SENATI

Fullstack Developer Software

MATERIAL TÉCNICO DE APOYO

RDA
RECURSO DIDÁCTICO PARA EL APRENDIZAJE

TAREA N° 01

Conoce y aplica prácticas de desarrollo de software colaborativo

1. FUNDAMENTOS DE GIT Y GITHUB

Git es un sistema de control de versiones distribuido que permite llevar un historial completo de los cambios realizados en un proyecto. Es utilizado para facilitar el trabajo colaborativo y evitar la pérdida de información.

Los comandos claves son:

```
git init                # Inicia un repositorio local
git clone <URL>         # Clona un repositorio remoto
git status              # Muestra el estado de los archivos
git add .               # Agrega todos los cambios al staging
git commit -m "mensaje" # Guarda los cambios con un mensaje
```

Cuyo flujo de trabajo básico es dado por:

```
# Clonar un repositorio existente
git clone https://github.com/usuario/proyecto.git

# Realizar cambios y subirlos
git add .
git commit -m "Corrección de errores"
git push
```

Git y GitHub

- **Historia y evolución del control de versiones.**

  Antes, los programadores usaban sistemas centralizados como SVN, donde un error podía afectar todo el proyecto. Git fue creado por Linus Torvalds (creador de Linux) en 2005 como un sistema distribuido de control de versiones, rápido y seguro.

- **Diferencias entre Git y GitHub.**

  - **Git:** herramienta que se instala localmente para llevar el control de versiones de tu proyecto.

  - **GitHub:** plataforma online que permite almacenar repositorios Git, colaborar con otros y gestionar proyectos.

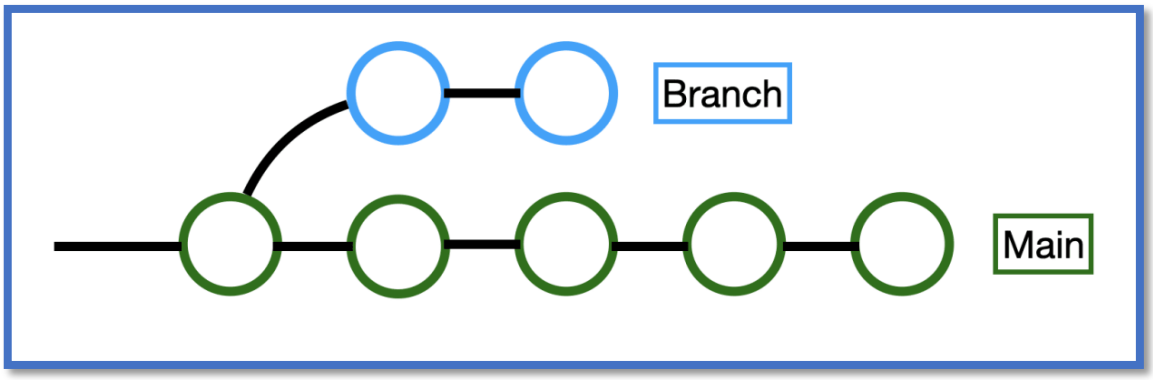
- **Flujo de trabajo en Git (local, remoto y colaborativo).**

  - **Local:** Trabajas en tu computadora (working directory → staging area → commit).

  - **Remoto**: Subes tu trabajo a GitHub (push) y descargas cambios de otros (pull).

  - **Colaborativo:** Se crean ramas para trabajar en nuevas funciones sin afectar el proyecto principal.

2. MANEJO DE REPOSITORIOS Y RAMAS



Ramas

- **Creación y configuración de un repositorio en GitHub.**

  Puedes crear uno desde la web de GitHub, luego lo clonas con Git (git clone) para trabajar en tu PC. También puedes conectar un proyecto local con git remote add origin.

- **Tipos de repositorios: Público, privado y fork.**

- **Público:** Visible para todos.
- **Privado:** Solo tú y quienes invites pueden verlo.
- **Fork:** Copia de un repositorio ajeno que puedes modificar sin afectar el original.

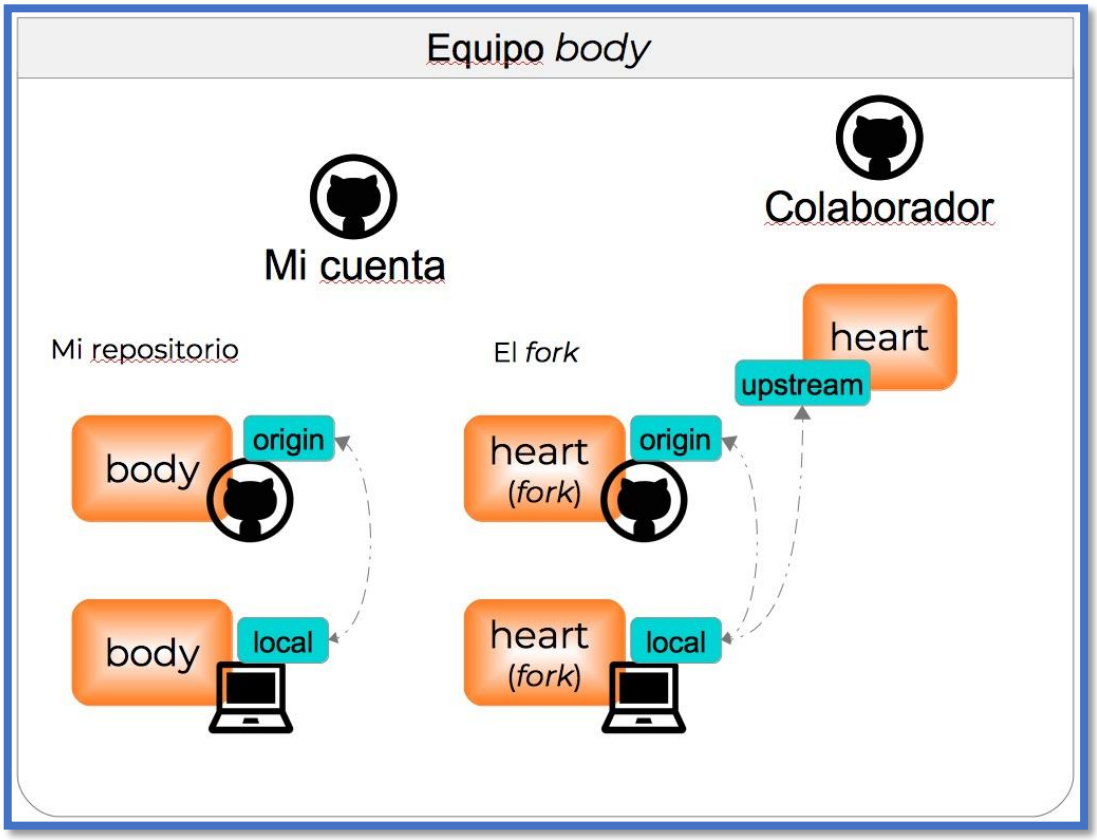
- **Gestión de ramas y estrategias de branching (Git Flow).**
  - Las ramas (branches) permiten trabajar en paralelo sin dañar el código principal (por ejemplo: main, dev, feature-login).
  - **Git Flow:** Es una estrategia que organiza las ramas para mejorar el trabajo en equipo, usando ramas como develop, feature, release, y hotfix.
    - **Git Flow (flujo estructurado)**
      - ✓ **main:** versión estable.
      - ✓ **develop:** integración de nuevas funcionalidades.
      - ✓ **feature/*:** desarrollo de nuevas funciones.
      - ✓ **release/*:** preparación de versiones.
      - ✓ **hotfix/*:** corrección de errores críticos.
    - **Ejemplo práctico:**

```
git checkout -b feature/formulario-login
# Hacer cambios...
git commit -am "Agrega formulario de login"
git checkout develop
git merge feature/formulario-login
```

3. CONTROL DE VERSIONES Y COLABORACIÓN

- **Uso de comandos Git básicos y avanzados (Commit, Push, Pull, Merge, Rebase).**
  - **git init:** Inicia un repositorio.
  - **git add .:** Agrega todos los archivos al área de staging.
  - **git commit -m "mensaje":** Guarda los cambios en el historial.

- **git push:** Envía los cambios al repositorio remoto.
- **git pull:** Descarga y fusiona cambios del remoto.
- **git merge:** Une ramas (por ejemplo, una feature con main).
- **git rebase:** Reorganiza commits para mantener un historial más limpio (avanzado).



Colaboración

- **Manejo de conflictos en Git.**

  Ocurren cuando dos personas modifican la misma parte del código. Git avisa y el desarrollador debe elegir qué línea conservar o combinar los cambios manualmente.

  - **Ejemplo:**

    ```
    <<<<<<< HEAD
    console.log("versión A");
    =======
    console.log("versión B");
    >>>>>>> rama-remota
    ```

    El desarrollador debe elegir cuál línea conservar o fusionarlas.

- **Estrategias de colaboración: Pull Requests, Issues y Code Reviews.**
  - **Pull Requests:** Solicitudes para fusionar código nuevo, permiten revisión previa.
  - **Issues:** Reportes de errores, ideas o tareas pendientes.
  - **Code Reviews:** Revisión del código por otros miembros del equipo antes de aprobar un cambio.

4. SEGURIDAD Y BUENAS PRÁCTICAS EN GITHUB

- **Uso de llaves SSH y autenticación en GitHub.**

  Permite conectar tu máquina a GitHub de forma segura sin ingresar usuario y contraseña cada vez. Se genera una clave SSH y se agrega a tu cuenta de GitHub.

  Conectarse a GitHub mediante llaves SSH evita tener que ingresar usuario y contraseña.

  ```
  ssh-keygen -t ed25519 -C "tu_correo@example.com"
  # Luego se agrega la llave pública a GitHub
  ```

- **Integración de GitHub Actions para automatización de flujos de trabajo.**

  Herramienta para automatizar tareas como pruebas, despliegues o validaciones cada vez que se sube código. Ejemplo: ejecutar tests automáticamente al hacer un pull request.

- **Implementación de GitHub Projects para la organización de tareas.**

  Funciona como un tablero Kanban para organizar tareas con tarjetas. Ideal para gestionar el progreso de un proyecto, asignar tareas y visualizar el flujo de trabajo del equipo.

TAREA N°02

Usa entorno de ejecución backend con JavaScript

1. CONFIGURACIÓN DEL ENTORNO BACKEND

- **Instalación y configuración de Node.js.**

  Node.js es un entorno de ejecución para JavaScript del lado del servidor.

  - **¿Por qué usar Node.js en desarrollo backend?**

    Node.js permite crear servidores web escalables y aplicaciones de alto rendimiento usando JavaScript en el servidor. Es asíncrono, basado en eventos y single-threaded, lo cual lo hace ideal para APIs REST y microservicios.

Node js

Se instala desde https://nodejs.org/es

Al instalarlo, también se incluye npm (Node Package Manager).

  - **Verificación de la instalación:**

```
node -v    # Ver versión de Node.js
npm -v     # Ver versión de npm
```

- **Configuración de proyecto inicial:**

  ```
  mkdir mi-api
  cd mi-api
  npm init -y
  ```

  Esto genera un archivo package.json, necesario para gestionar dependencias.

- **Uso de npm para la gestión de dependencias y scripts.**

  - **npm init:** Crea el archivo package.json con la información del proyecto.

  - **npm install <paquete>:** Instala librerías (como express).

    - **Ejemplo de package.json con scripts útiles:**

      ```
      "scripts": {
        "start": "node index.js",
        "dev": "nodemon index.js",
        "test": "jest"
      }
      ```

    - **Instalación de herramientas útiles:**

      ```
      npm install nodemon --save-dev   # Reinicia automáticamente tu app
      npm install dotenv               # Manejo de variables de entorno
      ```

  - **npm run <script>:** Ejecuta scripts personalizados definidos en package.json.

2. DESARROLLO DE APIS REST CON EXPRESS.JS

- **Creación de un servidor con Express.js.**

  Express es un framework minimalista para Node.js que facilita la creación de servidores y APIs.

  Ejemplo básico:

  ```
  const express = require('express');
  const app = express();

  app.get('/', (req, res) => {
    res.send('Hola, mundo!');
  });
  ```

```
app.listen(3000, () => {
  console.log('Servidor corriendo en puerto 3000');
});
```

- **Estructura recomendada de carpetas:**

```
/mi-api
  /routes
    usuarios.js
  /controllers
    usuarioController.js
  index.js
```

- **Archivo routes/usuarios.js:**

```
const express = require('express');
const router = express.Router();
const { listarUsuarios } = require('../controllers/usuarioController');

router.get('/', listarUsuarios);

module.exports = router;
```

- **Archivo index.js:**

```
const express = require('express');
const app = express();
const usuariosRouter = require('./routes/usuarios');

app.use('/usuarios', usuariosRouter);
app.listen(3000, () => console.log('Servidor activo'));
```

- **Manejo de rutas y middleware en Express.**

- **Rutas:** Definen los endpoints (GET, POST, PUT, DELETE) que la API va a manejar.

```
app.post('/usuarios', (req, res) => { /* lógica */ });
```

- **Middleware:** Funciones que se ejecutan antes de llegar a las rutas. Se usan para validaciones, logs, manejo de errores, etc.

```
app.use(express.json()); // Middleware para leer JSON
```

  - **Middleware personalizado:**

```
function logger(req, res, next) {
```

```
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
}

app.use(logger);
```

➢ **Orden de ejecución de middlewares:**

El orden importa. Si colocas un middleware después de las rutas, no se ejecutará para esas rutas.

3. CONEXIÓN CON BASE DE DATOS RELACIONALES

MySQL

- **Instalación y configuración de MySQL en el backend.**

  MySQL es un sistema de base de datos relacional. Puedes instalarlo desde https://www.mysql.com/ y luego conectarlo desde Node.js con librerías como mysql2.

  - **Ejemplo de conexión con mysql2:**

```
const mysql = require('mysql2');
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'tu_password',
  database: 'mi_base'
});
```

```
connection.connect(err => {
  if (err) throw err;
  console.log('Conexión exitosa a MySQL');
});
```

- **Uso de Sequelize como ORM para MySQL.**

  Sequelize es un ORM (Object Relational Mapper) que permite trabajar con bases de datos usando objetos en lugar de SQL.

  Ejemplo:

  ```
  npm install sequelize mysql2
  ```

  - **Ejemplo de modelo básico:**

    ```
    module.exports = (sequelize, DataTypes) => {
      const Usuario = sequelize.define('Usuario', {
        nombre: DataTypes.STRING,
        email: DataTypes.STRING
      });

      return Usuario;
    };
    ```

  - **Sincronizar modelos con la base de datos:**

    ```
    sequelize.sync().then(() => {
      console.log('Tablas sincronizadas');
    });
    ```

- **Implementación de modelos y migraciones de bases de datos.**

  - **Modelos:** Representan las tablas (por ejemplo, Usuario, Producto) como clases.

  - **Migraciones:** Scripts para crear/modificar la estructura de las tablas en la base de datos.

    Sequelize CLI facilita su uso:

    ```
    npx sequelize-cli init
    npx sequelize-cli model:generate --name Usuario --attributes
    nombre:string,email:string
    npx sequelize-cli db:migrate
    ```

➢ **Archivo de migración generado:**

```
'use strict';
module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('Usuarios', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      nombre: Sequelize.STRING,
      email: Sequelize.STRING,
      createdAt: Sequelize.DATE,
      updatedAt: Sequelize.DATE
    });
  },
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Usuarios');
  }
};
```

4. SEGURIDAD EN APLICACIONES BACKEND

Token JWT

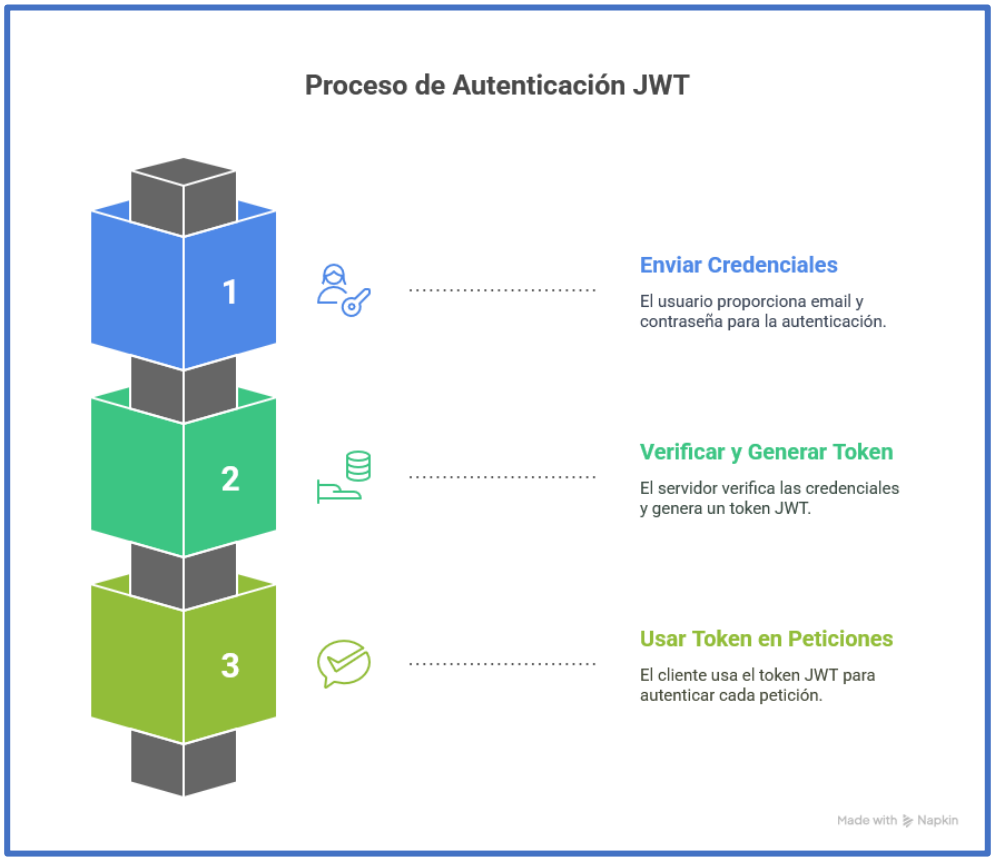
- **Implementación de autenticación con JSON Web Tokens (JWT).**

  JWT permite autenticar usuarios mediante un token seguro que se genera al iniciar sesión y se envía en cada petición.

  Paquetes comunes: jsonwebtoken, bcrypt para encriptar contraseñas.

  ```
  const token = jwt.sign({ id: usuario.id }, 'secreto', { expiresIn: '1h' });
  ```

- **Proceso básico:**
  - El usuario envía email y contraseña.
  - El servidor verifica y genera un token JWT.
  - El cliente usa ese token en cada petición.



Proceso de autenticación JWT

- **Verificar token en rutas protegidas:**

```
const jwt = require('jsonwebtoken');

function verificarToken(req, res, next) {
  const token = req.headers['authorization'];
  if (!token) return res.status(403).send('Token requerido');

  jwt.verify(token, 'secreto', (err, decoded) => {
    if (err) return res.status(401).send('Token inválido');
    req.usuarioId = decoded.id;
    next();
  });
}
```

- **Protección contra ataques comunes (CORS, SQL Injection, XSS).**

- **CORS (Cross-Origin Resource Sharing):** Permite controlar qué dominios pueden consumir tu API.

```
const cors = require('cors');
app.use(cors());
```

➢ **Instalación y uso de Helmet:**

```
npm install helmet

const helmet = require('helmet');
app.use(helmet()); // Protege cabeceras HTTP
```

➢ **Validación de entrada para prevenir XSS:**

```
const { body, validationResult } = require('express-validator');

app.post('/usuarios',
  body('email').isEmail().normalizeEmail(),
  (req, res) => {
    const errores = validationResult(req);
    if (!errores.isEmpty()) return res.status(400).json({ errores: errores.array() });
    // Procesar usuario...
  }
);
```

- **SQL Injection:** Se evita al usar ORMs como Sequelize que hacen consultas seguras.

- **XSS (Cross-Site Scripting):** Se mitiga validando y limpiando la entrada de datos del usuario.

Protección contra ataques comunes

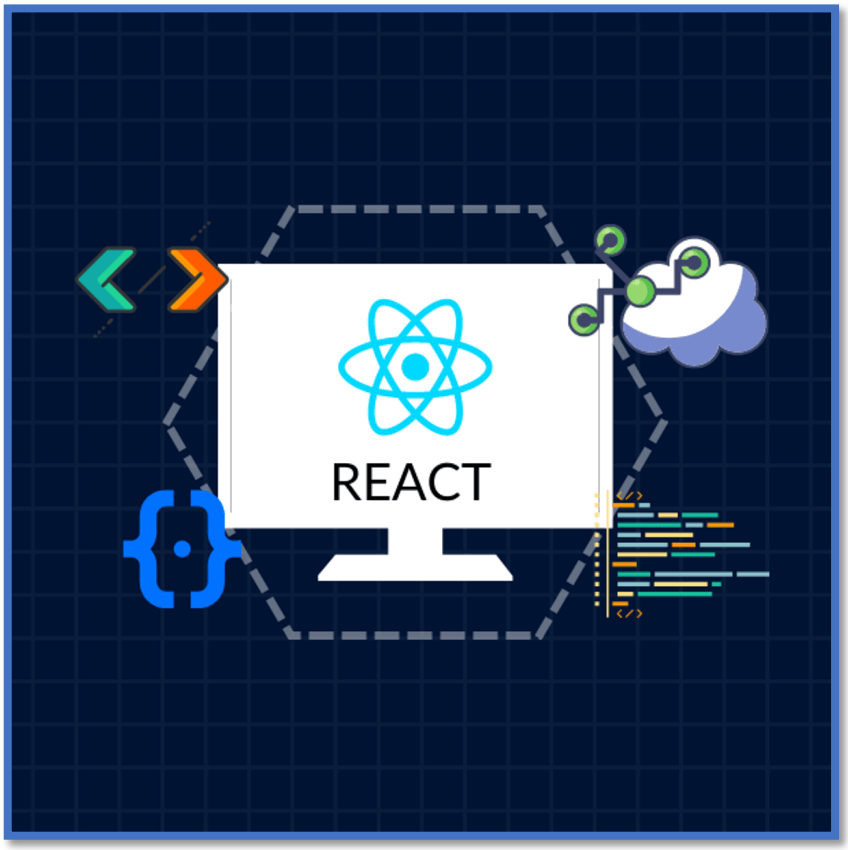
TAREA N° 03

Usa tecnología frontend con JavaScript

1. FUNDAMENTOS DE DESARROLLO FRONTEND CON REACT

- **Concepto y ventajas de SPA.**

  Una SPA es una aplicación web que carga una sola página HTML y actualiza el contenido dinámicamente sin recargar la página completa.



React

  - **Ventajas:** mayor velocidad, mejor experiencia de usuario, y menor carga del servidor.

  - **¿Por qué React es ideal para SPA?**

    React utiliza un DOM virtual que actualiza solo los componentes necesarios, mejorando el rendimiento. Además, permite construir interfaces dinámicas reactivas y reutilizables.

  - **Ejemplo comparativo entre SPA y MPA:**

    - **SPA:** Carga única, navegación con <Link />, no recarga la página.

- **MPA:** Cada navegación genera una nueva petición al servidor y recarga completa.

- **Configuración de un entorno React con Vite.**

  Vite es una herramienta moderna que permite crear proyectos React de forma rápida y ligera.

  - **Comando básico:**

    ```
    npm create vite@latest nombre-proyecto -- --template react
    cd nombre-proyecto
    npm install
    npm run dev
    ```

  - **Ventajas de Vite frente a Create React App (CRA):**

    - Arranque más rápido (server con ES modules).
    - Compilación más liviana.
    - Integración nativa con TypeScript y JSX.

  - **Dependencias útiles en proyectos reales:**

    ```
    npm install axios react-router-dom dotenv
    ```

    Estas se usan para llamadas HTTP, enrutamiento y variables de entorno.

- **Creación y estructuración de proyectos en React.**

  Un proyecto React bien estructurado incluye carpetas como:

  ```
  /src
    /components → Componentes reutilizables
    /pages      → Vistas o páginas principales
    /assets     → Imágenes, estilos
    App.jsx     → Componente principal
    main.jsx          → Punto de entrada
  ```

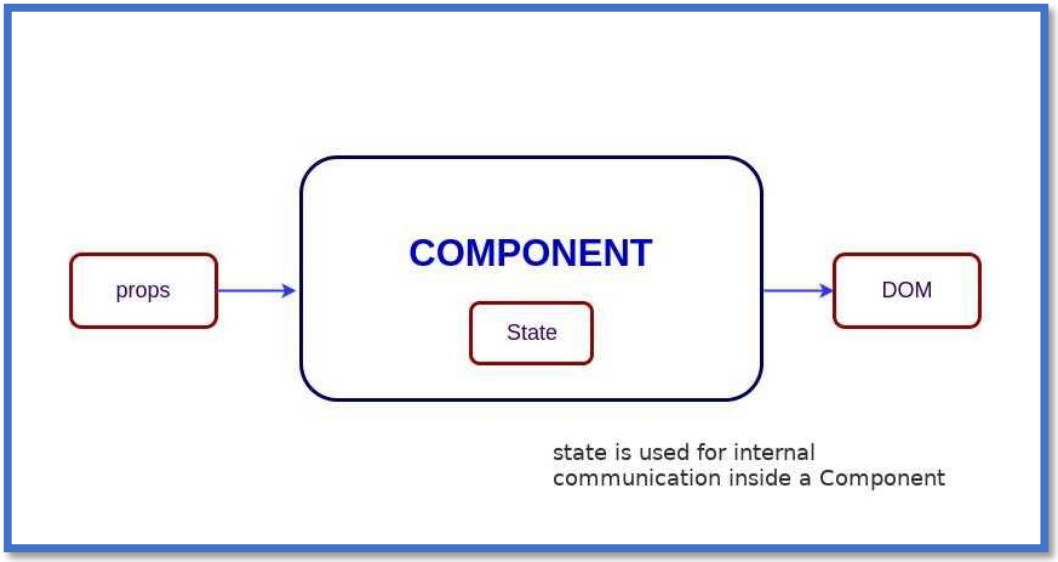
  - **Buenas prácticas:**

    - Separar lógica y presentación.
    - Usar nombres consistentes y descriptivos.
    - Centralizar constantes y rutas.

- **Ejemplo de rutas centralizadas:**

```
// routes.js
export const RUTAS = {
  INICIO: '/',
  CONTACTO: '/contacto',
};
```

2. COMPONENTES Y ESTADO EN REACT



Componente y estado

- **Uso de JSX para estructurar interfaces.**

  JSX permite escribir HTML dentro de JavaScript. Ejemplo:

```
function Saludo() {
  return <h1>¡Hola, React!</h1>;
}
```

  - A tener en cuenta con JSX:
    - Debe haber un solo nodo padre.
    - Usa className en vez de class.
    - Usa camelCase para eventos y props.
  - **Ejemplo:**

```
<div className="card">
  <h2>{titulo}</h2>
  <button onClick={handleClick}>Enviar</button>
</div>
```

- **Estados y ciclos de vida de los componentes.**
  - **Estado (useState):** Guarda datos que cambian con la interacción del usuario.
  - **Ciclo de vida:** Son etapas por las que pasa un componente (montaje, actualización, desmontaje).

    En componentes funcionales, se manejan con Hooks.

- **Manejo de Hooks en React (useState, useEffect).**
  - **useState:** Crea y actualiza el estado local de un componente.

    ```
    const [contador, setContador] = useState(0);
    ```

  - **useEffect:** Ejecuta funciones cuando el componente se monta o cambia un valor.

    ```
    useEffect(() => {
      console.log("Contador actualizado");
    }, [contador]);
    ```

    - **¿Cuándo se ejecuta useEffect?**

      ```
      useEffect(() => {
        console.log('Se montó el componente');
      }, []);
      ```

      Se ejecuta una sola vez al montar. Si se incluye una dependencia, se ejecuta cada vez que esta cambie.

    - **Manejo combinado de estado y efecto:**

      ```
      const [contador, setContador] = useState(0);

      useEffect(() => {
        document.title = `Clicks: ${contador}`;
      }, [contador]);
      ```

3. ENRUTAMIENTO Y NAVEGACIÓN EN REACT

- **Instalación y configuración de React Router.**

  React Router permite manejar múltiples vistas/páginas en una SPA. Se instala con:

```
npm install react-router-dom
```

Luego, se configura en el archivo principal:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  ); }
```

- **Rutas anidadas:**

```
<Routes>
  <Route path="/" element={<Layout />}>
    <Route index element={<Home />} />
    <Route path="about" element={<About />} />
  </Route>
</Routes>
```

- **Creación de rutas dinámicas y navegación entre páginas.**

Rutas dinámicas permiten mostrar contenido diferente según un parámetro:

```
<Route path="/usuario/:id" element={<Usuario />} />
```

Navegación:

```
import { Link, useNavigate } from 'react-router-dom';

<Link to="/about">Ir a Acerca</Link>

const navigate = useNavigate();
navigate('/home');
```

- **Acceder al parámetro dinámico:**

```
import { useParams } from 'react-router-dom';

function Usuario() {
```

```
  const { id } = useParams();
  return <h2>ID de usuario: {id}</h2>;
}
```

- **Redireccionar programáticamente:**

```
const navigate = useNavigate();
navigate('/dashboard');
```

4. MANEJO DE FORMULARIOS Y EVENTOS EN REACT

Formulario

- **Captura de datos con formularios controlados y no controlados.**
  - **Controlado:** Usa useState para manejar los valores.

```
const [nombre, setNombre] = useState('');
<input value={nombre} onChange={e => setNombre(e.target.value)} />
```

➢ **Ventajas de formularios controlados:**

✓ Más control sobre validación y envío.

✓ Mayor integración con librerías como Formik.

➢ **Formulario no controlado (con useRef):**

```
const inputRef = useRef();

const handleSubmit = () => {
  console.log(inputRef.current.value);
};

<input ref={inputRef} />
```

  - **No controlado:** Usa ref para acceder al valor directamente (menos

usado en React).

- **Validación de formularios con librerías como Formik y Yup.**
  - **Formik:** Facilita el manejo de formularios complejos.
  - **Yup:** Define reglas de validación (esquemas).

    npm install formik yup

    Ejemplo básico:

```
<Formik
  initialValues={{ email: '' }}
  validationSchema={Yup.object({ email: Yup.string().email().required() })}
  onSubmit={(values) => { console.log(values); }}
>
  <Form>
    <Field name="email" type="email" />
    <ErrorMessage name="email" />
    <button type="submit">Enviar</button>
  </Form>
</Formik>
```

- **Uso de eventos en React para interacción del usuario.**

  React usa eventos similares a HTML, pero en camelCase.

  Ejemplo:

```
<button onClick={() => alert('Clic!')}>Haz clic</button>
```

  - **Eventos comunes en React:**
    - **onSubmit:** para formularios.
    - **onChange:** para inputs.
    - **onMouseEnter / onMouseLeave:** para efectos visuales.
  - **Ejemplo con onChange:**

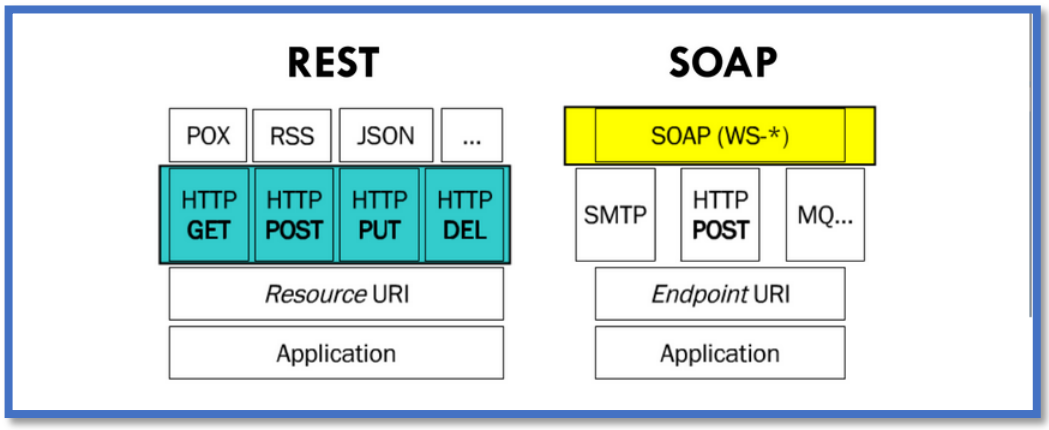
    <input onChange={e => console.log(e.target.value)} />

TAREA N° 04

Diseña y crea servicios API RESTful.

1. FUNDAMENTOS DE APIS Y SERVICIOS WEB

- **Definición y diferencias entre REST y SOAP.**



REST Y SOAP

  - **API (Application Programming Interface):** Permite que dos aplicaciones se comuniquen entre sí.
  - **REST (Representational State Transfer):** Estilo de arquitectura web que usa HTTP y respuestas en formato JSON.
  - **SOAP (Simple Object Access Protocol):** Protocolo más antiguo que usa XML y reglas estrictas.
  - **Diferencia clave:** REST es más ligero, flexible y común en desarrollo web moderno, mientras que SOAP es más seguro, pero más complejo.

- **Principios de diseño de API RESTful.**
  - **Usar recursos:** URLs representan objetos (ej. /usuarios, /productos).
  - **Métodos HTTP bien definidos:**
    - **GET** para obtener datos.
    - **POST** para crear.
    - **PUT** para actualizar.
    - **DELETE** para eliminar.

- **Stateless:** Cada solicitud debe contener toda la información necesaria (no guarda estado entre llamadas).

- **Formato estándar:** JSON para las respuestas y solicitudes.

- **Otros principios clave:**

  - **Versionado:** Asegura compatibilidad hacia atrás. Ejemplo: /api/v1/usuarios.

  - **HATEOAS:** Permite descubrir acciones posibles desde las respuestas.

  - **Códigos HTTP adecuados:** 200 (OK), 201 (CREATED), 400 (BAD REQUEST), 404 (NOT FOUND), 500 (ERROR INTERNO).

- **Ejemplo de respuesta REST bien formada:**

```
{
  "data": {
    "id": 1,
    "nombre": "Carlos",
    "email": "carlos@ejemplo.com"
  },
  "links": {
    "self": "/api/v1/usuarios/1"
  }
}
```

2. IMPLEMENTACIÓN DE API REST CON NODE.JS Y EXPRESS.JS

- **Creación y configuración de un servidor con Express.js.**

  Express es un framework para Node.js que permite crear servidores de forma rápida:

```
npm init -y
npm install express
```

  Código básico:

```
const express = require('express');
const app = express();

app.use(express.json()); // Middleware para leer JSON

app.listen(3000, () => {
```

```
  console.log('Servidor escuchando en puerto 3000');
});
```

- **Buenas prácticas iniciales:**
  - Usar variables de entorno con dotenv.
  - Crear archivos separados para rutas y controladores.
  - Modularizar el servidor para escalarlo.

```
npm install dotenv

require('dotenv').config();
const PORT = process.env.PORT || 3000;
```

- **Definición de endpoints REST con métodos HTTP (GET, POST, PUT, DELETE).**

```
// Obtener todos
app.get('/usuarios', (req, res) => { res.json([]); });

// Crear nuevo
app.post('/usuarios', (req, res) => { res.status(201).json({}); });

// Actualizar
app.put('/usuarios/:id', (req, res) => { res.json({}); });

// Eliminar
app.delete('/usuarios/:id', (req, res) => { res.status(204).end(); });
```

- **Estructura típica de CRUD completo:**

```
// GET: Obtener todos los usuarios
app.get('/usuarios', obtenerUsuarios);

// POST: Crear un nuevo usuario
app.post('/usuarios', crearUsuario);

// PUT: Actualizar usuario existente
app.put('/usuarios/:id', actualizarUsuario);

// DELETE: Eliminar usuario
app.delete('/usuarios/:id', eliminarUsuario);
```

- **Uso de middlewares en Express.js para validaciones y seguridad.**

  Los middlewares son funciones que se ejecutan antes de llegar al endpoint final. Ejemplos:

  - **Validar datos:**

    ```
    function validarUsuario(req, res, next) {
      if (!req.body.nombre) return res.status(400).send("Falta nombre");
      next();
    }

    app.post('/usuarios', validarUsuario, (req, res) => { /* ... */ });
    ```

  - **Seguridad básica:**

    - **cors** para controlar accesos

    - **helmet** para proteger cabeceras

      ```
      npm install cors helmet
      app.use(cors());
      app.use(helmet());
      ```

  - **Otros middlewares útiles:**

    - **morgan (logs HTTP):**

      ```
      npm install Morgan

      const morgan = require('morgan');
      app.use(morgan('dev'));
      ```

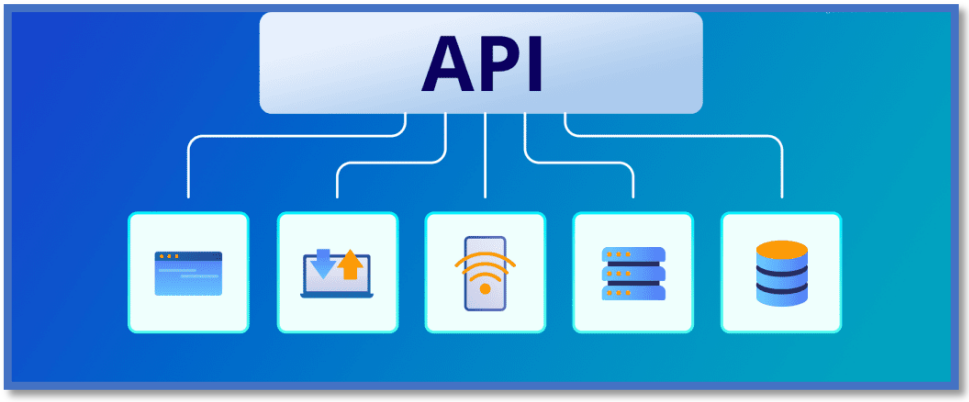
    - **express-validator (validación de entrada):**

      ```
      npm install express-validator

      const { body } = require('express-validator');

      app.post('/usuarios', [
        body('nombre').notEmpty(),
        body('email').isEmail()
      ], (req, res) => {
        // manejar validaciones
      });
      ```

3. MANEJO DE DATOS EN APIS



API

- **Serialización y deserialización con JSON.**
  - **Serializar:** Convertir objetos JavaScript en texto JSON para enviar por API.

    ```
    const usuario = { nombre: 'Ana' };
    res.json(usuario); // Express lo serializa automáticamente
    ```

    - ➢ **Serialización en profundidad:**
      - ✓ Puedes controlar qué campos incluir o excluir.
      - ✓ Puedes convertir objetos a JSON con .toJSON() en ORMs como Sequelize.
  - **Deserializar:** Convertir texto JSON recibido desde una petición en un objeto usable.

    ```
    app.use(express.json()); // Convierte el cuerpo JSON en objeto JS
    ```

    - ➢ **Deserialización segura:**
      - ✓ Asegura que los datos recibidos sean limpios y validables.
      - ✓ Útil para evitar errores de tipo o estructuras inesperadas.
- **Conexión y consultas a bases de datos relacionales y NoSQL.**
  - **Relacional (MySQL/PostgreSQL):**

    Uso de Sequelize o Knex para manejar modelos y consultas.

    ```
    const { Usuario } = require('./models');
    const usuarios = await Usuario.findAll();
    ```

- **NoSQL (MongoDB):**

```
npm install mongoose

const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/miapp');
```

- **Consultas complejas en Sequelize:**

```
const usuarios = await Usuario.findAll({
  where: { activo: true },
  include: [Rol]
});
```

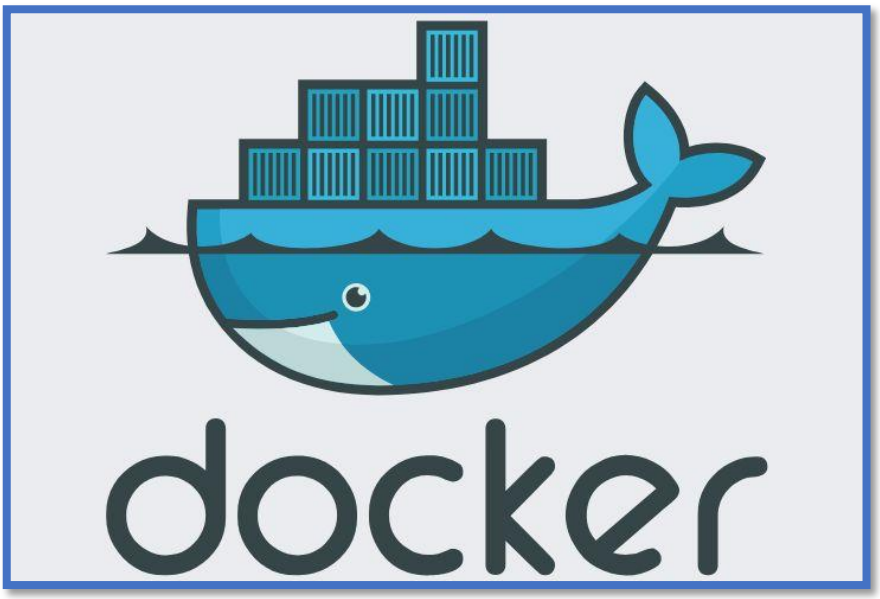
- **Ejemplo de esquema en MongoDB con Mongoose:**

```
const UsuarioSchema = new mongoose.Schema({
  nombre: String,
  email: { type: String, required: true, unique: true }
});

module.exports = mongoose.model('Usuario', UsuarioSchema);
```

4. CONTENEDORES Y DESPLIEGUE CON DOCKER

- **Conceptos de virtualización y contenedores.**



Docker

- **Virtualización:** Emula sistemas operativos completos (lento y pesado).
- **Contenedores (Docker):** Ejecutan solo la app con sus dependencias (ligeros y rápidos).

Permiten empaquetar una API para que funcione igual en cualquier máquina.

- **Configuración y despliegue de una API REST en Docker.**
  - **Crear un Dockerfile para definir cómo construir la imagen:**

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "index.js"]
```

  - **Crear un contenedor con Docker:**

```
docker build -t mi-api .
docker run -p 3000:3000 mi-api
```

    - **Variables de entorno en Docker:**

      Puedes definirlas en el Dockerfile o mediante docker run:

```
ENV PORT=3000
```

      O en docker-compose.yml:

```
environment:
  - NODE_ENV=production
  - DB_PASSWORD=secreto
```

  - **Uso de Docker Compose (opcional) para conectar app + base de datos:**

```
version: '3'
services:
  api:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - db
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
```

REFERENCIAS

- OpenJS Foundation. (n.d.). Node.js. https://nodejs.org
- Express.js. (n.d.). Express - Node.js web application framework. https://expressjs.com
- Sequelize. (n.d.). Sequelize: Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. https://sequelize.org
- MongoDB, Inc. (n.d.). Mongoose ODM v7. https://mongoosejs.com
- Docker Inc. (n.d.). Docker Documentation. https://docs.docker.com
- PostgreSQL Global Development Group. (n.d.). PostgreSQL documentation. https://www.postgresql.org/docs/
- Chacon, S., & Straub, B. (2014). Pro Git (2nd ed.). Apress. https://git-scm.com/book/en/v2
- GitHub. (n.d.). GitHub Docs. https://docs.github.com
- Atlassian. (n.d.). Git tutorials and best practices. https://www.atlassian.com/git

RDA

RECURSO DIDÁCTICO PARA EL APRENDIZAJE