



Team Lead: Nicolás Picón

# Prueba de desempeño Node.js

## Propósito:

Como equipo de formación Riwi, queremos evaluar las habilidades de desarrollo en Node.js y TypeScript de un aspirante a Junior Developer, específicamente en la implementación de soluciones back-end. Esta prueba permitirá determinar si el candidato posee los conocimientos técnicos y la capacidad de resolver problemas necesarios para contribuir efectivamente en proyectos reales y entornos laborales.

## Metodología:

La prueba técnica se estructurará como una historia de usuario Scrum con una duración estimada de 8 horas. Esta metodología ágil permite simular un entorno de trabajo realista, donde los desarrolladores reciben tareas con plazos definidos y deben gestionar su tiempo y recursos de manera eficiente. No se permitirá el uso de IA, verificando así la apropiación real de conocimientos sobre los elementos vitales del lenguaje de programación que no se deberían delegar a sistemas externos. Las herramientas permitidas son el editor de código, Postman, el navegador para probar el API, notas tomadas en clase, documentación oficial de internet, el terminal para uso de Git y demás herramientas locales. Otras herramientas que se requieran utilizar deberán ser bajo aprobación del TL.

La prueba se centrará en la capacidad del candidato para aplicar los conocimientos teóricos en la práctica, resolver problemas de manera creativa y demostrar un dominio sólido de las tecnologías y conceptos fundamentales de



Node.js y TypeScript. Esta metodología no solo evalúa las habilidades técnicas del candidato, sino también su capacidad de adaptación y gestión del tiempo, cualidades esenciales para un Junior Developer exitoso.

## Historia de usuario (Enunciado):

¡Coder, bienvenido a tu prueba de desempeño de Node.js y TypeScript!

- **Las reglas:**

- Tienes 8 horas máximo para la culminación de la prueba. (2pm a 10pm) con 3 descansos (04:00 pm a 04:20 pm, 06:00 pm a 06:20 pm y 08:00 pm a 08:20 pm)
- **Debes tener la capacidad de explicar tu código.**
- Debes realizar la prueba de forma individual.
- Se deben cumplir los criterios de aceptación de la HU.
- No se permite uso de ninguna inteligencia artificial.
- No se permite uso de computador personal.
- Revisa las rúbricas de evaluación para que entiendas en que debes enfocarte a la hora de implementar tu solución y que se te evaluará y determinará tu puntaje final (El TL, te informará donde las puedes consultar)

- **Las metodologías:**

- Trabajarás en una historia de usuario similar a un entorno de trabajo real.
- Demuestra tus conocimientos técnicos y tu capacidad para resolver problemas.
- **¡Buena suerte y manos a la obra!**



- **Historia de usuario:**

**Empresa:** E-comFast

**Contexto:** Empresa dedicada a la venta de productos en línea necesita implementar una API RESTful para gestionar productos, carritos de compra, usuarios, roles y pedidos.

**Tarea asignada:**

Como desarrollador junior de una plataforma de E-commerce, deseo implementar las funcionalidades de una API RESTful considerando la gestión de permisos definida en la sección “*Gestión de permisos*” de este mismo documento.

- Órdenes
  - Creación de órdenes
  - Actualización de órdenes
  - Eliminación de una orden
  - Obtención de órdenes
  - Obtener todas las órdenes realizadas por ID de usuario
- Productos
  - Creación de productos
  - Actualización de productos y su stock
  - Eliminación de productos
  - Obtención de todos los productos
  - Obtener todos los productos por ID de orden
- Usuarios
  - Creación de usuarios
  - Eliminación de usuarios
  - Actualización de usuarios
  - Obtención de todos los usuarios
- Carrito de compras



- o Agregar productos al carrito de compras
  - o Eliminar productos del carrito de compras
  - o Actualizar cantidad de producto en el carrito de compras
- Implementación de Autenticación con email y contraseña usando el estándar JWT

### Plus (Opcional)

- Encriptación de contraseña en creación de usuarios

### Especificaciones técnicas:

- Descripción de entidades
  - o **Product**: id (PK), name (Varchar (200)), price (DECIMAL (10, 2)), description (TEXT), stock (INT)
    - *Descripción: Producto no inventariado de la tienda*
  - o **User**: id (PK), email (Varchar (200)), password (Varchar (200)), roleId (FK)
    - *Descripción: Usuario cliente y admin de la app*
  - o **Role**: id (PK), name (Varchar (200))
    - *Descripción: Se manejan dos roles:*
      - Admin
      - Client
  - o **Cart**: id (PK), userId (FK)
    - *Descripción: Carrito de compras generado por cada usuario al iniciar una orden de pedido.*
  - o **ProductCart**: id (PK), cartId (FK), productId (FK), quantity (INT)



- *Descripción: Relación de muchos a muchos entre productos y carrito de compras.*
- o **Order:** id (PK), UserId (FK), ProductCartId (FK), total (DECIMAL (10, 2)).
  - *Descripción: Descripción de la orden de compra del usuario de tipo Cliente.*
- o **Entities:** id (PK), name (Varchar (200)).
  - *Descripción: Entities es una entidad que alberga las entidades que involucran la lógica de negocio. Para propósitos de esta prueba se recomienda quemar 2 entidades:*
    - Order
    - User
- o **Permissions:** id (PK), roleId (FK), entityId (FK), canCreate (boolean), canUpdate (boolean), canDelete (boolean), canGet (boolean)
  - *Descripción: Esta entidad es fundamental para la administración de accesos, gestionando los permisos asociados a cada usuario y su rol. Más información en la sección de permisología.*

### Relaciones:

- **Product ↔ ProductCart:** Un producto puede estar en muchos carritos, y un carrito puede tener muchos productos.
- **User ↔ Role:** Un usuario tiene un rol, y un rol lo pueden tener muchos usuarios.



- **User ↔ Cart:** Un usuario tiene un carrito, y un carrito pertenece a un usuario.
- **Cart ↔ ProductCart:** Un carrito puede tener muchos productos, y un producto puede estar en muchos carritos.
- **Order ↔ User:** Un pedido pertenece a un usuario, y un usuario puede tener muchos pedidos.
- **Order ↔ ProductCart:** Un pedido tiene un producto de carrito, y un producto de carrito puede estar en muchos pedidos.

## Gestión de permisos

La gestión de permisos es crucial para la administración de accesos y determina qué acciones pueden realizar los usuarios en diferentes entidades según su rol. Aquí se describen los permisos para los roles "Admin" y "Client" en las entidades "Order" y "User".

### 1. Admin:

#### o User:

- Crear (canCreate): Sí
- Actualizar (canUpdate): Sí
- Eliminar (canDelete): Sí
- Consultar (canGet): Sí

### 2. Client:

#### o Order:

- Crear (canCreate): Sí
- Actualizar (canUpdate): Si
- Eliminar (canDelete): No
- Consultar (canGet): Sí

**Tiempo:** 8 horas

Se recomienda distribuir el tiempo adecuadamente.



## Criterios de Aceptación:

- Uso de Estilo de Arquitectura de Transferencia de estado representacional:
  - Utilizar adecuadamente la creación y estructuración de endpoints en lo que respecta a:
    - Nombrado de endpoints en inglés y en plural por recurso.
    - El no uso de verbos en la estructura de la URI.
    - Jerarquía de recursos, como se ilustra en el siguiente ejemplo:

```
/users           - colección de usuarios
/users/{userId}   - un usuario específico
/users/{userId}/orders - colección de pedidos de un usuario específico
/orders          - colección de pedidos
/orders/{orderId} - un pedido específico
```

- Control adecuado de errores:
  - Validar las respuestas de la API y mostrar efectivamente mensajes al usuario, además controlar posibles errores y avisar al usuario.
- Documentación:
  - Incluir un archivo *README.md* que explique cómo configurar y ejecutar (Levantar) el proyecto.



- Cada una de las entidades o clases base, deben llevar su correspondiente comentario de tipo documentación js docs, a continuación, un ejemplo que ilustra el criterio:

```
/*
 * Clase que representa un Producto en la tienda.
 */
...

class Product {
    /**
     * Crea un nuevo Producto.
     * @param {number} id - El ID del producto.
     * @param {string} name - El nombre del producto.
     * @param {number} price - El precio del producto.
     * @param {string} description - La descripción del producto.
     * @param {number} stock - La cantidad en stock del producto.
    */
    constructor(id, name, price, description, stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.description = description;
        this.stock = stock;
    }
}
```

- Git-flow:
  - Implementaremos un Git-flow bastante sencillo, el cual consta de 3 ramas:
    - Rama *master*
    - Rama *develop*
    - Ramas feature para cada funcionalidad i.e Si te creas un CRUD de usuarios, feature/users#C.C en donde C.C es el número de tu cédula de ciudadanía o documento de identidad.



- Uso de ORM
  - Se evaluará el correcto funcionamiento de las entidades creadas para interactuar con los datos utilizando Sequelize como ORM
- Uso de sintaxis adecuada de NodeJS y Typescript:
  - En su mayoría, esta prueba debe ser realizada utilizando el paradigma orientado a objetos con Clases.
  - Todas las variables, funciones deben estar tipadas. Todas las clases deben implementar su respectiva *interface*.
  - Utilizar express como servidor web y enrutador

### Entregables:

- Código fuente del proyecto en un repositorio GitHub. Adjuntar url del repositorio (El cual debe estar público) y el código en ZIP en Moodle. (*solo se revisan ramas con commits que máximo tengan la fecha y hora de finalización de la prueba*) (*proyectos que contengan el folder node\_modules sea en .zip o en el repo de GitHub NO SE CALIFICARÁN y su nota automáticamente será 0*)



### **Notas Adicionales:**

Se valorará la claridad y organización del código, así como la correcta implementación de las herramientas y prácticas solicitadas.