UNIVERSIDAD ABIERTA PARA ADULTOS UAPA



**CARRERA DE INGENIERIA EN SOFTWARE ESCUELA DE ING EN SOFTWARE**

**TAREA 8**

**PRESENTADO POR: AMBIORIX DE LA ROSA**

**JOSÉ ELPIDIO GUICHARDO MIESES MOISÉS GÓMEZ**

**MATRÍCULA: 100033665**

# 100035493

**100041036**

**ASIGNATURA: SEMINARIO DE PROYECTOS II**

**FACILITADORA: LOIDA CHARLES RAMÍREZ**

**SANTIAGO DE LOS CABALLEROS REPÚBLICA DOMINICANA**

**LUNES 14 DE SEPTIEMBRE DEL 2024**

# Introducción

La documentación es una parte fundamental en el desarrollo de cualquier aplicación, y no es diferente para Gestick, un sistema de gestión de tickets diseñado para facilitar la comunicación y resolución de problemas técnicos dentro de las empresas. A través de una documentación clara y bien estructurada, los usuarios, administradores y desarrolladores pueden comprender cómo utilizar, mantener y mejorar el sistema. Además, permite una mejor gestión de las versiones y facilita la incorporación de nuevos miembros al equipo de desarrollo, asegurando la continuidad y escalabilidad del proyecto.

# Introducción Técnica

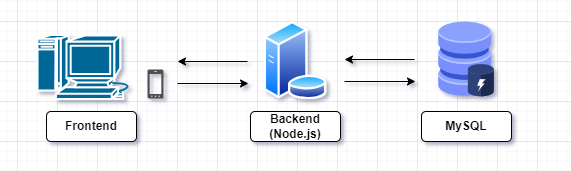
**GesTicks** es una aplicación web diseñada para optimizar la gestión de tickets en las empresas, facilitando la comunicación entre las áreas de trabajo y sus departamentos técnicos. La aplicación permite reportar y gestionar incidentes como problemas con sistemas operativos, aplicaciones, dispositivos, impresoras, y otros inconvenientes relacionados, todo a través de un sistema centralizado de tickets.

La aplicación está desarrollada utilizando una combinación de tecnologías modernas que aseguran tanto su funcionalidad como su eficiencia:

* + **Backend:** Node.js, una plataforma de alto rendimiento para la creación de aplicaciones web escalables.
  + **Frontend:** Tailwind CSS para la creación de interfaces de usuario responsivas y modernas, junto con Pug, un motor de plantillas que permite la renderización eficiente de las vistas.
  + **Base de Datos:** MySQL, que maneja el almacenamiento y gestión de los datos, asegurando la integridad y rapidez en la consulta de la información.
  + **Lenguaje de Programación Principal:** JavaScript, utilizado tanto en el frontend como en el backend, proporcionando una experiencia fluida en toda la aplicación.

El principal objetivo de GesTicks es facilitar la gestión de incidentes en las empresas, mejorando la eficiencia en la resolución de problemas técnicos y garantizando una mejor trazabilidad de los reportes enviados por los usuarios.

1. **Arquitectura del Sistema**



**Componentes Principales:**

**Usuarios (Frontend)**

* + Los usuarios interactúan con la aplicación web desde diferentes dispositivos (computadoras, móviles) a través de un navegador web.
  + El frontend está desarrollado con JavaScript, Pug (motor de plantillas), y Tailwind CSS para las interfaces visuales y responsivas.
  + Las peticiones de los usuarios se envían al servidor backend usando HTTP/HTTPS.

# Servidor Web (Backend con Node.js)

* + Node.js actúa como el servidor principal que recibe las solicitudes de los usuarios, procesa la lógica de negocio, y responde con los resultados.
  + Este servidor se encarga de manejar la autenticación de usuarios, la gestión de tickets, y otras operaciones relacionadas con la aplicación.

# Base de Datos (MySQL)

* + MySQL se encarga de almacenar todos los datos del sistema: usuarios, tickets, estados de los tickets, categorías de problemas, etc.
  + El backend se comunica con MySQL utilizando controladores o librerías que permiten hacer consultas, insertar o modificar datos.

# Dependencias (npm dependencies).

1. **bcrypt** (5.1.1)
   * **Descripción**: Librería para el **hashing** y la verificación de contraseñas. Se utiliza para encriptar las contraseñas de los usuarios antes de almacenarlas en la base de datos.
   * **Uso**: Autenticación de usuarios y seguridad de contraseñas.
2. **cookie-parser** (1.4.6)
   * **Descripción**: Middleware que permite analizar cookies en las peticiones HTTP. Facilita la gestión de las cookies, que pueden ser usadas para almacenar información como tokens de autenticación.
   * **Uso**: Gestión de cookies para mantener sesiones de usuario.
3. **csurf** (1.11.0)
   * **Descripción**: Middleware para proteger la aplicación contra ataques de **Cross-Site Request Forgery (CSRF)**. Asegura que las peticiones realizadas por los usuarios son legítimas.
   * **Uso**: Seguridad contra ataques CSRF en formularios y peticiones sensibles.
4. **dotenv** (16.4.5)
   * **Descripción**: Carga variables de entorno desde un archivo. env en process.env. Se utiliza para mantener las credenciales y configuraciones sensibles separadas del código fuente.
   * **Uso**: Manejo de configuraciones sensibles como claves API, credenciales de bases de datos, etc.
5. **express** (4.19.2)
   * **Descripción**: Framework de servidor web minimalista y flexible para Node.js. Es el núcleo del backend de la aplicación y gestiona todas las rutas, middleware y lógica del servidor.
   * **Uso**: Principal framework del servidor.
6. **express-validator** (7.2.0)
   * **Descripción**: Librería para la **validación** y **sanitización** de los datos de entrada en las peticiones HTTP. Se usa principalmente en los formularios de creación y edición de tickets.
   * **Uso**: Validación de datos en formularios y peticiones de usuarios.
7. **jsonwebtoken** (9.0.2)
   * **Descripción**: Implementación de **JSON Web Tokens (JWT)**. Se usa para la autenticación de usuarios, creando y verificando tokens que identifican a cada usuario en el sistema.
   * **Uso**: Autenticación de usuarios mediante tokens.
8. **mysql2** (3.11.0)
   * **Descripción**: Cliente de MySQL para Node.js, que permite la conexión y consulta de la base de datos MySQL. Es más eficiente que la librería estándar mysql.
   * **Uso**: Interacción con la base de datos MySQL.
9. **nodemailer** (6.9.14)
   * **Descripción**: Módulo para el envío de correos electrónicos a través de Node.js. Se utiliza para enviar notificaciones por correo cuando se crean o actualizan tickets.
   * **Uso**: Envío de correos electrónicos (notificaciones).
10. **pug** (3.0.3)
    * **Descripción**: Motor de plantillas de Node.js que permite generar HTML de manera dinámica y eficiente. Se utiliza en el frontend para la renderización de vistas.
    * **Uso**: Renderización de plantillas HTML dinámicas.
11. **sequelize** (6.37.3)
    * **Descripción**: ORM (Object-Relational Mapping) que facilita las interacciones con bases de datos SQL, permitiendo escribir consultas de manera sencilla y estructurada en JavaScript.
    * **Uso**: Interacción con la base de datos MySQL usando modelos.
12. **webpack** (5.94.0) & **webpack-cli** (5.1.4)
    * **Descripción**: Webpack es un **empaquetador de módulos** para aplicaciones JavaScript. Combina y optimiza los archivos JavaScript y recursos estáticos (CSS, imágenes, etc.).
    * **Uso**: Compilación y optimización del código frontend.

# Dependencias de desarrollo (npm devDependencies)

1. **concurrently** (8.2.2)
   * **Descripción**: Ejecuta múltiples comandos de scripts de npm al mismo tiempo. Se usa principalmente durante el desarrollo para correr el servidor de Node.js y Tailwind CSS simultáneamente.
   * **Uso**: Herramienta de desarrollo que facilita la ejecución de múltiples scripts a la vez.
2. **nodemon** (3.1.4)
   * **Descripción**: Herramienta que reinicia automáticamente el servidor cuando detecta cambios en los archivos del proyecto. Ahorra tiempo en el desarrollo al eliminar la necesidad de reiniciar manualmente.
   * **Uso**: Reinicio automático del servidor Node.js durante el desarrollo.
3. **tailwindcss** (3.4.10)
   * **Descripción**: Framework de CSS basado en utilidades que permite construir interfaces de usuario rápidas y responsivas mediante clases predefinidas. Se utiliza para el diseño del frontend de la aplicación.
   * **Uso**: Estilización del frontend.

# Comunicación

* Las comunicaciones entre el frontend y backend se realizan mediante peticiones HTTP/HTTPS, enviadas a través de métodos como GET, POST, PUT, DELETE.
* El backend envía respuestas en formato JSON al frontend para que los datos se puedan renderizar en las vistas.

# Flujo de Datos (Tickets)

* El flujo de datos describe cómo un ticket es creado, procesado y almacenado en la base de datos.
* El usuario llena un formulario en el frontend (JavaScript + Pug) para crear un nuevo ticket.
* La información es enviada al backend (Node.js) que procesa la solicitud.
* El backend registra el ticket en la base de datos MySQL.
* Una vez creado, el backend responde con un mensaje de confirmación o el ticket creado, que se muestra al usuario en el frontend.

# Estructura del Código

Este proyecto está desarrollado bajo el **Modelo Vista Controlador (MVC)** que es un patrón de diseño de software que separa una aplicación en tres componentes principales, cada uno con responsabilidades específicas, lo que facilita la organización y el mantenimiento del código.

1. **Modelo:** El modelo representa la lógica de datos y el comportamiento de la aplicación. Gestiona el acceso a los datos, las reglas de negocio, y las interacciones con la base de datos. El modelo también notifica a la vista cuando hay cambios en los datos, de manera que pueda actualizarse.
2. **Vista:** La vista es responsable de la interfaz de usuario. Es la presentación visual de los datos del modelo. Se encarga de mostrar los datos al usuario y recibir las interacciones (como clics o entradas de formularios). La vista no contiene lógica de negocio, solo maneja la presentación.
3. **Controlador:** El controlador actúa como un intermediario entre el modelo y la vista. Recibe las entradas del usuario desde la vista, las procesa, y realiza las actualizaciones necesarias en el modelo. Luego, solicita que la vista se actualice con la nueva información.

# Estructura de Carpetas y Archivos

**Config (Configuraciones y conexiones a la base de datos)**

* + **Db.js:** Gestiona la conexión a la base de datos utilizando el motor de base de datos (MySQL). Contiene las cadenas de conexión y credenciales.
  + **Incidentes.js:** Maneja configuraciones específicas relacionadas con la gestión de incidentes, como la configuración de umbrales, tiempos de espera o prioridades.

# Controllers (Lógica que conecta los modelos con las rutas)

* + **catalogoIncidentesControllers.js:** Define la lógica para manejar las operaciones del catálogo de incidentes, como la creación, actualización, recuperación y eliminación de incidentes.
  + **ticketsList.js:** Administra las operaciones relacionadas con los tickets, incluyendo la obtención de listas de tickets, filtrado por estado o prioridad, y paginación.
  + **usuarioController.js:** Gestiona la funcionalidad relacionada con los usuarios, como la creación de usuarios, inicio de sesión, actualización de perfiles, y autenticación.

# Helper (Funciones auxiliares usadas en toda la aplicación)

* + **Mailtrap.js:** Implementa la funcionalidad de envío de correos utilizando el servicio Mailtrap para enviar y validar correos electrónicos de los usuarios (para registro, recuperación de contraseñas, etc.).
  + **Token.js:** Proporciona funciones para generar, firmar y verificar tokens (por ejemplo, JWT) utilizados para la autenticación de usuarios, gestión de sesiones, y otras operaciones seguras.

# Middleware (Funciones que interceptan y gestionan las solicitudes, típicamente para la autenticación)

* + **authMiddleware.js:** Asegura que ciertas rutas solo puedan ser accedidas por usuarios autenticados. Verifica tokens válidos o sesiones activas, previniendo el acceso no autorizado a rutas protegidas.

# Model (Definiciones de esquemas de base de datos para ORM/ODM)

* + **Categoría.js:** Define el esquema y la estructura de datos para la tabla de categorías, incluyendo campos como nombre, descripción, y fecha de creación.
  + **Rols.js:** Gestiona el acceso basado en roles definiendo roles (por ejemplo, Admin, Usuario) y sus permisos en la base de datos.
  + **TicketHistory.js:** Almacena datos históricos de cada ticket, incluyendo cambios de estado, comentarios, marcas de tiempo y acciones de usuarios.
  + **Tickets.js:** Contiene el esquema para gestionar tickets, con campos como descripción del problema, departamento asignado, prioridad, estado, y marcas de tiempo.
  + **Usuario.js:** Define el esquema para la gestión de usuarios, incluyendo campos como nombre de usuario, correo electrónico, hash de contraseña, rol y marcas de tiempo.

# Index.js:

**Public (Archivos accesibles públicamente, normalmente para el frontend)**

* + **Css:** Contiene hojas de estilo (Tailwind CSS) para el diseño del frontend de la aplicación.
  + **js:** Archivos JavaScript necesarios para la interactividad del frontend.

# Queries (Consultas SQL y lógica de recuperación de datos)

* + **Consult.js:** Contiene la lógica de las consultas a la base de datos necesarias para iniciar la aplicación, como la obtención de datos de configuración o la inicialización de componentes clave.
  + **Queries.sql:** Archivo SQL con consultas para la configuración y gestión del esquema de la base de datos, creación de tablas, relaciones y procedimientos almacenados.

# Routes (Declaración de rutas API y de la aplicación web)

* + **catalogoIncidentesRoutes.js:** Declara los endpoints de la API relacionados con el catálogo de incidentes, entre otros., mapeando las rutas a sus respectivas funciones del controlador.
  + **usuarioRoutes.js:** Maneja todas las rutas relacionadas con los usuarios, como registro de usuarios, inicio de sesión, gestión de perfiles y recuperación de contraseñas.

# Seed (Datos iniciales para poblar la base de datos)

* + **Categoría.js:** Proporciona un conjunto de datos predefinidos para inicializar la tabla de categorías, asegurando que haya categorías básicas disponibles sin consultar constantemente la base de datos.
  + **Rols.js:** Prepobla la base de datos con roles predeterminados (por ejemplo, Admin, Usuario) para garantizar que el control de acceso esté configurado desde el inicio.
  + **Seeder.js:** Script central de seed que ejecuta todos los archivos de seed, poblando la base de datos con datos iniciales para pruebas o desarrollo.

# Src (Código fuente para la funcionalidad del frontend)

* + **Js:** Contiene scripts auxiliares utilizados en el frontend.
  + **setTimeout.js:** Implementa la lógica para gestionar temporizadores en la aplicación, como retrasar acciones o activar funciones después de un tiempo determinado.
  + **viewPass.js:** Proporciona funcionalidad para alternar la visibilidad de la contraseña en los formularios de entrada de usuario.

# Views (Vistas del frontend organizadas como layouts y plantillas)

* + **auth:** Contiene las vistas relacionadas con la autenticación de usuarios, como formularios de inicio de sesión, registro y recuperación de contraseñas.
  + **catalogo:** Muestra el catálogo de incidentes, permitiendo a los usuarios ver, buscar y filtrar incidentes.
  + **layouts:** Define el diseño general de la aplicación, incluyendo encabezados, pies de página y elementos de navegación reutilizados en múltiples páginas.
  + **template:** Plantillas para renderizar secciones o componentes específicos de la aplicación, permitiendo un código modular en el frontend.

# Archivos Principales:

* + **Index.html:** El archivo HTML principal que actúa como punto de entrada para el frontend de la aplicación, normalmente cargado primero en el navegador.
  + **Index.js:** El punto de entrada principal para el backend, donde la aplicación se inicializa, inicia el servidor, y carga las rutas y middlewares.

# Archivos de Configuración:

* + **.gitignore:** Especifica los archivos y directorios que Git debe ignorar para evitar que se suban archivos innecesarios (por ejemplo, node\_modules, archivos de entorno).
  + **README.md:** Archivo de documentación que explica el proyecto, las instrucciones de configuración y las pautas de uso.
  + **Package-lock.json:** Archivo autogenerado por npm que bloquea las versiones de las dependencias para garantizar que las compilaciones sean consistentes.
  + **Package.json:** Define los metadatos del proyecto, dependencias, scripts y otras configuraciones.
  + **Tailwind.config.js:** Archivo de configuración para personalizar Tailwind CSS, como la configuración de temas, colores personalizados o puntos de ruptura responsivos.
  + **Webpack.config.js:** Archivo de configuración para Webpack, encargado de empaquetar los archivos como JavaScript, CSS e imágenes.

# Configuración del Entorno Requisitos del Sistema

Para garantizar el funcionamiento óptimo de la aplicación web de gestión de tickets, es importante tener en cuenta tanto los requisitos de hardware como los de software:

* + **Sistema Operativo**: Linux (Ubuntu 18.04+ recomendado), macOS, o Windows 10/11.
  + **CPU**: Procesador de al menos 2 núcleos (preferiblemente Intel i3 o superior, o equivalente AMD).
  + **Memoria RAM**: 4GB de RAM como mínimo (8GB recomendados para un mejor rendimiento).
  + **Almacenamiento**: Al menos 20GB de espacio libre en disco para la instalación de dependencias y almacenamiento de la base de datos (más espacio según la cantidad de datos que se gestionen).
  + **Node.js**: Versión 16 o superior.
  + **Base de datos**: MySQL o MariaDB (versión 5.7 o superior).
  + **Servidor Web**: NGINX o Apache (según la configuración del servidor).

# Instalación de Dependencias

Una vez clonado el repositorio de la aplicación, se deben instalar todas las dependencias necesarias para su funcionamiento. Para ello, ejecuta el siguiente comando en el directorio raíz del proyecto:

bash

Copy code npm install

O, de forma abreviada:

bash

Copy code npm i

Este comando instalará todas las dependencias listadas en el archivo package.json, asegurando que la aplicación esté lista para ser ejecutada.

# Variables de Entorno

Es necesario configurar las siguientes variables de entorno para que la aplicación funcione correctamente. Estas variables deben ser almacenadas en un archivo

.env en la raíz del proyecto o configuradas directamente en el entorno de producción:

* + **DB\_DATABASE:** Nombre de la base de datos.
  + **DB\_USER:** Usuario de la base de datos.
  + **DB\_PASS:** Contraseña del usuario de la base de datos.
  + **DB\_HOST:** Host de la base de datos (por ejemplo, localhost o una dirección IP).
  + **DB\_PORT:** Puerto de la base de datos (por defecto, 3306 para MySQL).
  + **MAIL\_HOST:** Servidor SMTP para el envío de correos electrónicos.
  + **MAIL\_PORT:** Puerto del servidor SMTP (por ejemplo, 587).
  + **MAIL\_USER:** Usuario del servidor SMTP (generalmente el correo electrónico que envía).
  + **MAIL\_PASS:** Contraseña para el usuario del servidor SMTP.
  + **URL\_BACKEND:** URL del backend de la aplicación.
  + **SECRET\_KEY:** Clave secreta utilizada para firmar tokens y otras operaciones de seguridad.

# Configuración del Servidor

El proceso más sencillo para configurar el servidor donde se desplegará la aplicación incluye los siguientes pasos:

1. **Instalar Node.js**: Asegúrate de tener Node.js instalado en el servidor. Puedes instalar Node.js con el siguiente comando (en sistemas basados en Ubuntu):

bash

Copy code

sudo apt update

sudo apt install nodejs sudo apt install npm

1. **Clonar el repositorio de la aplicación**: Clona el repositorio de tu aplicación en el servidor:

bash

Copy code

git clone https://github.com/usuario/gestick.git cd gestick

1. **Instalar las dependencias**: Ejecuta el comando npm install para instalar todas las dependencias necesarias.
2. **Configurar las variables de entorno**: Crea un archivo .env en la raíz del proyecto con las variables mencionadas anteriormente.
3. **Iniciar la aplicación**: Inicia la aplicación utilizando npm

bash

Copy code

npm run server npm run dev

1. **Configurar NGINX** (opcional, si estás usando NGINX como servidor web):

Si usas NGINX para servir la aplicación, configura un archivo de servidor para redirigir las solicitudes HTTP al puerto donde está corriendo tu aplicación Node.js (por ejemplo, el puerto 3000).

Configuración básica de NGINX:

nginx Copy code server {

listen 80;

server\_name tu\_dominio.com;

location / {

proxy\_pass http://localhost:3000; proxy\_http\_version 1.1; proxy\_set\_header Upgrade $http\_upgrade; proxy\_set\_header Connection 'upgrade'; proxy\_set\_header Host $host; proxy\_cache\_bypass $http\_upgrade;

}

}

# Reiniciar NGINX:

Después de configurar el archivo, reinicia NGINX para aplicar los cambios:

bash

Copy code

sudo systemctl restart nginx

1. **Opcional: Configuración SSL**: Para habilitar HTTPS en tu servidor, puedes usar **Certbot** para obtener un certificado SSL gratuito de Let's Encrypt:

bash

Copy code

sudo apt install certbot python3-certbot-nginx sudo certbot --nginx

# Conclusión

En resumen, la documentación en aplicaciones como Gestick no solo proporciona una guía clara sobre el uso y las funcionalidades del sistema, sino que también actúa como una herramienta clave para garantizar su mantenimiento y evolución a largo plazo. Una buena documentación mejora la eficiencia, reduce errores, y asegura que el conocimiento no dependa únicamente del equipo original, haciendo que la aplicación sea más accesible y fácil de gestionar.