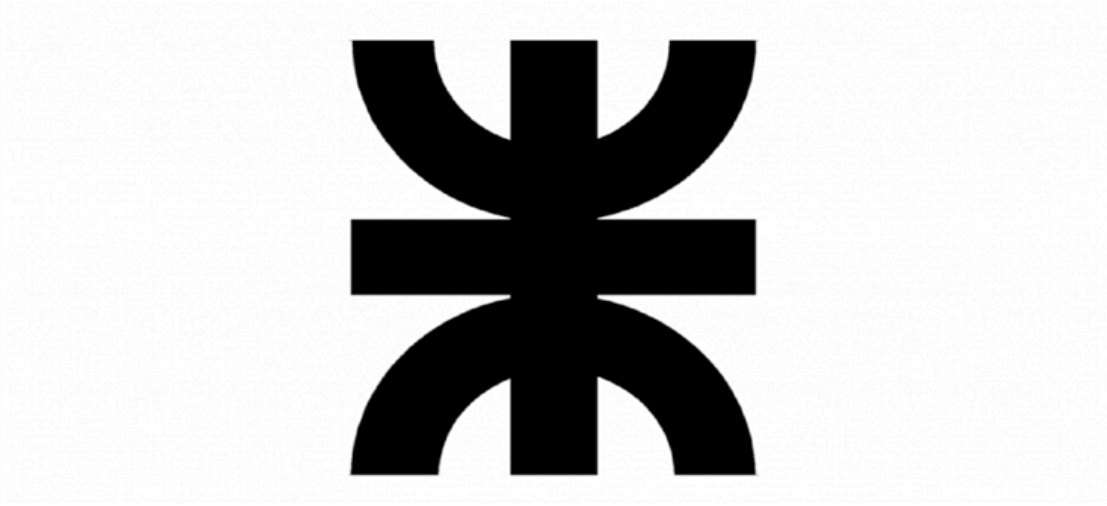
UNIVERSIDAD TECNOLÓGICA NACIONAL

FACULTAD REGIONAL CÓRDOBA



Cátedra Ingeniería y Calidad de Software

Comisión 4k1

Práctico N°6: Implementación de User Stories

Documento de Estilo de Código

Grupo 5

Integrantes:

Camargo Mano, Juan Ignacio - 85308
Fragherazzi, Leo Martin - 90106
Gallo, Juan Ignacio - 90994
Maero, Augusto - 91104
Rodriguez Saseta, Valentín - 90796
Torti, Jeremías - 87531
Villane, Ignacio - 62687

Docentes:

- ➔ Ing. Mickaela Crespo
- ➔ Ing. Judith Meles
- ➔ Constanza Garnero

User Story: Aceptar cotización

"Como dador de carga quiero aceptar una cotización para contratar el servicio de transporte."

- Criterios de Aceptación:

- Debe mostrar el nombre del transportista y su calificación.
- Debe mostrar la fecha de retiro y entrega del traslado.
- Debe visualizar el importe del viaje y las formas de pago establecidas.
- Debe elegir una de las formas de pago (tarjeta, contado al retirar o contado contra entrega) definidas por el transportista.
- Si la forma de pago es tarjeta, debe ingresar los datos de la tarjeta (número, pin, nombre completo, tipo y número de documento) y procesar el pago mediante la pasarela de pago elegida.
- Debe informar que el pago se procesó correctamente y el nro. de pago devuelto por la pasarela de pago.
- Debe informar que el pago se rechaza y debe permitir elegir otro medio de pago o ingresar otra tarjeta.
- Debe cambiar el estado del pedido de envío a "Confirmado".
- Debe enviar una notificación PUSH al transportista informando que se confirmó su cotización y la forma de pago elegida.
- Debe enviar un email al transportista informando que se confirmó su cotización y la forma de pago elegida.

- Pruebas de Usuario:

- Probar aceptar una cotización con pago al retirar o contra entrega (pasa).
- Probar aceptar una cotización sin elegir la forma de pago (falla).
- Probar pagar la cotización con tarjeta de crédito vigente y con saldo (pasa).
- Probar pagar la cotización con tarjeta de crédito sin saldo suficiente (falla).
- Probar pagar la cotización con tarjeta de crédito con datos no válidos (falla).
- Probar pagar la cotización con tarjeta de crédito sin ingresar los datos de la tarjeta (falla).
- Probar pagar la cotización con tarjeta de débito vigente y con saldo (pasa).
- Probar pagar la cotización con tarjeta de débito sin saldo suficiente (falla).
- Probar pagar la cotización con tarjeta de débito con datos no válidos (falla).
- Probar pagar la cotización con tarjeta de crédito sin ingresar los datos de la tarjeta (falla).
- Probar recepción de email de confirmación de pago por parte del transportista (pasa).
- Probar recepción de notificación PUSH de confirmación por parte del transportista (pasa).
- Probar que no se pueda aceptar otra cotización cuando ya está confirmada (falla).

Tecnologías:

Para la implementación de la User Story “Aceptar Cotización” decidimos utilizar las siguientes tecnologías:

- ***Lenguajes***: JavaScript, HTML, CSS.
- ***Control de Versión***: GitHub.
- ***Frameworks***: React.
- ***Entorno***: Visual Studio Code.
- ***Librerías***:
    - EmailJS Librería para enviar correos electrónicos desde el frontend
    - React-Router-DOM**:** Maneja el enrutamiento en aplicaciones de React.
    - React-Toastify: Librería para mostrar notificaciones emergentes (toasts) en aplicaciones de React.

Estructura del Proyecto:

- **Componentes**: Mantuvimos los componentes en una carpeta `src/components`. Cada componente tiene su propio archivo y su respectiva hoja de estilos si es necesario. Los nombres de los archivos de componentes están en **PascalCase** (ejemplo: `CardDetailsForm.js`).
- **Estilos**: Las hojas de estilos están en la carpeta `src/styles`. Usamos nombres descriptivos para los archivos CSS, dejando claro el componente al que están asociados (ejemplo: `CardDetailsForm.css`).
- **Datos:** Cualquier fuente de datos (por ejemplo, tarjetas o transportistas) debe estar en una carpeta `src/data`. Este contenido debe ser reutilizable y modular.

Convenciones de Nombres

- **Componentes:** Usamos **PascalCase** para componentes (`CardDetailsForm.js`, `ConfirmationModal.js`).
- **Funciones:** Utilizamos **camelCase** para nombres de funciones y manejadores de eventos (`handlePaymentSubmit`, `confirmPayment`).
- **Variables de Estado:** Utilizamos **camelCase** para variables de estado y nombres descriptivos que representan el valor almacenado (`orderStatus`, `cardDetails`, `paymentNumber`).
- **Clases CSS:** Usamos **kebab-case** para las clases en CSS (`payment-form`, `modal-overlay`).
- **Constantes:** Para aquellos valores constantes o inmutables, usamos el siguiente formato: **UPPERCASE_SNAKE_CASE**

Estructura del Código:

- **Componentes Funcionales:** Se elige definir **componentes funcionales** con hooks sobre los componentes de clase para mejorar la simplicidad y la capacidad de reutilización.

- **Desestructuración de Props y Estado:** según sea necesario a modo de facilitar la lectura y evitar repetir `props.` ó `state.`

  Ej: `const { number, pin, name } = cardDetails;`

- **Hooks**: Usamos hooks (`useState, useEffect`) para manejar el estado y efectos secundarios manteniendo la lógica dentro de los hooks organizada. Separamos las funciones si son demasiado largas o complicadas.

  ```
  useEffect(() => {

  }, []);
  ```

- **Condiciones y Validaciones:** Evitamos que haya anidamientos de condiciones usando **guard clauses** y **early returns** para salir rápidamente de las funciones cuando sea necesario.

  ```
  if (!cardType) return "Debe seleccionar el tipo de tarjeta.";

  if (!number || number.length < 16) return "Número de tarjeta
  inválido.";
  ```

Manejo de Formularios y Validaciones

- **Validación de Campos:** Centralizamos la validación de los formularios en una función separada para facilitar su mantenimiento y evitar duplicación de código.

  ```
  const validateFields = () => { const { number, pin, name,
  documentType, documentNumber } = cardDetails; // Validación de cada
  campo... };
  ```

- **Formateo de Datos**: Limpiar y formatear los datos del formulario antes de procesarlos (ejemplo: eliminando caracteres no numéricos en los números de tarjeta o pin).

  ```
  const newValue = (name === "number" || name === "pin") ?
  value.replace(/\D/, '') : value;
  ```

Manejo de Estados y Sesiones

- **SessionStorage:** Usamos `sessionStorage` para guardar datos temporales de la sesión, como el estado del pedido o el número de pago y se limpia de ser necesario mediante el uso de `clear()` o eliminando elementos específicos.
  ```
  sessionStorage.setItem("paymentNumber", generatedPaymentNumber);
  ```

- **Reseteo de Sesión**: Agregamos feedback visual o notificaciones al usuario cuando se resetea la sesión.
  ```
  const handleResetSession = () => { sessionStorage.clear();
  setOrderStatus("Pendiente"); setConfirmedPaymentMethod(null); };
  ```

Manejo de Errores y Notificaciones

- **Notificaciones:** usamos la biblioteca **react-toastify** para mostrar notificaciones de éxito o error de manera consistente y agradable para el usuario.
  ```
  showSuccessNotification("Pago procesado correctamente");
  ```

- **Manejo de Errores:** Se manejan los errores de validación y procesamiento de pago mostrando mensajes claros y concisos.

  ```
  if (validationError) {
    showErrorNotification(validationError);
    return;
  }
  ```

Estilos CSS

- **Modularización de Estilos:** Los estilos CSS están separados y organizados por componente. No se utilizan estilos globales a menos que sea necesario.
- **Nombres de Clases:** Usamos nombres de clases consistentes que reflejan su propósito o que se asocian a su respectivo componente al que están asociados.
  ```
  .payment-form {
    margin: 20px;
    padding: 10px;
  }
  ```

Reglas utilizadas basadas en “JavaScript Standard Style”

Algunos de los estándares implementados en nuestro código son:

- Usar 2 espacios como sangría.
  eslint: *indent*

```
function App() {
  return (
    <div className="App">
      <PaymentForm /> . . .
```

- Mantener la declaración “else” en la misma línea que sus llaves.
  eslint: *brace-style*

```
 if (isPaymentSuccessful) {
        . . .
} else
```

- Espacio luego de las comas en un array.
  eslint: *comma-spacing*

```
const transportistas = [
      { id: 1, name: "Jorge Rodriguez", rating: "4.5 de 5",
retiro: "10-09-2024", entrega: "11-09-2024", importe: 1500 }, . .
.
]
```

- *Espacio luego de condicionales.*
  eslint: *keyword-spacing*

```
 if (isPaymentSuccessful){ . . .
```

- Usar === en vez ==, a excepción de obj == null
  eslint: *eqeqeq*

```
 if (paymentMethod === "tarjeta" . . .
```

- Agregar espacios dentro de un mismo bloque { }.

eslint: *block-spacing*

```
const { name, value } . . .
```

- Usar camelcase para definir funciones, variables, archivos, etc.

  eslint: *camelcase*

  ```
  const newValue
  ```

- No dejar variables sin usar.
  eslint: no-unused-vars
- Espacio después de las palabras claves.
  eslint: keyword-spacing

  ```
  if (storedOrderStatus) {
     setOrderStatus(storedOrderStatus);
  }
  ```

- Agregar un espacio antes de los paréntesis de la función declarada.
  eslint: space-before-function-paren

  ```
  const generatePaymentNumber = () => {
      return Math.floor(100000 + Math.random() * 900000);
  };
  ```

- Las comas deben tener un espacio después de ellas.
  eslint: espaciado entre comas

  ```
     setOrderStatus("Confirmado");
     sessionStorage.setItem("orderStatus", "Confirmado");
  ```

Comentarios:

Se agregan comentarios breves que describan funcionalidades, variables y su uso y lógica implementada a lo largo del código desarrollado.

Además, usaremos comentarios para indicar modificaciones que se hagan sobre el código

Use spaces inside comments.

- Utilizar espacio dentro de los comentarios
  eslint: spaced-comment

  ```
  // comment
  /* comment */
  ```

Documentación:

Estandares “JavaScript Standar Styles”: https://standardjs.com/rules