# **Trabajo Práctico N°1 – PARTE I**

**Criterios de evaluación:**

**Funcionalidad y Requisitos:**

● Completar todas las tareas requeridas.

● Cumplir con los requisitos específicos del enunciado.

● Manejar adecuadamente casos extremos y situaciones inusuales.

**Diseño y Estructura del Código:**

● Organizar el código en funciones/módulos coherentes.

● Mantener un código legible y bien comentado.

● Utilizar nombres descriptivos para variables y funciones.

● Seleccionar y emplear las estructuras de datos adecuadas.

● Dividir el código en módulos reutilizables.

**Pruebas y Validación:**

● Incluir casos de prueba relevantes y abarcar diversos escenarios.

● Obtener resultados coherentes con los esperados.

● Manejar errores de manera adecuada y sin fallos inesperados.

**Creatividad y Mejoras:**

● Agregar características adicionales que demuestren comprensión

profunda.

● Contribuir con soluciones innovadoras o enfoques únicos.

**Documentación:**

● Proporcionar instrucciones claras para compilar, ejecutar y usar el

programa.

● Acompañar el código con comentarios explicativos que aclaren su

funcionamiento.

**Formato de entrega:**

● Código: se realizará a través de un repositorio de código en GitHub,

cuyo nombre deberá seguir el siguiente formato:

“nombre\_del\_alumno-trabajo\_practico\_1-backend”.

● Documento: se realizará en formato PDF (sin excepción) y con la fuente

'Times New Roman', tamaño N° 12 para los párrafos, N° 16 para los

títulos y se permiten negritas e itálicas.

Extras:

● El trabajo es de modalidad individual (solamente una persona puede

participar en la creación del trabajo y debe ser la persona responsable de

entregar dicho trabajo en el aula virtual).

● Aquellos trabajos donde se demuestre una clara complicidad entre 2 o

más involucrados terminará en la descalificación del trabajo práctico.

Consigas:

● **Crear un servidor http** que el cual permita hacer las 5 operaciones básicas sobre una entidad en particular de las que van a estar disponibles más adelante.

● **Además, el servidor** deberá tener la capacidad de registrar usuarios. La entidad seleccionada debe tener una relación de uno a muchos con la entidad de usuarios, lo que significa que un usuario podrá tener varios registros de la entidad seleccionada. Por lo tanto, si no hay un usuario previamente registrado,

no se podrán crear registros de la entidad seleccionada.

● **Los campos de entrada** del servidor deben estar validados previamente antes de llegar a los controladores.

● **Aplicar** la división de carpetas y archivos correspondiente a cada módulo con la modularización correspondiente aplicando el patrón de diseño MVC.

● **Presentar un documento técnico** en el que se detalle la elección de la temática elegida, el porqué de dicha elección y donde también se grafique con un formato de red conceptual la lógica que se emplea en dicha temática.

● **El documento** deberá contener además un diagrama con las tablas y relaciones se utilizarán en la base de datos.

● **El trabajo** (código) deberá coincidir concretamente en la lógica y aplicación de los pasos con lo que se haya graficado en el punto anterior.

● **El trabajo** (código) deberá obligatoriamente utilizar un sistema ORM (como Sequlize) para crear las consultas de la base de datos, la cuál, deberá ser MySQL o PostgreSQL de forma obligatoria sin excepciones.

● **El trabajo (código)** debe presentar la sucesión de commits aplicados en el repositorio de código, con los comentarios correspondientes a cada una de las etapas de su desarrollo. Se recomienda (aunque no es obligatorio) seguir la

convención de commits que se presenta en este

post:

<https://www.conventionalcommits.org/es/v1.0.0/>

**Temática a elección:**

***Aclaraciones:***

● Las temáticas son de elección del alumno, pero se deben respetar rigurosamente los requisitos de cada posibilidad presentada. Esto implica que no se respetarán tanto las cardinalidades como las funcionalidades mínimas de cada una de las opciones, y estas deberán presentarse de esta manera tanto en los gráficos solicitados en la sección de documentación, como también deberán quedar reflejadas en el código.

● En todos los casos, los atributos de deben tener cada una de las entidades no será especificada pero será tenido en cuenta la capacidad del alumno de abstraer aquellas propiedades que crea necesaria en cada caso para con la nota final.

**Temáticas:**

**Temática 1:** “Usuario - Producto - Comentario”

Cada **“Usuario”** deberá poder tener uno o más **“Productos”** por ende deberá

de haber una cardinalidad de uno a muchos y que a su vez sea capaz de tener

uno o más **“Comentarios”**, por ende también deberá estar relacionada con una

cardinalidad de uno a muchos con cada comentario.

**Temática 2:** "Usuario - Post - Comentario"

Cada **"Usuario"** puede crear uno o más **"Posts"**. Existe una relación de uno a

muchos entre **"Usuario"** y **"Post"**. Cada **"Post"** puede tener varios

**"Comentarios"**, lo que implica otra relación de uno a muchos entre **"Post"** y

**"Comentario"**.

**Temática 3:** "Usuario - Proyecto - Tarea"

Cada "Usuario" puede estar asociado con uno o más **"Proyectos"**. Esto

establece una relación de uno a muchos entre **"Usuario"** y **"Proyecto"**. Cada

**"Proyecto"** puede tener varias **"Tareas"**, lo que implica una relación de uno a

muchos entre **"Proyecto"** y **"Tarea"**.

**Temática 4:** "Usuario - Evento - Asistencia"

Cada **"Usuario"** puede crear uno o más **"Eventos"**. Existe una relación de uno

a muchos entre **"Usuario"** y **"Evento"**. Cada "Evento" puede tener múltiples

**"Asistencias"**, lo que establece una relación de uno a muchos entre **"Evento"**

y **"Asistencia"**.

**Temática 5:** "Usuario - Playlist - Canción"

Cada **"Usuario"** puede crear una o más **"Playlists"**. Existe una relación de

uno a muchos entre **"Usuario"** y **"Playlist"**. Cada **"Playlist"** puede contener

varias **"Canciones"**, lo que implica una relación de uno a muchos entre **"Playlist"** y **"Canción"**.

**Trabajo Práctico N°1 – Documento Técnico**

**Tema: "Usuario - Playlist - Canción"**

**Estructura del Código y Organización**

Este documento tiene como objetivo proporcionar una descripción de la aplicación desarrollada, basada en la temática "Usuario - Playlist - Canción".

Se explicarán los conceptos clave, la lógica de funcionamiento y la estructura del código. Además, se presentarán los aspectos de diseño, validación y persistencia de datos utilizados en la implementación.

**Funcionalidad y Requisitos**

El servidor cumple con una serie de requisitos y funcionalidades:

1. Registro de Usuarios: Los usuarios pueden registrarse proporcionando un apodo (nickname) y una contraseña. Cada usuario registrado tiene la capacidad de crear y administrar sus propias listas de reproducción.
2. Operaciones CRUD en Playlists: Los usuarios pueden realizar las operaciones básicas (Crear, Leer, Actualizar, Eliminar) en las listas de reproducción. Esto les permite crear nuevas listas, ver información detallada de una lista específica, actualizar detalles y eliminar listas.
3. Relación Usuario - Playlist: Cada usuario puede tener varias listas de reproducción. Esta relación de "uno a muchos" permite que los usuarios administren múltiples listas.
4. Operaciones CRUD en Canciones: Dentro de cada lista de reproducción, los usuarios pueden gestionar las canciones asociadas. Pueden crear nuevas canciones, ver la lista de canciones disponibles y obtener detalles específicos de una canción.
5. Relación Playlist - Canción: Cada lista de reproducción puede contener múltiples canciones. Esta relación de "uno a muchos" permite que una lista almacene varias canciones.
6. Validación de Datos: Antes de que los datos lleguen a los controladores, se realiza una validación. Esto garantiza que los datos ingresados cumplan con los requisitos mínimos y evita errores en la base de datos.

**Modelos**

* **Usuario**: Contiene los datos de los usuarios registrados, como su nombre de usuario y contraseña. Además, se establece una relación uno a muchos con las listas de reproducción que un usuario puede crear.
* **Playlist**: Almacena información sobre las listas de reproducción creadas por los usuarios. Cada lista de reproducción tiene un nombre y puede contener múltiples canciones. Existe una relación muchos a uno con el usuario que creó la lista y una relación uno a muchos con las canciones que contiene.
* **Canción**: Guarda los detalles de las canciones contenidas en las playlists.

**Controladores**

* **Crear Usuario**: La función **crearUsuario** valida los datos recibidos y crea un nuevo usuario en la base de datos.
* **Obtener Usuario**: La función **obtenerUsuario** busca y devuelve información sobre un usuario específico por su ID.
* **Obtener Usuarios**: La función **obtenerUsuarios** recupera todos los usuarios registrados en la base de datos.
* **Crear Playlist**: La función **crearPlaylist** valida y crea una nueva lista de reproducción para un usuario. Se verifica si el usuario existe antes de crear la lista.
* **Obtener Playlists**: La función **obtenerPlaylists** obtiene todas las listas de reproducción disponibles, incluyendo las canciones asociadas.
* **Obtener Playlist**: La función **obtenerPlaylist** obtiene detalles sobre una lista de reproducción específica, incluyendo sus canciones.
* **Actualizar Playlist**: La función **actualizarPlaylist** permite actualizar los datos de una lista de reproducción existente.
* **Eliminar Playlist**: La función **eliminarPlaylist** elimina una lista de reproducción por su ID.
* **Crear Canción**: La función **crearCancion** valida y crea una nueva canción asociada a una lista de reproducción.
* **Obtener Canciones**: La función **obtenerCanciones** recupera todas las canciones disponibles, incluyendo los detalles de la lista de reproducción.
* **Obtener Canción**: La función **obtenerCancion** obtiene detalles sobre una canción específica por su ID.

**Validación de Datos y Middleware**

Se implementa la validación de datos en las rutas utilizando el módulo Express Validator. Antes de llegar a los controladores, los datos de entrada se validan según los esquemas definidos para cada entidad. Si hay errores de validación, se devuelve una respuesta con un código de estado 400 y los detalles de los errores.

**Relaciones de Bases de Datos**

Se establecen relaciones entre las entidades Usuario, Playlist y Canción utilizando el sistema ORM Sequelize. Estas relaciones permiten asociar usuarios con listas de reproducción y canciones, y listas de reproducción con canciones.

**Uso de Sequelize y Base de Datos**

Se utiliza Sequelize como sistema ORM para interactuar con la base de datos. Los modelos definidos en Sequelize representan las tablas de la base de datos. Las operaciones CRUD se realizan utilizando métodos proporcionados por Sequelize, como **create**, **findOne**, **findAll**, **update** y **destroy**.

**Documentación**

El código está acompañado de comentarios explicativos que detallan el funcionamiento de cada función y bloque de código. Además, se incluyen descripciones en los nombres de variables y funciones para mejorar la legibilidad del código.

**Diagrama de entidades:**

A continuación, se presenta un diagrama básico que representa visualmente las entidades involucradas y como se relacionan.

