

Ejercicios semanales

El objetivo de estos ejercicios es practicar lo visto durante cada semana del módulo, en total debe haber 5 ejercicios, **se debe entregar al menos 3 ejercicios** por el medio especificado por el instructor para pasar el módulo, este es un requisito obligatorio.

Los ejercicios deben ser entregados a mas tardar al día y hora especificada por el instructor, no hay excepciones. Hay un margen de tolerancia de 30 min, es decir, si el ejercicio se debe entregar un día a las 5 pm, el alumno tiene hasta las 5:30 pm para seguir siendo considerado como valido su ejercicio; pero esto solo se aplica hasta 2 veces, significando que el alumno puede entregar su tarea con este margen extra de tiempo 2 veces.

Se deben seguir las instrucciones de código limpio, para ello, el instructor les proporcionara una configuración de Prettier para que la agreguen a sus ejercicios.

El ejercicio se considera invalido si:

- Se entrego tarde, pasando el margen permitido o si es la tercera vez que se entrega tarde.
- Si el ejercicio no cumple con lo especificado por el instructor.
- Si gran parte del proyecto no cumple con la configuración de Prettier, esto para enforzar el tener el código organizado

Semana 1:

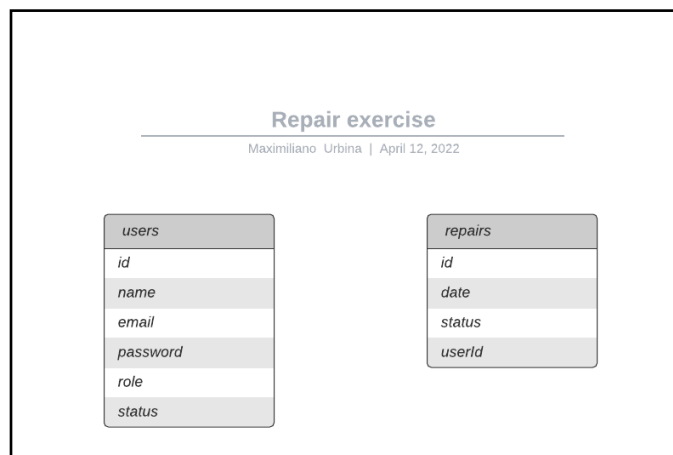
Temas:

- Express
- Routers
- Sequelize
- Modelos

“Trabajamos en una empresa que se encarga de la reparación de computadoras, donde el usuario puede agendar una cita para que pueda dejar su equipo de cómputo, y el personal de la empresa actualizara cuando este listo el equipo del cliente.”

La empresa nos contrato como desarrolladores de backend para crear una API que sea capaz de poder agendar citas para los usuarios, y que los empleados puedan actualizar cuando los equipos ya estén listos para que los recojan”

- Inicializar un proyecto con *npm* (para poder instalar paqueterías)
- Instalar *express* y *nodemon* (la ultima como dependencia de desarrollo)
- Crear un servidor de *express*, corriendo en el puerto de su gusto
- Conectarse a su base de datos en PostgreSQL
- Crear los siguientes modelos basados en el siguiente diagrama:
 - a. Para el status del modelo **user**, asignar como valor default **available**.
 - b. Para el status del modelo **repairs**, asignar como valor default **pending**.



- Definir los siguientes endpoints

<i>/api/v1/users</i>		
<i>HTTP VERB</i>	<i>URL</i>	<i>Descripción</i>
GET	/	Obtener la lista de los usuarios en la base de datos
GET	/:id	Obtener un solo usuario dado un id
POST	/	Crear un nuevo usuario, se debe proporcionar por el req.body (name, email, password, role), el role (rol) puede ser <i>client</i> o <i>employee</i>
PATCH	/:id	Actualizar los datos de un usuario dado un id, solo puede actualizar su <i>name</i> y <i>email</i>
DELETE	/:id	Deshabilitar la cuenta de un usuario

Para los endpoints con */:id*, se debe validar que el usuario exista, en caso de que no, enviar mensaje de error

NOTA: De momento, no se preocupen por la parte de validación de roles o inicio de sesión, esto se cubrirá en el futuro.

<i>/api/v1/repairs</i>		
<i>HTTP VERB</i>	<i>URL</i>	<i>Descripción</i>
GET	/	Obtener la lista de equipos pendientes (pending) de reparar
GET	/:id	Obtener un equipo pendiente de reparar por su id
POST	/	Crear una cita, se debe incluir en el req.body lo siguiente (date, userId) El userId siendo el id del usuario quien solicita la reparación.
PATCH	/:id	Actualizar el status de una reparación ha completado (cambiar status a completed)
DELETE	/:id	Cancelar la reparación de un usuario (cambiar status a cancelled)

Para los endpoints con */:id*, se debe validar que el servicio exista con status **pending**, en caso de que no, enviar mensaje de error.

Semana 2:

Temas:

- Middlewares
- Express-validators
- Error handling
- Variables de entorno
- Relaciones

Basado en el ejercicio de la semana pasada, nuestro senior developer nos hizo el siguiente comentario:

“Muy buen trabajo con tu solución de la semana pasada, estuve revisando tu código y creo que podemos optimizarlo al crear middlewares para validar si los usuarios existen o no, ya que esa lógica se puede reutilizar.

Debemos actualizar los modelos, te adjunto una imagen para que veas los cambios que hay que realizar, y hay que establecer una relación entre estos 2 modelos, recuerda que un usuario puede tener muchos servicios (repairs) y un servicio le pertenece a un usuario.

Nuestros empleados también me comentaron que no hay una validación muy eficiente a la hora de levantar servicio o al crear usuarios, debemos implementar una mejor validación, creo que había escuchado de express-validators

Y debemos implementar una mejor manera para el manejo de errores, al menos para facilitar el que podamos enviar y leer los errores.

También nuestros empleados nos pidieron lo siguiente, que al buscar los servicios pendientes y los completados, podemos adjuntar la información del usuario que solicito este servicio.

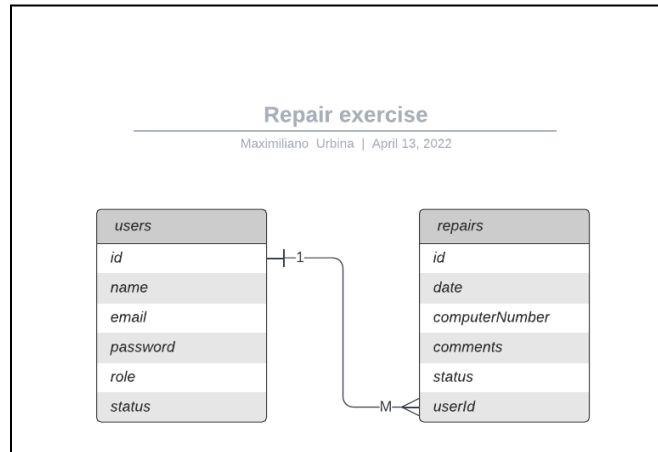
Antes de que se me olvide, asegúrate de poner nuestras credenciales de nuestra base de datos en un archivo de variables de entorno, pero estoy seguro que ya hiciste eso.”

Se deben crear 2 middlewares que cumplan con los especificado:

- Validar que el usuario existe dado un ID, en caso de que no, enviar mensaje de error

- Validar que un servicio pendiente (status **pending**) exista, en caso de que no, enviar mensaje de error

Implementa los siguientes cambios en los modelos, y establece la relación entre ambos



Instala *express-validator* y úsalo para validar los siguientes campos antes de crear un usuario o servicio:

Modelo	Campos
Users	Name, email, password
Repairs	Date, computerNumber, comments

Instalar *dotenv* y aplicarlo para usar variables de entorno para nuestras credenciales de nuestra base de datos.

Utilizar lo visto para aplicar *error handling* en nuestro proyecto (catchAsync, globalErrorHandler y AppError) y optimizar nuestro código.

Semana 3:

Temas:

- Encriptar contraseñas
- JWT
- Niveles de acceso
- Enviar JWT a React

“Hola de nuevo, vimos que haz hecho un buen trabajo con las validaciones y las relaciones entre los modelos, para que esa parte ya quedo lista por el momento. Ahora debemos asegurarnos que nuestros usuarios se sientan seguros al tener una cuenta con nosotros.

Por favor, asegúrate de encriptar las contraseñas de los usuarios, como ya sabrás, estas no pueden ser contraseñas planas, debemos hacerlas ilegibles para cualquier usuario que quiera robarnos nuestra información.

Junto con las contraseñas, por favor implementa un endpoints para que el usuario tenga que iniciar sesión, y regrésale un JWT para que pueda acceder a nuestros demás endpoints, solo los usuarios autenticados pueden hacer uso de nuestra app.

Junto con eso, implementa un middleware para validar el rol del usuario, ya que los empleados de nuestra compañía deberían ser los únicos para realizar ciertas acciones en la app, mas adelante te especifico los endpoints que necesito que protejas.

También, implementa otro middleware para asegurarte que para editar o deshabilitar una cuenta de un usuario, únicamente el dueño de esa cuenta puede hacer y no un tercero, ejemplo, usuario 1 solo puede editar o deshabilitar su cuenta, pero usuario 2 no puede editar o deshabilitar la cuenta del usuario 1 o de cualquier otro excepto la suya.

Por último, el equipo de front-end necesitan que los ayudes a implementar estas nuevas cosas, es decir, les podrías ayudar a que se envíe el JWT a tu back.”

Instala **bcrypt** para encriptar las contraseñas

Implementa el siguiente endpoint

<i>/api/v1/users</i>		
<i>HTTP VERB</i>	<i>URL</i>	<i>Descripción</i>
POST	/login	Recibir por req.body (email, password), y validar que los datos sean validos

Este endpoint debe regresar un JWT con el id del usuario como *payload*, posteriormente utilizarlo para validar la sesión del usuario.

Los siguientes endpoints deben ser protegidos para que únicamente el usuario con rol “**employee**” pueda hacer peticiones a estos:

<i>/api/v1/repairs</i>		
<i>HTTP VERB</i>	<i>URL</i>	<i>Descripción</i>
GET	/	Obtener la lista de equipos pendientes (pending) de reparar
GET	/:id	Obtener un equipo pendiente de reparar por su id
PATCH	/:id	Actualizar el status de una reparación ha completado (cambiar status a completed)
DELETE	/:id	Cancelar la reparación de un usuario (cambiar status a cancelled)

Estos endpoints se deben proteger para que el dueño de la cuenta que se intenta actualizar o deshabilitar pueda realizar dichas acciones:

<i>/api/v1/users</i>		
<i>HTTP VERB</i>	<i>URL</i>	<i>Descripción</i>
PATCH	/:id	Actualizar los datos de un usuario dado un id, solo puede actualizar su <i>name</i> y <i>email</i>
DELETE	/:id	Deshabilitar la cuenta de un usuario

Utilizar el front-end que su instructor les entregue para conectar los endpoints necesarios y enviar el JWT para validar las sesiones.

Semana 4:

Temas:

- Deploy con Heroku

“Muy bien estimado desarrollador junior, creo que podemos lanzar a producción nuestra aplicación para ver como se comporta en un entorno de producción, por favor sube tu proyecto a la plataforma de Heroku, y asegúrate que el API funcione bien en línea”

Crea una cuenta en Heroku

Habilita una base de datos en Heroku

Configura las variables de entorno en Heroku

Conecta tu cuenta de GitHub con Heroku para lanzar a producción

Sigue las buenas prácticas vistas en clase (Morgan, compression, helmet, rate-limit)

Semana 5:

Temas:

- Multer y FormData
- Correos

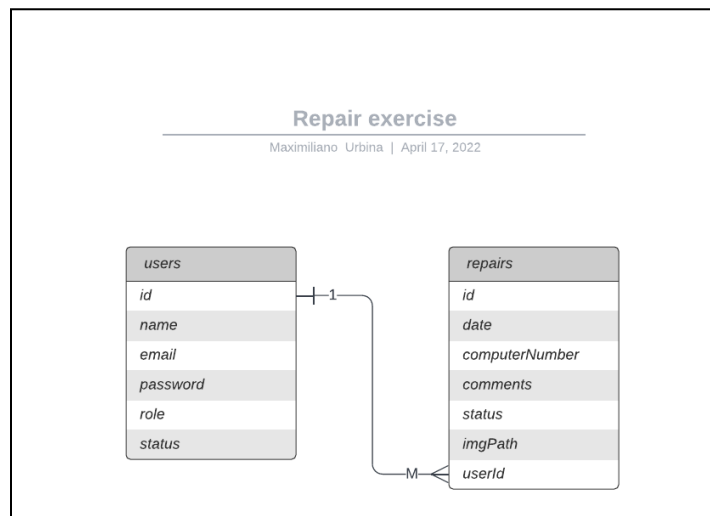
“Parece ser que nuestra app en Heroku esta funcionando correctamente. Muy buen trabajo. Nuestro equipo de QA (Quality Assurance) nos recomienda que el usuario le sería útil subir una imagen para poder tener mas contexto de la situación que quieren que atendamos. Con una imagen debería bastar de momento. Y sería útil guardar esta imagen en la nube, creo que podemos empezar con Firebase de momento.

Actualiza el modelo de repairs para que se asemeje al de la imagen que te adjunto, recuerda aplicar los cambios.

Una vez lista la lógica en el servidor, actualizar el front-end para que pueda enviar una imagen.

También estaría bien que el usuario reciba un correo cuando cree una cuenta con nosotros, y otro correo cuando actualicemos sobre su equipo, ya sea si fue reparado o cancelado.”

Realiza los siguientes cambios a los modelos (agregar una columna a repairs, imgPath)



Instala e implementa Multer para recibir una sola imagen

Instala y conecta tu cuenta de Firebase a Express

Sube la imagen del usuario a la nube

El campo de la imagen puede ser nulo, es decir, el usuario puede subir una imagen, pero no está obligado.

Instala **nodemailer**, **pug**, y **mailtrap**

Implementa la lógica para enviar un correo como se vio en clase, para enviar correos cuando el usuario:

- Cree una nueva cuenta
- Cuando su servicio sea actualizado, ya sea cancelado o completado