# EcoHarmony Park - App

## HOJA DE ESTILOS

### Estándares y convenciones a seguir para trabajar con las implementaciones a desarrollar.


Integrantes:

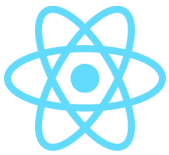
- 86657 - Juan Salvador Barbera.
- 90297 - Francisco Cornejo.
- 83009 - Mateo Romero Plaza.
- 98717 - Mateo Maldonado.
- 85291 - Nicolás Ranalli.
- 95034 - Matias Sciarra.
- 89058 - Luciano Gomez.
- 82397 - Diego Sosa.

# Front:

1. **Stack tecnológico:**

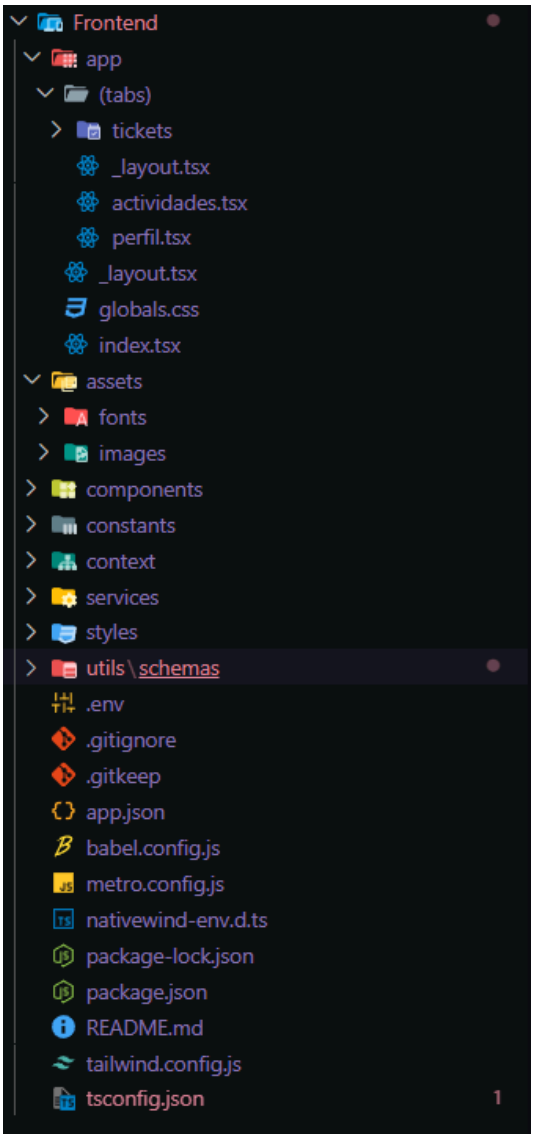
Expo

React Native



Native Wind

2. **Organización del código:**




- **app:** Tendremos las diversas “páginas” o pestañas de la app.

- **assets:** Tendremos todas las herramientas de soporte para el front - imágenes, fuentes de letras, etc.
- **components:** En ella tendremos todos los componentes de las diversas pantallas, partes del form.
- **constants:** Encontraremos constantes globales a todo el sistema, colores por ejemplo.
- **context:** Encontraremos el archivo que gestiona el estado de los tickets.
- **services:** Encontraremos todos los accesos a los endpoints del back. Conecta al front con el back.
- **styles:** Tendremos los estilos generales del proyecto.
- **utils:** Tendremos archivos de configuración del proyecto.

3. **Librerías usadas:**
   - Expo router: Esta librería nos ofrece organizar el proyecto de forma “file base”, donde se estructura el ruteo del código en forma de árbol siguiendo o basándose en una configuración de carpetas. Los componentes actuarán como padre e hijo o rama - hoja.

4. **Styling:**
   - Utilizaremos estilos en línea, se definen propio para el componente.
   - Solo contamos con una hoja de estilos, la cual se usa para definir las características comunes de toda la app.

5. **Reglas de nombrado:**
   - Componentes: PascalCase.
   - No Componentes: camelCase.
   - Archivos y carpetas: camelCase.
   - Variables: camelCase.
   - Props: camelCase.
   - Hoja de estilos: camelCase.

6. **Misceláneas:**
   - File encoding:
     - Los archivos están codificados en formato UTF-8.
   - Comentarios:
     - // para comentarios de una línea.

        - /**/ para comentarios multilínea.
    - Idioma:
        - No aplicamos restricciones de lenguaje, nos manejamos entre el español y el inglés.
    - Imports:
        - Seguimos las reglas de la ES6 (Ecmascript 6), respetando tanto rutas relativas como absolutas.

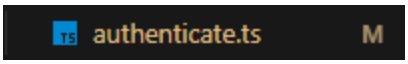
| Tipo de import | Ejemplo | Uso |
|---|---|---|
| **Module import** | import * as foo from '...'; | TypeScript imports |
| **Desestructurado** | import {SomeThing} from '...'; | TypeScript imports |
| **Default** | import SomeThing from '...'; | Solo para código externo que lo requiere |
| **Side-effects** | import '...'; | Solo para código que se ejecuta en segundo plano |

    - Exports: En este caso lo recomendable sería no usar *export default.* Pero debido a que expo router maneja dicha forma para aplicar la estructura de árbol.
    - Generalidades:
        - Un archivo por componente.
        - Se usará la sintaxis de TSX.
        - Se utilizará definición funcional de componentes con funciones asíncronas o arrow functions, en lugar de clases, debido a que tenemos que considerar muchos estados por los que pasarán los componentes.
        - El tipo de letra elegida por nuestra experta en el dominio es: Monserrat
    - Variables: Se usarán variables locales creadas a partir de const y let, en línea única, es decir no se usarán declaraciones del tipo:

```
let a = 1, b = 2;
```

# Back

1. **Nomenclatura:**
   - Archivos: camelCase (ej. authenticate.ts)

   - Variables y funciones: camelCase (ej. createTicket)

     ```
     export const createTicket = async (req: Request, res: Response) => {
     ```

     ```
     const newOperation: Tickets = {
     ```

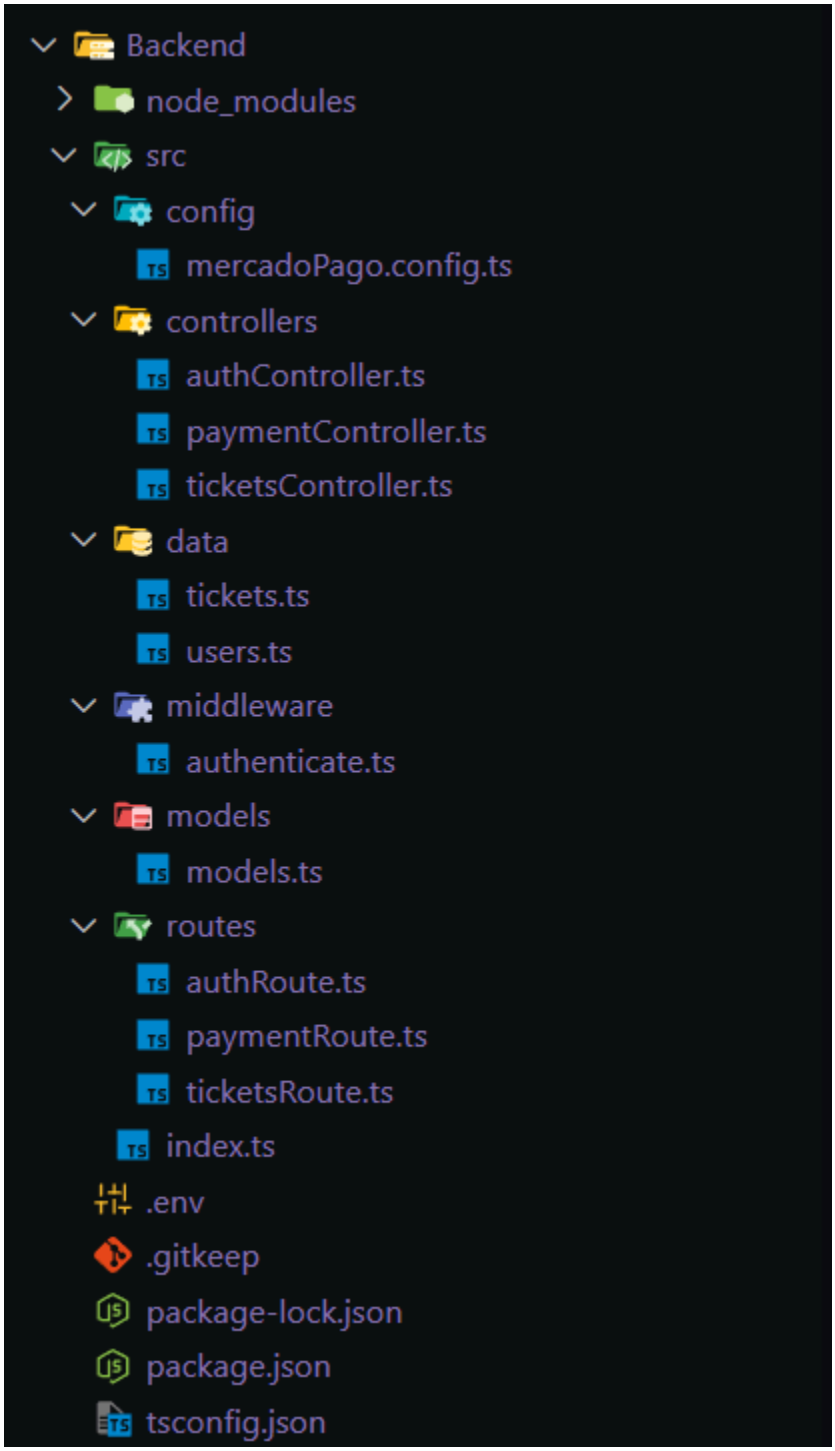
   - Clases e interfaces: PascalCase (ej. Ticket, CreatePreferenceResponse)

     ```
     export interface CreatePreferenceResponse {
     ```

   - Constantes: SNAKE_CASE (ej. JWT_SECRET)

     ```
     JWT_SECRET=secreto123
     ```

2. **Organización del Código:**




- **node_module:** Dependencias - librerías de node.
- **src/config**: Configuración del cliente de Mercado Pago para que la aplicación pueda interactuar con su API.

- **src/controllers:** Encargados de manejar la lógica del negocio.
- **src/data:** Almacena información estática de los modelos Tickets y Users, para aislar la conexión con una base de datos real.
- **src/middleware:** Aloja funciones que se ejecutan antes de que la solicitud llegue al controlador. En este caso, resolvemos la autenticación del usuario antes que él mismo pueda ejecutar cierta acción.
- **src/models:** Definen la estructura de datos de la aplicación.
- **src/routes:** Definen las rutas de los endpoints de la API.
- **.env:** Almacena variables de entorno sensibles como claves API o configuraciones del servidor.
- **.gitkeep:** Se usa para forzar a Git a incluir directorios vacíos en el repositorio.
- **package-lock.json:** Registra las versiones exactas de las dependencias instaladas para garantizar instalaciones consistentes.
- **package.json:** Define el proyecto Node.js, incluyendo nombre, scripts y dependencias necesarias.
- **tsconfig.json:** Configura las opciones del compilador TypeScript para el proyecto.

### 3. Estructura de Funciones:

Para la gran mayoría utilizamos Arrows Functions propias del lenguaje, dando una responsabilidad única y clara para cada función.

```
export const login = (req: Request, res: Response) => {

    const { email, password } = req.body;
    const user = users.find(user => user.email === email && user.password === password);
    if (user) {
        const token = jwt.sign({ id: user.id }, "secreto123", { expiresIn: '1h' });
        res.json({ token, user: { id: user.id,name:user.name, email: user.email } });
        return
    } else {
        res.status(401).json({ message: 'Email o contraseña incorrectos' });
        return
    }
}
```

También usamos funciones asíncronas, para que no salten errores no contemplados como Timeouts, o si el servidor tiene que esperar que se extraiga algún dato de la base de datos, que no se saltee esa petición y pueda proporcionar ciertos errores.

```
export const getTicketById = async (req: Request, res: Response) => {
  const ticketId = req.params.id;
  const ticket = ticketsList.find((ticket) => ticket.idOperation === ticketId);

  if (!ticket) {
    res.status(404).json({ error: "Ticket no encontrado" });
    return;
  }

  res.status(200).json({ ticket });
  return;
};
```

4. **Control de Errores:**

En el backend tomamos los errores y los devolvemos en formato de JSON en caso que los hubiere.

```
export const verifyTicket = async (req: Request, res: Response) => {
  const ticketId = req.params.id;

  let foundTicket: Tickets | undefined;

  const operationId = ticketsList.find(
    (t) => t.idOperation.toString() === ticketId
  );

  if (operationId) {
    foundTicket = operationId;
  }

  if (!foundTicket) {
    res.status(404).json({ valid: false, message: "Ticket no encontrado" });
    return;
  }

  if (foundTicket.usedOperation) {
    res.status(400).json({ valid: false, message: "El ticket ya fue usado" });
    return;
  }

  res.status(200).json({
    valid: true,
    message: "Ticket válido y marcado como usado",
    ticket: foundTicket,
  });
  return;
};
```

También utilizamos bloques de código try-catch, para mejorar la visibilidad del código.

```
export const authenticate = (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {···
  } catch (error) {
    res.status(500).json({ error: "Internal server error" });
    return;
  }
};
```

5. **Comentarios y Documentación:**

Usamos comentarios sólo cuando el código no sea autoexplicativo. También utilizamos algún comentario para explicar una parte del código que prosigue, a modo de "separar" en bloques el mismo.

Formato:

- // para comentar una línea
- /* */ para comentar un bloque de código.

6. **Librerías**
    - **qrcode**: Nos permitió la generación del código qr para la entrada.
    - **nodemailer**: Para gestionar el envío de mails.
    - **Mercadopago**: Le delegamos la responsabilidad del cobro a la api de mercadopago, permitiendo así una mejor gestión del cobro de los tickets cuando el método de pago seleccionado es por tarjeta.