UNIVERSIDAD DON BOSCO

# Documentación
# INVESTIGACIÓN APLICADA 1

*API REST, DOCKER Y KUBERNETES*

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA EN COMPUTACIÓN

LENGUAJES INTERPRETADOS EN EL SERVIDOR

DOCENTE: INGENIERA KARENS MEDRANO

# ÍNDICE

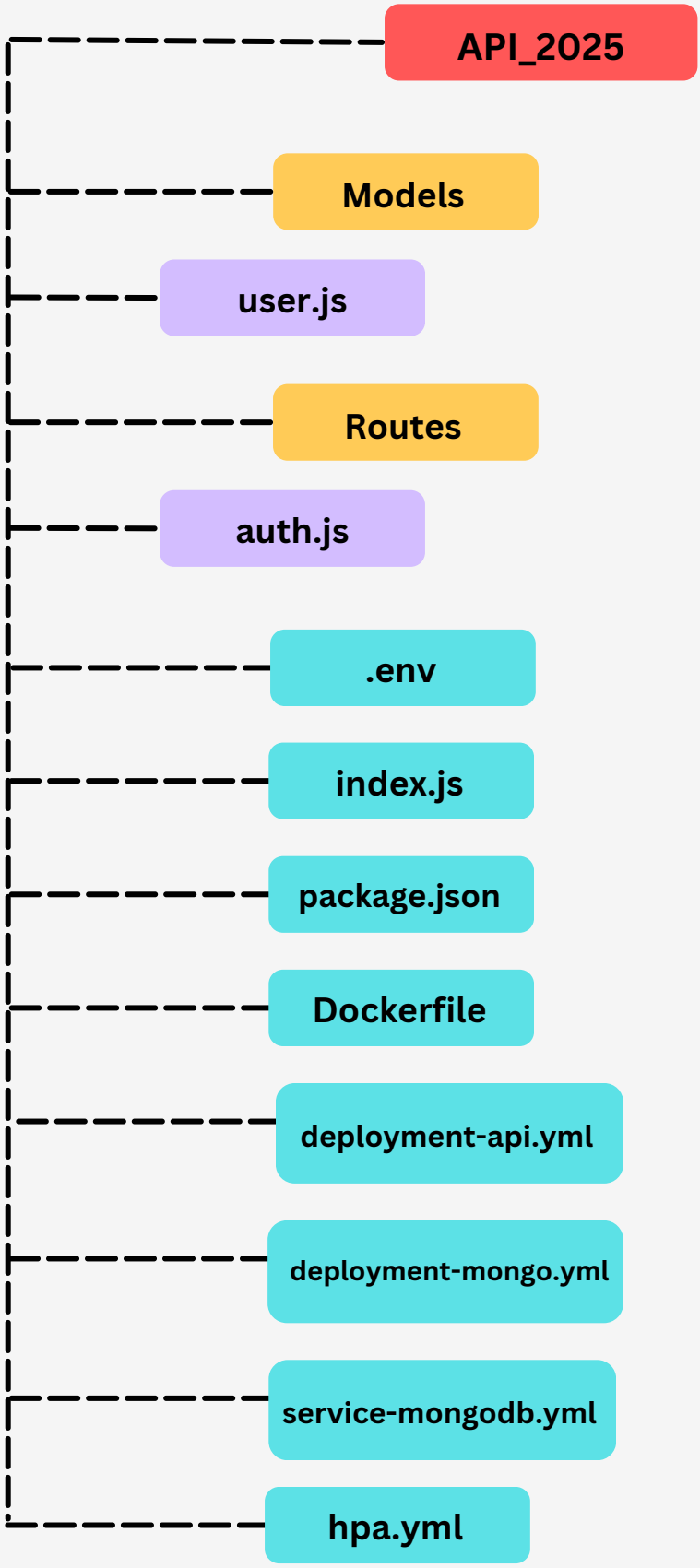
# INTRODUCCIÓN

- La API fue desarrollada en Node.js con Express y conecta a una base de datos MongoDB

## Tecnologías utilizadas

- **Node.js:** Entorno de ejecución para JavaScript en el servidor.
- **Express.js:** Framework para la creación de APIs REST.
- **MongoDB:** Base de datos NoSQL para almacenar usuarios.
- **JWT (JSON Web Token):** Para la autenticación de usuarios.
- **Docker:** Para contenedorización de la API.
- **Kubernetes:** Para la orquestación y escalado de contenedores.

# ESTRUCTURA DEL PROYECTO



# DESARROLLO DE LA API

## Modelo de Usuario (models/User.js)

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');

const UserSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

UserSchema.methods.matchPassword = async function (password) {
  return await bcrypt.compare(password, this.password);
};

UserSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});

const User = mongoose.model('User', UserSchema);
module.exports = User;
```

# DESARROLLO DE LA API

## Rutas de autenticación (routes/auth.js)

```
const express = require('express');
const User = require('../models/User');
const jwt = require('jsonwebtoken');
const router = express.Router();

router.post('/register', async (req, res) => {
  const { username, email, password } = req.body;
  try {
    const userExists = await User.findOne({ email });
      if (userExists) return res.status(400).json({ message:
'El correo ya está registrado' });

        const user = await User.create({ username, email,
password });
        res.status(201).json({ message: 'Usuario registrado
exitosamente', user });
  } catch (error) {
      res.status(500).json({ message: 'Error en el servidor',
error: error.message });
  }
});
```

# DESARROLLO DE LA API

```
router.post('/login', async (req, res) => {
 const { username, password } = req.body;
 try {
 const user = await User.findOne({ username });
  if (!user) return res.status(400).json({ message: 'Usuario no encontrado' });

 const isMatch = await user.matchPassword(password);
   if (!isMatch) return res.status(400).json({ message: 'Contraseña incorrecta' });

     const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });
 res.status(200).json({ message: 'Inicio de sesión exitoso', token });
 } catch (error) {
res.status(500).json({ message: 'Error en el servidor', error: error.message });
 }
});

module.exports = router;
```

# DESARROLLO DE LA API

## Configuración Principal (index.js)

```
const express = require('express');
const mongoose = require('mongoose');
const dotenv = require('dotenv');
const cors = require('cors');
const authRoutes = require('./routes/auth');

dotenv.config();
const app = express();

app.use(express.json());
app.use(cors());
app.use('/api', authRoutes);

mongoose.connect(process.env.MONGO_URI,                    {
useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('MongoDB conectado'))
  .catch(err => console.log('Error de conexión con MongoDB:',
err));

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Servidor ejecutándose en
el puerto ${PORT}`));
```

# CONTENIDO DEL DOCKERFILE

```
FROM node:20-alpine
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install --omit=dev
COPY . .
EXPOSE 5000
CMD ["npm", "run", "start"]
```

**Este Dockerfile crea una imagen ligera de Node.js, instala las dependencias y expone el puerto 5000 para la API.**

# IMPLEMENTACIÓN EN KUBERNETES

## Construcción de la imagen Docker

*Ejecutamos el siguiente comando para crear la imagen de la API:*

docker build -t api-2025 .

## Habilitar Kubernetes en Docker Desktop

1. *Abrir Docker Desktop.*
2. *Ir a Settings > Kubernetes.*
3. *Activar la opción Enable Kubernetes y hacer clic en Apply & Restart.*
4. *Verificar que Kubernetes está corriendo*
5. *kubectl get nodes*

# IMPLEMENTACIÓN EN KUBERNETES

## Despliegue de MongoDB en Kubernetes

Archivo deployment-mongo.yml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mongodb-deployment
spec:
 replicas: 1
 selector:
   matchLabels:
     app: mongodb
 template:
   metadata:
     labels:
       app: mongodb
   spec:
containers:
 - name: mongodb
 image: mongo:6
 ports:
 - containerPort: 27017
```

# IMPLEMENTACIÓN EN KUBERNETES

```
volumeMounts:
 - name: mongo-storage
mountPath: /data/db
 volumes:
 - name: mongo-storage
 emptyDir: {}
```

Archivo service-mongodb.yml:

```
apiVersion: v1
kind: Service
metadata:
 name: mongodb-service
spec:
 selector:
 app: mongodb
 ports:
 - protocol: TCP
 port: 27017
 targetPort: 27017
```

*Aplicamos estos archivos en Kubernetes:*

```
kubectl apply -f deployment-mongo.yml
kubectl apply -f service-mongodb.yml
```

# IMPLEMENTACIÓN EN KUBERNETES

## Despliegue de la API en Kubernetes

Archivo deployment-api.yml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: api-deployment
spec:
 replicas: 2
 selector:
   matchLabels:
     app: api
 template:
   metadata:
     labels:
       app: api
   spec:
     containers:
       - name: api
         image: api-2025
         ports:
           - containerPort: 5000
         env:
           - name: MONGO_URI
             value: "mongodb://mongodb-service:27017/users_db"
           - name: JWT_SECRET
             value: "300901"
```

*Aplicamos el archivo:*

```
kubectl apply -f deployment-api.yml
```

# PRUEBAS Y VERIFICACIÓN

## Verificar que los pods están corriendo

kubectl get pods

## Exponer la API en Kubernetes

Archivo service-api.yml:

```
apiVersion: v1
kind: Service
metadata:
 name: api-service
spec:
 type: LoadBalancer
 ports:
   - port: 5000
     targetPort: 5000
 selector:
   app: api
```

# PRUEBAS Y VERIFICACIÓN

*Aplicamos el servicio:*

kubectl apply -f service-api.yml

Verificamos la IP asignada por Kubernetes:

kubectl get services

## Probar la API con Postman

*Hacemos una petición POST a:*

http://localhost:5000/api/register

Con el siguiente JSON:

```
{
"username": "usuario1",
 "email": "usuario1@email.com",
"password": "123456"
}
```

*Si la API responde correctamente, significa que está funcionando.*

# PRUEBAS Y VERIFICACIÓN

## Escalado Automático con HPA

Archivo hpa.yml:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: api-hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: api-deployment
 minReplicas: 2
 maxReplicas: 5
 metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
averageUtilization: 50
```

# PRUEBAS Y VERIFICACIÓN

*Aplicamos el escalado automático:*

kubectl apply -f hpa.yml

*Simulamos tráfico para probar el escalado:*

kubectl run -it --rm load-generator --image=busybox -- /bin/sh -c "while true; do wget -q -O- http://api-service:5000/api/register; done"

*Verificamos las réplicas activas:*

kubectl get hpa

kubectl get pods

Investigación 1 - Febrero 2025

# PRESENTADO POR


Méndez Parada, Luis Antonio | MP220885

Padilla Ramírez, Alexandra Guadalupe | NR221019

Pineda Fuentes, Geovany Arturo | PF211251

Quintanilla López, José Luis | QL210503

Valencia Rivera, Némesis Alejandra | VR211067