenados en variables de entorno.
* Crear el módulo de conexión  de manera global.

**ACTIVIDAD 02**

Definir las siguientes entidades de typeorm con sus respectivas relaciones.

**Users**

* id: debe ser un valor único generado automáticamente en formato UUID. No puede ser nulo y actúa como la clave primaria de la entidad.
* name: debe ser una cadena de texto de máximo 50 caracteres y no puede ser nulo.
* email: debe ser una cadena de texto de máximo 50 caracteres, único y no puede ser nulo.
* password: debe ser una cadena de texto de máximo 20 caracteres y no puede ser nulo.
* phone: debe ser un número entero.
* country: debe ser una cadena de texto de máximo 50 caracteres.
* address: debe ser un texto.
* city: debe ser una cadena de texto de máximo 50 caracteres.
* orders\_id: Relación 1:N con orders.

**Products**

* id: debe ser un valor único generado automáticamente en formato UUID. No puede ser nulo y actúa como la clave primaria de la entidad.
* name: debe ser una cadena de texto de máximo 50 caracteres y no puede ser nulo.
* description: debe ser un texto y no puede ser nulo.
* price: debe ser un número decimal con una precisión de 10 dígitos y una escala de 2 dígitos. No puede ser nulo.
* stock: debe ser un valor numérico. No puede ser nulo.
* imgUrl: debe ser una cadena de texto, en caso de no recibir un valor debe asignar una imagen por defecto.
* category\_id  (Relación 1:N).
* Relación N:N con orderDetails.

**Categories**

* id: debe ser un valor único generado automáticamente en formato UUID. No puede ser nulo y actúa como la clave primaria de la entidad.
* name: debe ser una cadena de texto de máximo 50 caracteres y no puede ser nulo.
* Relación: N:1 con products.

**Orders**

* id: debe ser un valor único generado automáticamente en formato UUID. No puede ser nulo y actúa como la clave primaria de la entidad.
* user\_id:  (Relación 1:N) con users.
* date.
* Relación 1:1 con orderDetails.

**OrderDetails**

* id: debe ser un valor único generado automáticamente en formato UUID. No puede ser nulo y actúa como la clave primaria de la entidad.
* price: debe ser un número decimal con una precisión de 10 dígitos y una escala de 2 dígitos. No puede ser nulo.
* order\_id: Relación 1:1 con orders.
* Relación N:N con products.

**ACTIVIDAD 03**

Al inicializar el servidor deben pre-cargarse a la base de datos las categorías y los productos del siguiente archivo a la base de datos. Ten en cuenta que la categoría debe ser cargada antes del producto.

* Vamos a suponer que del lado del cliente se ejecutará el endpoint /categories/seeder para realizar la carga de categorías al inicializar la aplicación así que puedes    contemplar este proceso como parte del flujo de un controlador normal. Lo mismo para la pre-carga de productos utilizando del endpoint /products/seeder (El orden de invocación es importante).
* Debes evitar la carga de registros bajo el mismo nombre tanto en categorías como en productos.
* Deberás crear el módulo, controlador, servicio y repositorio correspondiente para las categorías, este repositorio solo debe contener los métodos getCategories y addCategories.

[(opens in a new tab)](https://drive.google.com/drive/folders/15tc-nZjb2sXxpDEoBT94k-PxvVEt2GgA)

**ACTIVIDAD 04**

Crear el modelo,controllador, servicio y repositorio para las órdenes de compra (orders), dentro de este repositorio crearemos la lógica necesaria para que un usuario pueda realizar una compra de un “carrito de productos”.

* La orden de compra será recibida mediante una solicitud de HTTP Post al endpoint /orders cuyo cuerpo tendrá la siguiente estructura:

{

  "userId":"UUID del usuario",

  "products":[

     {

   "id":"UUID producto 1"

  },

  {

    "id":"UUID producto 2"

  }

    ]

}

* Por ahora los usuarios solo pueden agregar una unidad de cada producto dentro de su carrito.
* En el repositorio de orders tendrás que crear 2 métodos diferentes getOrder y addOrder.

**addOrder**

* Busca a un usuario por id.
* Crea un registro en la tabla orders con el usuario encontrado.
* Busca los productos por id recibidos en la request actualizando el total de la compra y reduciendo el stock del producto. correspondiente. (al realizar la búsqueda de todos los productos aquellos con stock igual a 0 no deben ser mostrados).
* Construye y registra un detalle de compra con los productos seleccionados.
* Devuelve la orden de compra con el precio y id del detalle de compra.

**getOrder**

* Busca una orden recibida por id.
* Devuelve un objeto con la orden y los detalles de la orden (el detalle de la orden debe contener un array con todos los productos adquiridos).

**ACTIVIDAD 05**

* Modificar los el contenido del repositorio para que utilice la entidad Users para la gestión de información. En el caso de la búsqueda por Id la respuesta debe devolver al usuario incluyendo un array con las órdenes de compras efectuadas  (únicamente id y date).

**ACTIVIDAD 06**

* Configurar las migraciones correspondientes y los comandos en el package.json para ejecutarlas.

**¡Bien hecho!**

* **TIPS**
* Tendrás que hacer algunos cambios en controladores y servicios para que funcione correctamente la aplicación en conjunto con la DB. Puedes almacenar el archivo de productos en un json dentro de la carpeta src de tu proyecto.

**NestJS File Upload**

**ACTIVIDAD 01**

Configurar una cuenta en Cloudinary y generar las credenciales de acceso correspondientes.

**ACTIVIDAD 02**

Crear la conexión dentro de la aplicación mediante un archivo de configuración.

**ACTIVIDAD 03**

Crear el módulo, servicio, controlador y repositorio correspondientes para la gestión de archivos.

**ACTIVIDAD 04**

Desarrollar la lógica para la carga de imágenes a Cloudinary y la actualización de imágenes de los productos en la DB. Este proceso será realizado por medio del endpoint  /files/uploadImage/:id  que recibe por parámetros el id del producto cuya imagen queremos actualizar y el archivo a emplear en el cuerpo de la solicitud.

**ACTIVIDAD 05**

La DB debe reflejar los cambios efectuados en el campo imgUrl

**ACTIVIDAD 06**

Implementar pipes para la validación del tamaño de imagen (no mayor a 200kb) así como los tipos de imagen permitidos.

**NestJS Auth**

**ACTIVIDAD 01**

**Sign Up**

* Sustituir el endpoint POST /users por el endpoint POST /auth/signup que será creado dentro del controlador de autenticación.
* Este endpoint recibirá la misma estructura que recibia el endpoint anterior y adicionalmente recibirá una propiedad de confirmación de contraseña, debes validar que ambas contraseñas sean recibidas y coincidan o devolver una excepción.
* Debe registrar al usuario dentro de la base de datos con una contraseña hasheada
* Debe retornar al usuario sin contraseña

**ACTIVIDAD 02**

**Sign In**

* Modificar la funcionalidad de signIn para que valide el password encriptado con el provisto en la solicitud.
* Enviar un error genérico en caso de existir algún error ya sea por que el usuario no es encontrado o por que el password es incorrecto
* Crear un token de acceso para el usuario registrado con una validez de 1 hora

**ACTIVIDAD 03**

**Auth Guard**

* Modificar la funcionalidad del guardián de autenticación para la validación de tokens.
* Enviar un error en caso de no recibir el token o en caso de que este o sea un token válido con código de error 401
* El token debe ser verificado por medio de una clave secreta que no debe ser mostrada directamente en el código (Variables de entorno).
* Una vez validado el token debes adjuntar la información correspondiente al tiempo de expiración de dicho token

**ACTIVIDAD 04**

Los endpoints protegidos por este guardián serán los siguientes

* POST /uploadImage/:productId
* POST /orders
* GET /orders/:id
* PUT /products/:id
* GET /users
* GET /users/:id
* PUT /users/:id
* DELETE /users/:id

**¡Bien hecho!**

**TIPs:**

* Recuerda modificar el DTO para la creación de usuarios.
* Puedes utilizar decoradores personalizados para la validación.

**NestJS Auth II**

**ACTIVIDAD 01**

Definir un guardián para la validación del rol de administrador en usuarios para el control de acceso.

**ACTIVIDAD 02**

Modificar Entidad y Dtos para implementar el campo de administrador:

* Todos los registros serán considerados usuarios por default.
* El campo admin no debe ser recibido dentro de la solicitud.
* El campo admin no debe ser mostrado en las rutas que devuelven un usuario (únicamente en la ruta GET /users/).

**ACTIVIDAD 03**

* Definir los roles de la aplicación (únicamente es necesario el rol de administrador).

**ACTIVIDAD 04**

* Asignar y verificar de rol junto con el proceso de firma de JWT.

**ACTIVIDAD 05**

Implementar de control de Acceso en las rutas:

* GET /users/
* PUT /products/:id

**TIPs:**

* No olvides utilizar un custom decorator para la definición de roles.

**NestJS Testing**

**ACTIVIDAD 01**

Crear e implementar pruebas unitarias en los diferentes módulos de la aplicación.

**ACTIVIDAD 02**

Crear e implementar pruebas de integración en la aplicación.

**TIPs:**

* La implementación de pruebas es un extra credit para el proyecto integrador.

**NestJS Open API Integration**

**Habilidades y competencias:**

Lograrás crear la documentación correspondiente para la aplicación utilizando swagger y Open API.

**Actividad 1**

* Integrar Swagger a la aplicación para la generación dinámica de la documentación en la ruta /API.
* Cada controlador debe tener su propia etiqueta para facilitar la legibilidad.

**Actividad 2**

* Mantener la protección de rutas que utilicen JWT, las rutas con control de acceso mediante roles únicamente pueden ser testeadas para validar errores.
* Los DTOs y entidades deben estar detallados en la documentación
* Las pruebas de la aplicación de forma integral en la interfaz de Open API deben ser funcionales.

## TIPs

Puedes utilizar el formato con comentarios o decoradores para la definición y personalización de los DTOs y entidades.

**NestJS Docker**

**HOMEWORK**

**Habilidades y competencias ha desarrollar:**

* Creación de imágenes y componentes de Docker.
* Uso de docker compose para el uso local de una aplicación.

## ACTIVIDAD 01

Definir la imagen de la aplicación dentro de un archivo docker.

**ACTIVIDAD 02**

Crear un archivo docker-compose para utilizar la aplicación en conjunto con una base de datos dockerizada.

**ACTIVIDAD 03**

Asegurar la persistencia de datos del contenedor.

**NestJs Deployment**

**Habilidades y competencias ha desarrollar:**

* Publicación de API en una plataforma de deployment.

## ACTIVIDAD 01

Conectar el repositorio de github a una plataforma de deployment para que tanto la API como la DB sean accesibles desde cualquier ubicación.

**ACTIVIDAD 02**

El deployment debe ser realizado desde una imagen de docker, puedes elegir cualquier plataforma de tu elección.

**TIPs**

* Te recomendamos utilizar Render ya que simplifica mucho el proceso y es una plataforma gratuita.