

INFORME DE PROYECTO FINAL

ASIGNATURA:	Programación Web 2				
PROYECTO WEB	PRODEC-AGROREGISTRO				
NOMBRE DE PRÁCTICA:	PROYECTO FINAL	AÑO LECTIVO:	2025 - A	NRO. SEMESTRE:	III
FECHA DE INICIO:	11/07/2024	FECHA FIN:	25/07/2025		
INTEGRANTE(s):					
<ul style="list-style-type: none">• Cornejo Hurtado, Darío Rafael• Cervantes Apaza, Diego Aristides					
DOCENTES:					
ESCOBEDO QUISPE, RICHART SMITH					
CORRALES DELGADO, CARLO JOSE LUIS					

INTEGRANTES:

- CORNEJO HURTADO DARÍO RAFAEL
- CERVANTES APAZA DIEGO ARISTIDES



Resumen

El presente informe detalla el desarrollo de **PRODEC** (Proyecto de Comunicación Digital para el Agricultor), una plataforma web diseñada para optimizar la interacción entre agricultores y clientes mediante una solución tecnológica moderna, accesible y segura. PRODEC forma parte del proyecto final AgroRegistro, implementado como una aplicación distribuida que integra tecnologías como Django, React, PostgreSQL (vía Supabase) y servicios en la nube como Railway y Vercel. El sistema permite el registro y autenticación de múltiples roles de usuario, habilitando funcionalidades específicas como gestión de productos, ofertas, solicitudes y notificaciones. A lo largo del documento se describe la arquitectura técnica, las herramientas utilizadas, la lógica de negocio, el despliegue, y los beneficios del sistema en términos de transformación digital del sector agrícola peruano.

1. Introducción

El proyecto denominado **PRODEC** consiste en una aplicación web diseñada para facilitar la comunicación directa entre agricultores y clientes, permitiendo que ambos puedan registrarse y acceder al sistema según su rol correspondiente. Esta iniciativa surge como respuesta a la necesidad de modernizar el sector agrícola peruano, promoviendo una interacción eficiente, segura y directa entre los actores principales del mercado agrícola.

PRODEC se integra como una solución tecnológica dentro del proyecto final **AgroRegistro**, desarrollado como parte del curso de Programación Web 2. Su objetivo fundamental es ofrecer una plataforma digital de fácil acceso que permita a los agricultores gestionar sus productos y ofertas, mientras que los clientes pueden visualizar, solicitar e interactuar con dichas ofertas de forma personalizada. Asimismo, el sistema incluye la figura del administrador, quien supervisa y controla los distintos aspectos operativos de la plataforma.

Desde su concepción, PRODEC fue diseñado con una arquitectura robusta, utilizando herramientas modernas como **Django** y **Django REST Framework** para el backend, **React JS** para el frontend y **PostgreSQL** como motor de base de datos, desplegado a través de **Supabase**. El despliegue completo se realizó utilizando plataformas como **Railway** (para el backend) y **Vercel** (para el frontend), garantizando disponibilidad en la nube, acceso remoto y escalabilidad.

El presente documento está estructurado para brindar una visión detallada del proceso de desarrollo del sistema. Incluye la descripción técnica de la base de datos, la configuración de las rutas API, la lógica de autenticación basada en JWT, el manejo de roles y permisos, el diseño de componentes en React, así como la documentación de las herramientas y servicios utilizados durante el ciclo de desarrollo.

Además, se presentan enlaces públicos de acceso, credenciales de prueba, capturas de pantalla y otros recursos que permiten validar el cumplimiento de los criterios definidos en la rúbrica del curso. Con esta iniciativa, se busca aportar una solución funcional y replicable que promueva el desarrollo sostenible del agro a través de la transformación digital.

1.1. Herramientas utilizadas

Para el desarrollo del sistema web AgroRegistro (PRODEC), se utilizaron diversas tecnologías y herramientas que permitieron estructurar el proyecto en componentes bien definidos, asegurando su funcionalidad, escalabilidad y experiencia de usuario.

Lenguajes de programación:

- **Python 3.11:** Utilizado para todo el backend, lógica del negocio y definición de modelos.
- **JavaScript (ES6):** Para desarrollar el frontend de la aplicación con React.

Frameworks y librerías:

- **Django REST Framework (DRF):** Para construir una API RESTful robusta y escalable.
- **React JS:** Para el desarrollo del frontend con una SPA (Single Page Application).
- **Fetch API:** Usada en el frontend para consumir endpoints del backend vía HTTP.
- **JWT (JSON Web Token):** Implementado para autenticación segura entre frontend y backend.

Base de datos:

- **PostgreSQL (vía Supabase):** Motor de base de datos en la nube. Supabase se utilizó para almacenar datos, gestionar autenticación de usuarios y explorar registros fácilmente.

Servicios y plataformas en la nube:

- **Railway:** Plataforma utilizada para el despliegue del backend (Django + DRF), ofreciendo un entorno de producción estable y soporte para CI/CD.
- **Vercel:** Utilizada para el despliegue del frontend desarrollado en React. Permite rutas dinámicas y actualizaciones automáticas.
- **Supabase:** Servicio backend-as-a-service usado como base de datos Postgres y sistema de autenticación básica.

Herramientas de desarrollo:

- **Visual Studio Code (VS Code):** Editor de código principal durante el desarrollo.

- **Git:** Sistema de control de versiones para gestión de cambios y ramas del proyecto.
- **GitHub:** Repositorio online para alojar y colaborar en el código del equipo.
- **Postman:** Utilizado para probar endpoints de la API REST en el backend.
- **Google Meet, WhatsApp y Google Drive:** Herramientas de comunicación, coordinación y almacenamiento compartido entre los integrantes del equipo.

1.2. Requisitos Funcionales

- **RF01** – El sistema debe permitir el registro y autenticación de tres tipos de usuarios: administrador, agricultor y cliente.
- **RF02** – El agricultor puede registrar productos, crear ofertas y responder solicitudes de clientes.
- **RF03** – El cliente puede visualizar ofertas, agregar productos al carrito, enviar solicitudes y recibir respuestas.
- **RF04** – El administrador tiene acceso total para gestionar usuarios, productos y verificar el funcionamiento general del sistema.
- **RF05** – Las operaciones CRUD deben estar disponibles según el rol del usuario autenticado.
- **RF06** – El sistema debe descontar automáticamente el stock al añadir productos al carrito.
- **RF07** – Debe existir un sistema de notificaciones para avisar a los agricultores cuando un cliente agrega productos suyos al carrito.
- **RF08** – El cliente debe poder ver las respuestas a sus solicitudes directamente en su panel.
- **RF09** – El sistema debe garantizar la integridad de los datos mediante validaciones en backend.

2. Desarrollo

2.1. Modelo de datos

Base de Datos: Se utiliza una base de datos relacional como PostgreSQL, alojada en Supabase. Django ORM (Object-Relational Mapping) facilita la interacción con la base de datos, permitiendo mapear los modelos de Python directamente a las tablas SQL.

Tablas y Relaciones: A través de Django, se definieron los modelos que representan entidades como usuarios, productos, ofertas, solicitudes, entre otros. Estas tablas mantienen relaciones clave para preservar la integridad y el comportamiento del sistema.

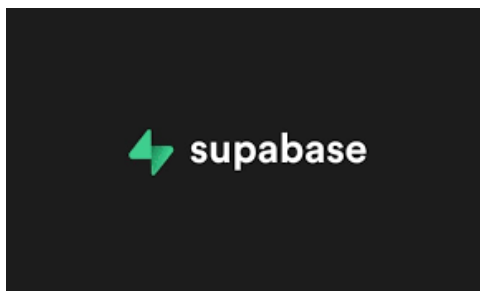


Figura 1: Logo oficial de Supabase, la plataforma utilizada para administrar la base de datos del proyecto.

Una de las ventajas que ofrece esta plataforma de base de datos es la fácil administración y visualización de las tablas, lo cual facilita mucho el cuidado de los modelos y plantillas utilizadas en producción.

Además, permite ver cada uno de los datos actualizados permanentemente al ingresar nueva información, ya sea de manera local o remota en el programa desplegado. En este caso, por ejemplo, se observa la tabla de los usuarios y sus atributos principales.

[illegible]

Figura 3: Exploración de registros desde Supabase

2.2. Backend: Consideraciones y Configuración

El backend del sistema está desarrollado con Django, un framework robusto y escalable en Python. Este se encargó de gestionar la lógica del servidor, el acceso a datos mediante modelos, y la construcción de API RESTful que permiten la comunicación entre el cliente (frontend) y la base de datos.

Configuración Inicial: Para comenzar, se realizó la configuración básica del proyecto Django en el archivo `settings.py`. En este archivo, uno de los pasos más importantes fue establecer correctamente la conexión con Supabase, la plataforma donde se aloja la base de datos PostgreSQL.

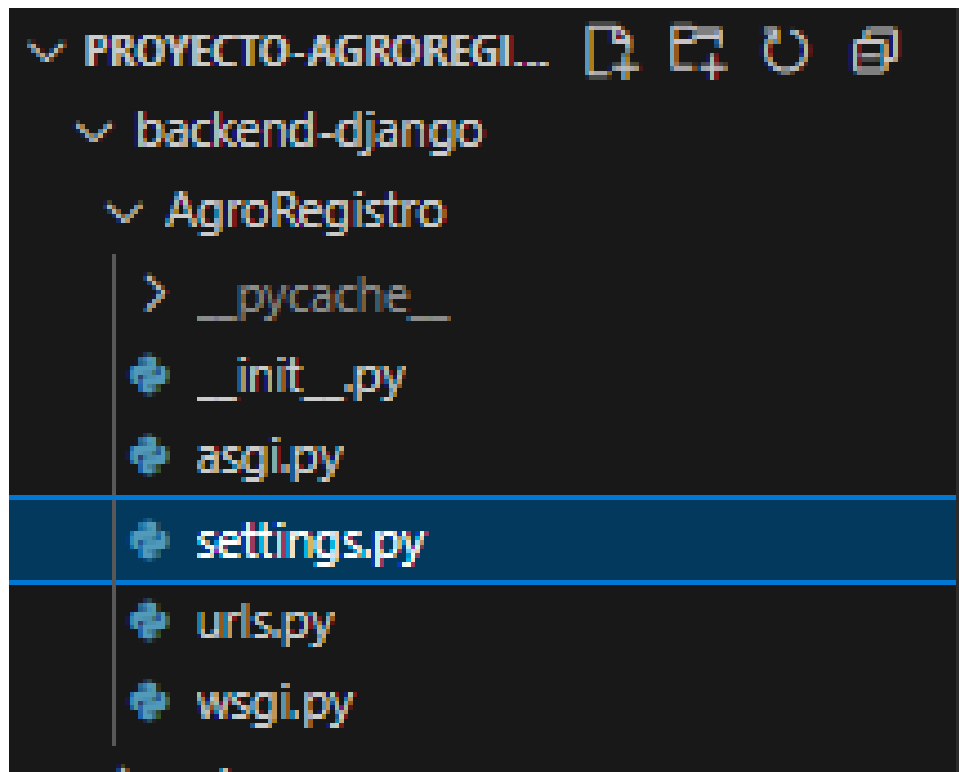


Figura 4: Configuración de conexión a Supabase dentro del archivo `settings.py`.

Parámetros esenciales del `settings.py`: Entre las variables configuradas en este archivo destacan:

- `DEBUG`: Activado o desactivado según entorno (desarrollo o producción).
- `ALLOWED_HOSTS`: Dominios permitidos para servir el backend.
- `DATABASES`: Parámetros de conexión con Supabase (nombre, usuario, host, puerto).
- `INSTALLED_APPS`: Registro de las apps utilizadas en el proyecto, incluyendo DRF y CORS.

```

backend-django > AgroRegistro > settings.py > ...
1  import logging
2  import sys
3  from pathlib import Path
4  from decouple import config
5  import dj_database_url
6  from corsheaders.defaults import default_headers
7
8  # BASE_DIR sirve para rutas relativas dentro del proyecto
9  BASE_DIR = Path(__file__).resolve().parent.parent
10
11 # Seguridad y entorno
12 SECRET_KEY = config('SECRET_KEY', default='fallback-key')
13 DEBUG = config('DEBUG', default=False, cast=bool)
14 ALLOWED_HOSTS = ['*'] # Railway acepta esto
15
16 # Aplicaciones instaladas
17 INSTALLED_APPS = [
18     'django.contrib.admin',
19     'django.contrib.auth',
20     'django.contrib.contenttypes',
21     'django.contrib.sessions',
22     'django.contrib.messages',
23     'django.contrib.staticfiles',
24
25     # Terceros
26     'rest_framework',
27     'rest_framework_simplejwt',
28     'corsheaders', # necesario para permitir CORS
29
30     # Tu app
31     'api',
32 ]
33
34 # MODELO DE USUARIO PERSONALIZADO
35 AUTH_USER_MODEL = 'api.Usuario'
36
37 # Middleware
38 MIDDLEWARE = [
39     'corsheaders.middleware.CorsMiddleware', # DEBE ir primero
40     'django.middleware.security.SecurityMiddleware',
41     'django.contrib.sessions.middleware.SessionMiddleware',
42     'django.middleware.common.CommonMiddleware',
43     'django.middleware.csrf.CsrfViewMiddleware',
44     'django.contrib.auth.middleware.AuthenticationMiddleware',
45     'django.contrib.messages.middleware.MessageMiddleware',
46     'django.middleware.clickjacking.XFrameOptionsMiddleware',
47 ]
48
49 ROOT_URLCONF = 'AgroRegistro.urls'
50
51 TEMPLATES = [
52     {
53         'BACKEND': 'django.template.backends.django.DjangoTemplates',
54         'DIRS': [],
55         'APP_DIRS': True,
56         'OPTIONS': {
57             'context_processors': [
58                 'django.template.context_processors.request',
59                 'django.contrib.auth.context_processors.auth',
60                 'django.contrib.messages.context_processors.messages',
61             ],
62         },
63     },
64 ]
65
66 WSGI_APPLICATION = 'AgroRegistro.wsgi.application'
67
68 # Base de datos
69 DATABASES = {
70     'default': dj_database_url.config(
71         default=config('DATABASE_URL'),
72         conn_max_age=600,
73         ssl_require=True
74     )
75 }
76
77 # Validación de contraseñas
78 AUTH_PASSWORD_VALIDATORS = [
79     {'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator'},
80     {'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator'},
81     {'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator'},
82     {'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator'},
83 ]
84

```

Figura 5: Configuración clave en `settings.py`: seguridad, apps instaladas y conexión.

Instalación de Django REST Framework (DRF): Para construir las APIs que conectan el frontend con la base de datos, se instaló Django REST Framework, que permite serializar modelos, manejar peticiones HTTP y estructurar controladores (views) de forma eficiente.

Variables de entorno (.env): Se creó en la raíz del proyecto un archivo `.env` que contiene las variables sensibles necesarias para el entorno de desarrollo y producción: claves secretas, configuración de la base de datos, token de Supabase, entre otros.

```
backend-django > cat .env
1 SECRET_KEY=django-insecure-rpsh_dirkha=6@6$9gs8o13yq6rvu*+uav-1vk#yoaxs"cy
2 DEBUG=False
3 ALLOWED_HOSTS=127.0.0.1,localhost,0.0.0.0,web-production-2486a.up.railway.app
4 DATABASE_URL=postgresql://postgres.1dvnkitceutqghoym:DarioSuper062825@aws-0-us-east-2.pooler.supabase.com:5432/postgres
5 CORS_ALLOW_ALL_ORIGINS=True
```

Figura 8: Contenido del archivo `.env`, esencial para el despliegue del proyecto.

Despliegue en la nube con Railway: Para desplegar el backend con una URL pública y funcional, se utilizó la plataforma **Railway**, la cual fue elegida por las siguientes razones:

- Permite subir proyectos directamente desde GitHub con integración continua.
- Ofrece entorno gratuito con PostgreSQL incorporado.
- Configuración sencilla de variables de entorno desde el panel web.
- Generación automática de URL pública accesible desde cualquier navegador o cliente.
- Escalabilidad fácil y monitoreo en tiempo real.

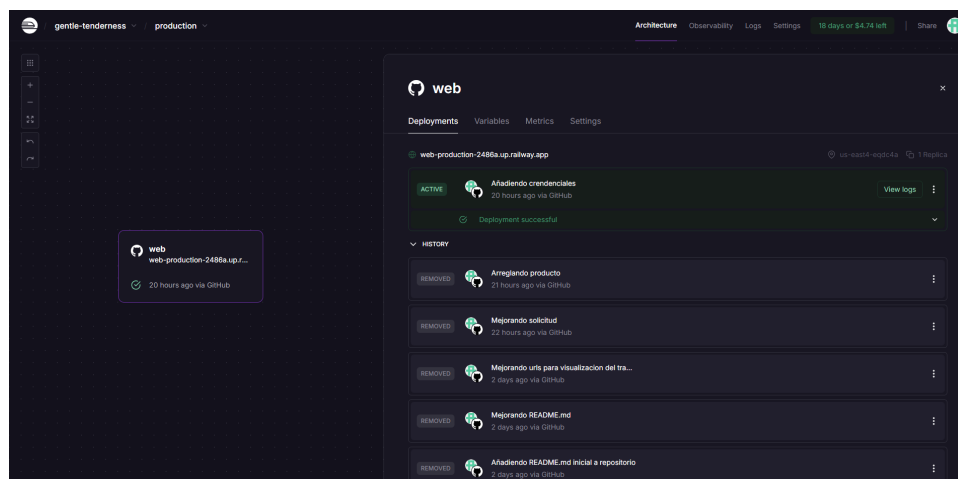


Figura 9: Railway: plataforma utilizada para desplegar el backend de manera estable y accesible.

Organización de Modelos: Para mantener una arquitectura limpia y modular, se creó una carpeta específica dentro del proyecto para contener todos los modelos. Esto permite realizar migraciones ordenadas, reutilizar lógica común y estructurar el desarrollo de forma escalable.

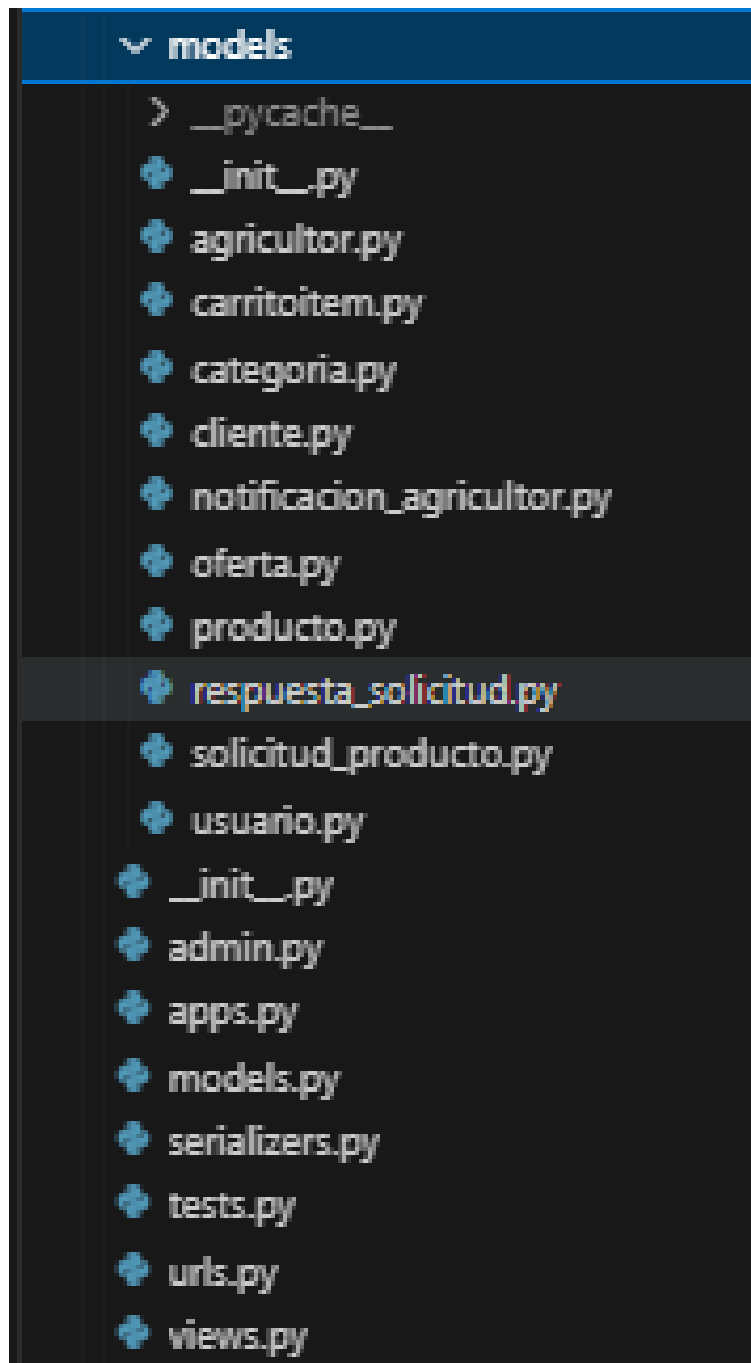


Figura 10: Estructura de carpetas para los modelos en Django, con buena organización.

2.3. API REST

La API REST de **AgroRegistro** fue desarrollada utilizando el *Django REST Framework*. Permite gestionar entidades como usuarios, productos, ofertas, carritos y solicitudes. Esta arquitectura garantiza una comunicación estructurada entre el frontend y el backend a través de operaciones HTTP estándar (GET, POST, PUT, DELETE).

Estructura general y autenticación: El archivo `urls.py` del backend configura los principales endpoints a través de `routers` y vistas. Se emplea autenticación mediante **JWT (JSON Web Tokens)**, permitiendo que usuarios autenticados puedan acceder a funcionalidades específicas.

```

backend-django > api > ✎ urls.py > ...
1  from django.urls import path, include
2  from rest_framework.routers import DefaultRouter
3  from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView
4  from .views import (
5      RegisterView, UserView,
6      AgricultorViewSet, CategoriaViewSet,
7      ProductoViewSet, OfertaViewSet,
8      ClienteViewSet, CarritoItemViewSet,
9      SolicitudProductoViewSet, RespuestaSolicitudViewSet, NotificacionAgricultorViewSet
10 )
11
12 # Routers
13 router = DefaultRouter()
14 router.register(r'agricultores', AgricultorViewSet)
15 router.register(r'categorias', CategoriaViewSet)
16 router.register(r'productos', ProductoViewSet)
17 router.register(r'ofertas', OfertaViewSet)
18 router.register(r'clientes', ClienteViewSet)
19 router.register(r'carrito', CarritoItemViewSet)
20 router.register(r'solicitudes', SolicitudProductoViewSet)
21 router.register(r'respuestas', RespuestaSolicitudViewSet)
22 router.register(r'notificaciones', NotificacionAgricultorViewSet)
23
24 # URL Patterns
25 urlpatterns = [
26     path('', include(router.urls)),
27     path('register/', RegisterView.as_view(), name='register'),
28     path('user/', UserView.as_view(), name='user'),
29     path('token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
30     path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
31 ]

```

Figura 11: Estructura de rutas en urls.py

Registro condicional de usuarios: Al momento del registro, el sistema crea dinámicamente un perfil de `Cliente` o `Agricultor` según el tipo de usuario.

```

def create(self, validated_data):
    tipo = validated_data.pop('tipo_usuario')
    password = validated_data.pop('password')

    # Extrae campos adicionales (no son parte de Usuario)
    nombre = validated_data.pop('nombre', '')
    telefono = validated_data.pop('telefono', '')
    direccion = validated_data.pop('direccion', '')
    ruc = validated_data.pop('ruc', '')
    empresa = validated_data.pop('empresa', '')

    usuario = Usuario(**validated_data)
    usuario.set_password(password)
    usuario.tipo_usuario = tipo
    usuario.save()

    if tipo == 'cliente':
        Cliente.objects.create(
            usuario=usuario,
            nombre=nombre,
            direccion=direccion,
            telefono=telefono
        )
    elif tipo == 'agricultor':
        Agricultor.objects.create(
            usuario=usuario,
            nombre=nombre,
            telefono=telefono,
            departamento='',
            provincia='',
            distrito='',
        )

    return usuario

```

Figura 12: Lógica del registro condicional en `serializers.py`

Gestión de productos y ofertas: Los agricultores pueden crear y administrar productos y ofertas, mientras que los clientes pueden visualizarlas. La relación entre productos y agricultores se mantiene mediante serialización anidada y protección con permisos de autenticación.

```

110 class ProductoViewSet(viewsets.ModelViewSet):
111     queryset = Producto.objects.all()
112     serializer_class = ProductoSerializer
113     permission_classes = [IsAuthenticated]
114

```

Figura 13: Vista del ViewSet de productos

```

class OfertaSerializer(serializers.ModelSerializer):
    agricultor = AgricultorSerializer(read_only=True)
    agricultor_id = serializers.PrimaryKeyRelatedField(
        queryset=Agricultor.objects.all(), source='agricultor', write_only=True
    )
    producto = ProductoSerializer(read_only=True)
    producto_id = serializers.PrimaryKeyRelatedField(
        queryset=Producto.objects.all(), source='producto', write_only=True
    )

    class Meta:
        model = Oferta
        fields = ['id', 'agricultor', 'agricultor_id', 'producto', 'producto_id', 'descripcion', 'precio', 'stock']

```

Figura 14: Serializador anidado para ofertas

Carrito con validación inteligente: El carrito está diseñado para evitar duplicados. Si un cliente intenta añadir una oferta ya existente, simplemente se actualiza la cantidad. Esto evita redundancias y mejora la experiencia del usuario.

```
def perform_create(self, serializer):
    try:
        usuario = self.request.user
        print(f"👤 Usuario autenticado: {usuario} (ID: {usuario.id})")

        cliente = Cliente.objects.get(usuario=usuario)
        print(f"👤 Cliente encontrado: {cliente} (ID: {cliente.id})")

        oferta = serializer.validated_data['oferta']

        existente = CarritoItem.objects.filter(cliente=cliente, oferta=oferta).first()
        if existente:
            existente.cantidad += serializer.validated_data.get('cantidad', 1)
            existente.save()
            print(f"🟡 Ya existía: se actualizó la cantidad a {existente.cantidad}")
        else:
            carrito_item = serializer.save(cliente=cliente)
            print(f"✅ CarritoItem creado con ID: {carrito_item.id}")

            # 📢 Crear notificación para el agricultor
            from .models import NotificacionAgricultor # Import local para evitar ciclos
            mensaje = f"{cliente.nombre} ha añadido tu oferta de {oferta.producto.nombre} al carrito."
            NotificacionAgricultor.objects.create(
                agricultor=oferta.agricultor,
                cliente=cliente,
                oferta=oferta,
                mensaje=mensaje
            )
            print("📢 Notificación enviada al agricultor.")

    except Cliente.DoesNotExist:
        print(f"❌ Cliente no encontrado para el usuario ID: {usuario.id}")
        raise serializers.ValidationError("Cliente no encontrado para el usuario.")

    except IntegrityError as e:
        print(f"❌ Error de integridad: {str(e)}")
        raise serializers.ValidationError("Este producto ya está en tu carrito.")

    except Exception as e:
        print(f"🔥 Error interno en perform_create: {str(e)}")
        raise
```

Figura 15: Lógica para añadir al carrito sin duplicados

Solicitudes personalizadas: Los clientes pueden realizar solicitudes específicas sobre productos, y los agricultores pueden responder a estas desde su sesión autenticada.

```
141 class RespuestaSolicitudSerializer(serializers.ModelSerializer):
142     solicitud = SolicitudProductoSerializer(read_only=True)
143     solicitud_id = serializers.PrimaryKeyRelatedField(
144         queryset=SolicitudProducto.objects.all(), source='solicitud', write_only=True
145     )
146     agricultor = AgricultorSerializer(read_only=True)
147     agricultor_id = serializers.PrimaryKeyRelatedField(
148         queryset=Agricultor.objects.all(), source='agricultor', write_only=True
149     )
150
151     class Meta:
152         model = RespuestaSolicitud
153         fields = ['id', 'solicitud', 'solicitud_id', 'agricultor', 'agricultor_id', 'mensaje', 'fecha_respuesta']
154
```

Figura 16: Vista de serialización de una respuesta a solicitud

URLs en la nube:

- URL Backend Admin: <https://web-production-2486a.up.railway.app/admin/>
- URL API Backend: <https://web-production-2486a.up.railway.app/api/>

Credenciales de prueba para administrador:

- Usuario: admin
- Contraseña: 1234

2.4. Frontend React y despliegue en la nube

Tecnologías y despliegue: El frontend de **AgroRegistro** fue desarrollado utilizando *React*, una biblioteca moderna de JavaScript para construir interfaces de usuario interactivas y reactivas. React permite organizar la interfaz en componentes reutilizables y facilita el manejo eficiente del estado y la navegación, lo que se traduce en una experiencia de usuario fluida y responsiva.

Para el despliegue de la aplicación, se utilizó la plataforma *Vercel*, que ofrece un servicio de hosting especializado en aplicaciones frontend modernas como React. Vercel permite integraciones sencillas con repositorios, despliegues automáticos tras cada cambio y una infraestructura global que mejora la velocidad de carga mediante distribución CDN. Esto asegura que la aplicación sea accesible de manera rápida y confiable desde cualquier ubicación geográfica.

Estructura del proyecto frontend: La organización del proyecto sigue una estructura modular y clara, facilitando la escalabilidad y mantenimiento:

- **src/**: carpeta raíz que contiene el archivo principal `App.js`, donde se configuran las rutas, estados globales y lógica principal de navegación y autenticación.
- **components/**: carpeta que alberga todos los componentes reutilizables, tales como formularios de login, registro, dashboards, listas y formularios de interacción.
- Enrutamiento implementado con la librería **react-router-dom**, permitiendo navegación SPA (Single Page Application) sin recargas completas de página.
- Manejo de estado y autenticación utilizando hooks de React y almacenamiento local (**localStorage**) para persistencia de sesión.
- Comunicación con la API backend a través de `fetch` con manejo de tokens JWT para garantizar seguridad en las solicitudes.

Capturas de pantalla de la aplicación: A continuación se presentan las principales interfaces de usuario que componen el frontend:



Figura 17: Pantalla de Inicio: punto de entrada que brinda acceso a las distintas funcionalidades y roles de usuario.

Figura 18: Login para Agricultor y Cliente: formulario para autenticación de usuarios

Registro de Agricultor


Usuario	Contraseña
Nombre	Teléfono
Departamento	Provincia
Distrito	Registrarse

Figura 19: Registro de Agricultor: formulario para crear nuevas cuentas con información específica como ubicación y contacto.

Registro de Cliente


Usuario	Contraseña
Nombre completo	Dirección
Teléfono	Registrarse

Figura 20: Registro de Cliente: formulario para crear nuevas cuentas con datos personales y contacto.

Bienvenido al Panel del Cliente 

Desde aquí puedes ver tus datos personales y explorar las ofertas disponibles.


[Cerrar Sesión](#)


 **Mis Datos**


Nombre: Julian Alvarez


Dirección: Rosario Central 123


Teléfono: 985632145


Ofertas Disponibles 


 **Ofertas disponibles**


-  **Col**


 Stock: 50


 Descripción: Recien cosechada y fresca.


 Precio: S/ 2.00


 Agricultor: Magaly Medina


[Añadir al carrito](#)
-  **Cebolla**


 Stock: 52


 Descripción: Cebolla (israel, no picante y agradable


 Precio: S/ 2.00


 Agricultor: Juancho Perez


[Añadir al carrito](#)
-  **Orégano**


 Stock: 2


 Descripción: Orégano natural


 Precio: S/ 2.00


 Agricultor: Angel Goat


[Añadir al carrito](#)
-  **Cebada**


 Stock: 24


 Descripción: Ajo fresco para cocina


 Precio: S/ 1.90


 Agricultor: Roberto Chinchay

[Añadir al carrito](#)
-  **Papaya**

 Stock: 80

 Descripción: Zanahoria orgánica recién cosechada

 Precio: S/ 1.80

 Agricultor: Ana Huamán

[Añadir al carrito](#)

Figura 21: Panel del Cliente: dashboard personalizado que permite explorar ofertas, gestionar el carrito, enviar solicitudes y ver respuestas.

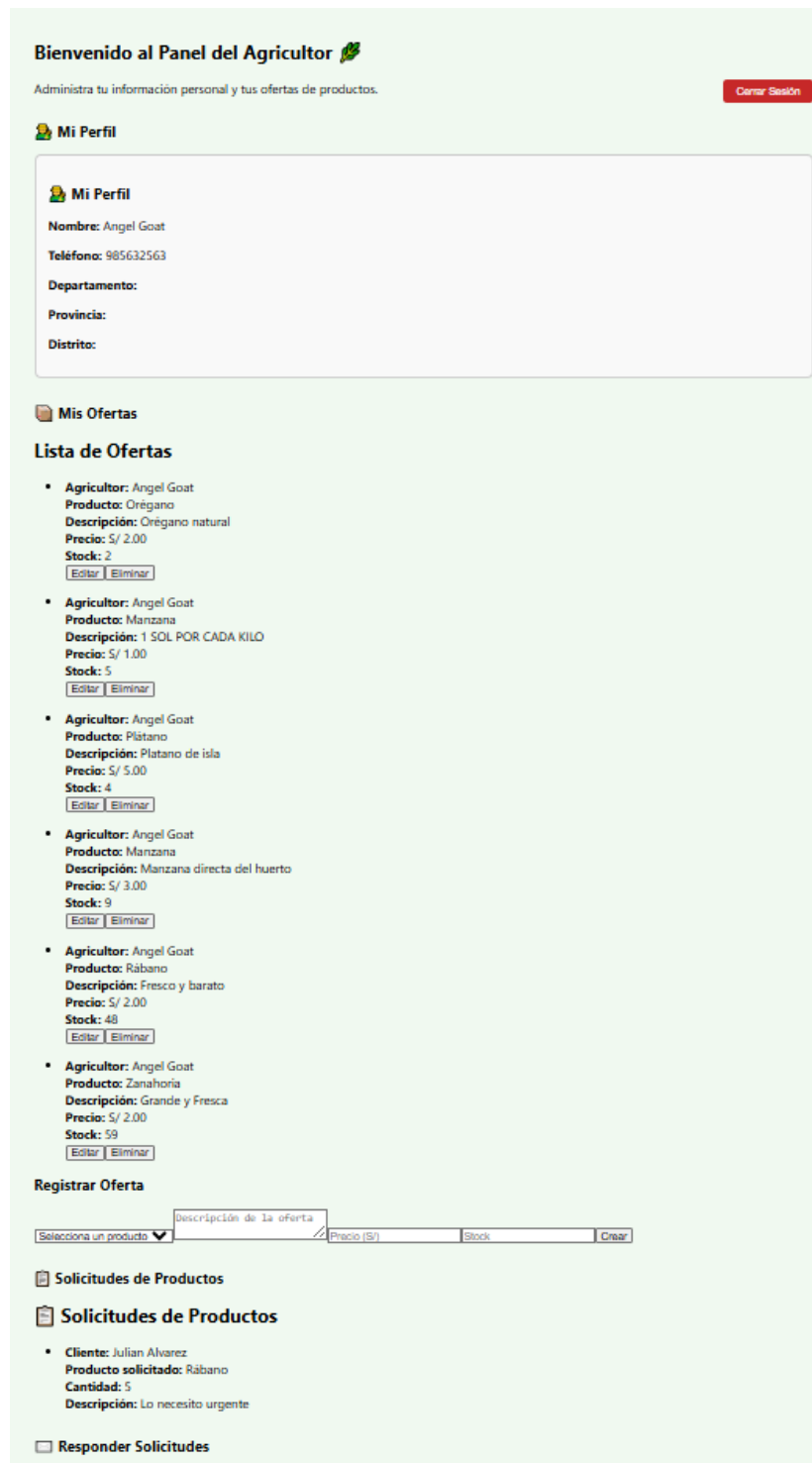


Figura 22: Panel del Agricultor: dashboard personalizado para administrar perfil, ofertas, solicitudes de clientes y notificaciones.

Descripción funcional de los paneles:

2.40a. *Panel del Agricultor:* Este panel está diseñado para que el agricultor gestione toda su actividad dentro de la plataforma. Incluye:

- Gestión del perfil personal con edición de datos relevantes.
- Administración de ofertas de productos mediante un sistema intuitivo para creación, edición y eliminación.
- Visualización y respuesta a solicitudes personalizadas de clientes, facilitando la comunicación directa.

- Consulta de notificaciones para estar al tanto de interacciones y eventos importantes.

2.40b. *Panel del Cliente:* El cliente cuenta con un panel desde donde puede:

- Consultar y actualizar su información personal.
- Explorar las ofertas activas disponibles de diferentes agricultores.
- Gestionar su carrito de compras, evitando duplicados y actualizando cantidades de forma eficiente.
- Enviar solicitudes específicas relacionadas a productos y recibir respuestas desde el panel.

Acceso y credenciales de prueba: **URL de la aplicación frontend en la nube:**

<https://agroregistro-frontend.vercel.app/>

Credenciales para pruebas:

- **Administrador:**
Usuario: `admin2`
Contraseña: 1234
- **Agricultor:**
Usuario: `agricultor`
Contraseña: 1234
- **Cliente:**
Usuario: `cliente01`
Contraseña: 1234

2.5. AJAX y Operaciones Asíncronas

La interacción entre el frontend y el backend de **AgroRegistro** se realiza principalmente a través de peticiones asíncronas utilizando AJAX (Asynchronous JavaScript and XML), en nuestro caso con la función `fetch` de JavaScript dentro de componentes React. Este enfoque permite que las operaciones se ejecuten en segundo plano, evitando recargas completas de la página y brindando una experiencia fluida y responsiva al usuario.

Fundamentos del uso de AJAX en AgroRegistro: AJAX permite a la aplicación cliente solicitar o enviar datos al servidor de manera dinámica y asíncrona, manteniendo la interfaz actualizada sin interrupciones visibles. Esto se logra mediante llamadas HTTP (GET, POST, DELETE, etc.) que intercambian información en formato JSON.

En AgroRegistro, estas peticiones son esenciales para:

- Autenticación y validación segura de usuarios mediante tokens JWT enviados en cabeceras HTTP.
- Consulta de datos del usuario y sus recursos (clientes, carritos, ofertas).
- Actualización dinámica de elementos del UI como el carrito de compras o paneles personalizados.
- Manejo de errores y estados para mostrar mensajes útiles al usuario.

Ejemplo práctico: Componente `CarritoCliente`: Para ilustrar el uso de AJAX, se presenta el componente React `CarritoCliente`, responsable de manejar el carrito de compras del cliente.

2.50a. *Flujo principal:*

1. Obtención del cliente

Al cargar el componente, se hace una petición `GET` a la API `/api/clientes/` para obtener el cliente asociado al usuario autenticado (según su token JWT). Esto se realiza en un `useEffect` que depende del token y el `userId` almacenado localmente.

La respuesta devuelve un listado de clientes, de donde se selecciona el que corresponde al usuario activo.

```
// Paso 1: obtener ID del cliente desde el backend
useEffect(() => {
  const fetchCliente = async () => {
    if (!token || !userId) return;

    try {
      const res = await fetch('https://web-production-2486a.up.railway.app/api/clientes/', {
        headers: {
          Authorization: `Bearer ${token}`
        }
      });

      if (!res.ok) throw new Error('Token inválido o expirado al obtener cliente');

      const data = await res.json();
      const cliente = data.find(c => c.usuario.id === userId);
      if (cliente) {
        setClienteId(cliente.id);
      } else {
        setError('No se encontró el cliente asociado al usuario');
      }
    } catch (err) {
      console.error('✗ Error al obtener cliente:', err);
      setError(err.message);
    }
  };

  fetchCliente();
}, [token, userId]);
```

Figura 23: Petición para obtener cliente desde el frontend React usando `fetch`.

2. Carga del carrito

Con el ID del cliente disponible, se hace otra petición GET a `/api/carrito/?cliente=<clienteId>` para obtener los productos agregados al carrito. Los datos se almacenan en el estado `carrito` del componente y se reflejan en la interfaz.

```
// Paso 2: función reutilizable para obtener el carrito
const fetchCarrito = async () => {
  if (!clienteId) return;

  console.log("🟡 Usando token:", token);
  console.log("🟢 Cliente ID en frontend:", clienteId);

  try {
    const res = await fetch('https://web-production-2486a.up.railway.app/api/carrito/?cliente=${clienteId}', {
      headers: { Authorization: `Bearer ${token}` }
    });

    if (!res.ok) throw new Error('Error al obtener carrito');

    const data = await res.json();
    console.log("🟡 Datos recibidos del carrito:", data);

    setCarrito(data);
  } catch (err) {
    console.error('✗ Error al cargar carrito:', err);
    setError(err.message);
  }
};
```

Figura 24: Datos JSON recibidos del carrito tras la petición asíncrona.

3. Eliminación de productos del carrito

Cuando el usuario decide eliminar un producto, se realiza una petición DELETE a `/api/carrito/<id>/`. Si es exitosa, el estado se actualiza para eliminar ese producto de la lista visible, sin recargar la página.

```

// Paso 4: eliminar del carrito
const eliminarDelCarrito = async (id) => {
  const confirm = window.confirm('¿Eliminar este producto del carrito?');
  if (!confirm) return;

  try {
    const res = await fetch('https://web-production-2486a.up.railway.app/api/carrito/${id}/', {
      method: 'DELETE',
      headers: {
        Authorization: 'Bearer ${token}'
      }
    });

    if (res.ok) {
      setCarrito(prev => prev.filter(item => item.id !== id));
    } else {
      throw new Error('No se pudo eliminar del carrito');
    }
  } catch (err) {
    alert(err.message);
  }
};

const total = carrito.reduce((sum, item) => {
  const precio = parseFloat(item.oferta?.precio || 0);
  return sum + precio * item.cantidad;
}, 0);

return (
  <div style={styles.carrito}>
    <h3>MI Carrito</h3>
    {error && <p style={{ color: 'red' }}>{error}</p>}
    {carrito.length === 0 ? (
      <p>No has añadido productos aún.</p>
    ) : (
      <ul>
        {carrito.map(item => (
          <li key={item.id} style={styles.item}>
            <strong>{item.oferta?.producto?.nombre || 'Producto desconocido'}</strong> (x{item.cantidad})<br />
            Precio unitario: $/ {item.oferta?.precio}<br />
            Total: $/ {(parseFloat(item.oferta?.precio || 0) * item.cantidad).toFixed(2)}<br />
            <button onClick={() => eliminarDelCarrito(item.id)}>Eliminar</button>
          </li>
        ))}
      </ul>
      <p><strong>Total del carrito: $/ {total.toFixed(2)}</strong></p>
      <button disabled>Finalizar compra (pendiente)</button>
    )}
  </div>
);

```

Figura 25: Operación DELETE para eliminar un producto y actualización del estado en React.

2.50b. *Manejo de errores y estado:* Durante las peticiones, se utilizan bloques `try/catch` para capturar errores y mostrarlos al usuario con mensajes claros. Por ejemplo, si el token es inválido o el servidor responde con error, se avisa al usuario para que tome acción, como volver a autenticarse.

Tecnologías y estrategias implementadas:

- **fetch API:** Uso nativo en JavaScript para realizar peticiones HTTP con soporte para promesas, facilitando la gestión asíncrona.
- **Hooks React (useEffect y useState):** Para sincronizar la carga y actualización de datos con el ciclo de vida del componente, garantizando que las operaciones asíncronas se ejecuten cuando es necesario.
- **Token JWT en cabeceras HTTP:** Seguridad en la comunicación asegurando que solo usuarios autenticados puedan acceder y modificar recursos protegidos.
- **Persistencia local (localStorage):** Se almacenan datos del usuario para mantener sesión activa y evitar múltiples inicios de sesión innecesarios.
- **Confirmaciones de usuario:** Antes de eliminar un producto, se muestra un diálogo de confirmación para evitar acciones accidentales.

Beneficios y experiencia del usuario: Este patrón de comunicación garantiza:

- Respuesta inmediata del frontend, mejorando la fluidez y percepción de la aplicación.
- Menor carga en el servidor, ya que solo se hacen peticiones puntuales según interacción.
- Flexibilidad para ampliar funcionalidades sin afectar la estructura principal del frontend.
- Facilidad para manejar errores y estados, mostrando mensajes claros y evitando confusión.

Otras operaciones asíncronas en AgroRegistro: Además del carrito, otros módulos que utilizan AJAX son:

- **Panel de administración de ofertas:** Creación, edición y eliminación de ofertas mediante peticiones POST, PUT y DELETE.
- **Autenticación y validación de sesiones:** Renovación y manejo del token JWT para mantener la sesión activa.
- **Notificaciones en tiempo real** (en desarrollo): Se planea incorporar WebSockets o polling para actualizaciones en vivo sin recarga.

2.6. Hosting y Despliegue

Para el despliegue del proyecto **AgroRegistro** se utilizaron dos plataformas especializadas que permiten una puesta en producción rápida y eficiente tanto para el backend como para el frontend.

Backend en Railway: El backend fue desplegado en *Railway*, una plataforma que facilita el despliegue de aplicaciones y servicios en la nube con soporte para variables de entorno, bases de datos y escalabilidad automática. Railway ofrece una consola intuitiva para administrar los recursos, monitorear logs y controlar las versiones desplegadas.

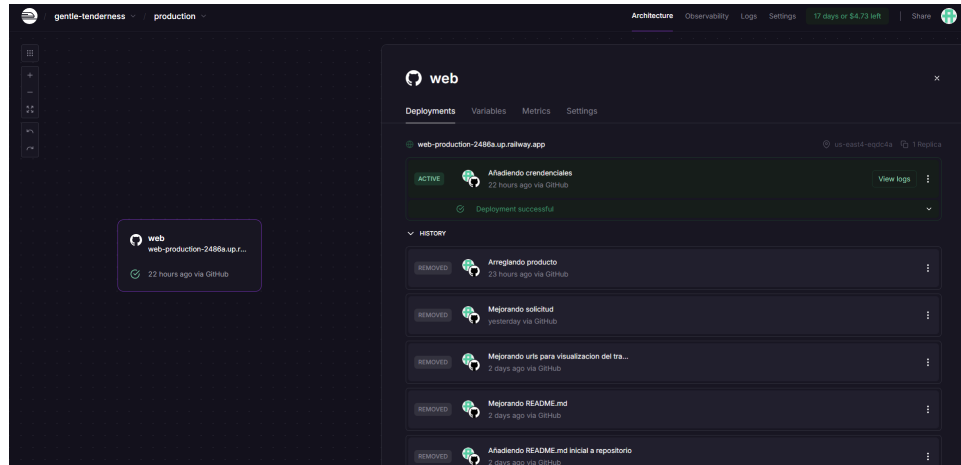


Figura 26: Consola de Railway mostrando el estado del proyecto backend y configuración general.

Para la configuración se establecieron variables de entorno importantes, tales como las credenciales de conexión a la base de datos, claves secretas para la generación de tokens JWT, y otros parámetros necesarios para el correcto funcionamiento de la API.

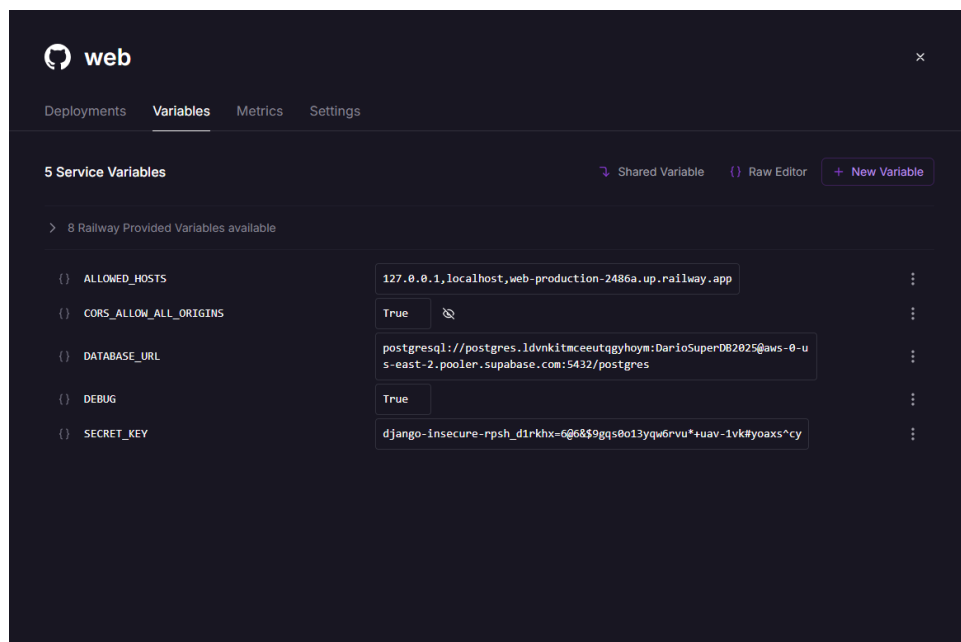


Figura 27: Variables de entorno configuradas en Railway para el backend.

Frontend en Vercel: El frontend fue desplegado en *Vercel*, una plataforma especializada en hosting para aplicaciones React y frameworks modernos. Vercel permite integración continua, donde cada cambio en el repositorio es automáticamente desplegado, asegurando que la versión en producción esté siempre actualizada.

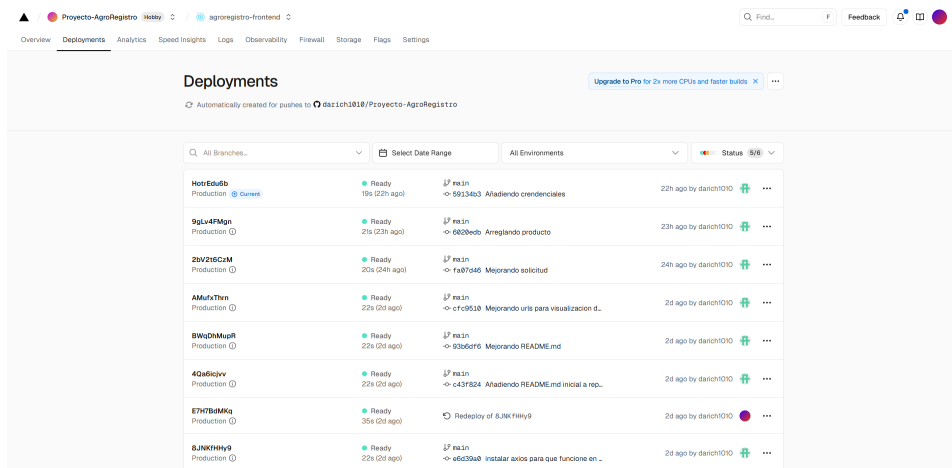


Figura 28: Panel de control de Vercel mostrando el proyecto frontend y el historial de despliegues.

URLs de despliegue:

- Backend API: <https://web-production-2486a.up.railway.app/>
- Frontend: <https://agregistro-frontend.vercel.app/>

Retos y consideraciones durante el despliegue: Conectar y desplegar ambos componentes de **AgroRegistro** no estuvo exento de desafíos. En particular, la integración con la base de datos *Supabase* presentó problemas iniciales, ya que Railway no reconocía adecuadamente la conexión, lo que provocó errores en tiempo de ejecución y dificultades para la autenticación y manejo de datos.

Fue necesario realizar configuraciones adicionales en las variables de entorno, ajustar los parámetros de red y permisos, y modificar el código para asegurar la compatibilidad. Además, el manejo de tokens JWT y la sincronización entre frontend y backend requerían pruebas exhaustivas para garantizar la seguridad y estabilidad del sistema.

A pesar de estos retos, la utilización de Railway y Vercel permitió un flujo de trabajo eficiente con despliegues automáticos y monitoreo en tiempo real, facilitando el mantenimiento y actualización continua del proyecto.

Esta experiencia resalta la importancia de entender a profundidad las plataformas de despliegue y la interoperabilidad con servicios externos, así como la necesidad de pruebas integrales para evitar problemas en producción.

2.7. Otras Consideraciones Técnicas

Uso de Gunicorn y archivo Procfile: Para el despliegue del backend en Railway, se utilizó **Gunicorn** (Green Unicorn), un servidor HTTP WSGI que es ampliamente usado para servir aplicaciones Python en producción, especialmente con frameworks como Django. Gunicorn permite manejar múltiples conexiones concurrentes de manera eficiente, proporcionando un entorno robusto y escalable para la API REST.

Es indispensable para plataformas como Railway especificar cómo iniciar la aplicación mediante un archivo llamado **Procfile** (sin extensión). En este archivo se indica el comando para lanzar Gunicorn apuntando al módulo WSGI del proyecto Django, garantizando que el servidor web pueda gestionar correctamente las solicitudes HTTP y mantener la aplicación en ejecución. Esta configuración es clave para que Railway pueda automatizar el inicio del servidor, controlar procesos y permitir escalabilidad bajo demanda.

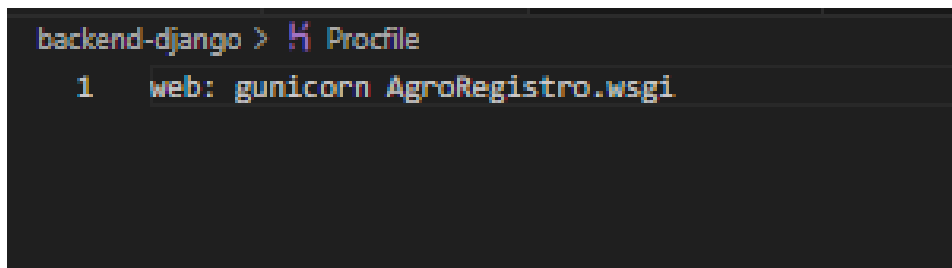


Figura 29: Contenido del archivo **Procfile** utilizado para inicializar Gunicorn en Railway.

Configuración de CORS y uso de HTTPS: Para que el frontend desplegado en Vercel pueda comunicarse de manera segura y sin restricciones con el backend en Railway, fue necesario configurar adecuadamente CORS (Cross-Origin Resource Sharing) en el backend. Esta configuración se realiza en el archivo `settings.py` de Django y permite definir qué orígenes externos pueden acceder a los recursos de la API, evitando errores de seguridad típicos en navegadores modernos.

Respecto al uso de HTTPS, aunque no se verificó directamente la configuración manual, tanto Railway como Vercel proporcionan certificados SSL automáticos para sus dominios, garantizando que la comunicación entre frontend y backend se realice mediante canales cifrados y seguros. Esto es fundamental para proteger la integridad y privacidad de los datos transmitidos, especialmente en aplicaciones que manejan información sensible.

```
110 CORS_ALLOW_CREDENTIALS = True # 🐼 Permite cookies/tokens si es necesario
111
112 CORS_ALLOW_HEADERS = list(default_headers) + [
113     'authorization',
114     'content-type',
115     'access-control-allow-origin',
116 ]
117
118 CORS_EXPOSE_HEADERS = ['Content-Disposition'] # 🐼 Útil para descargas, opcional
119
120 # 🛡️ CSRF desde el frontend de Vercel
121 CSRF_TRUSTED_ORIGINS = [
122     'https://agrorregistro-frontend.vercel.app',
123     'https://web-production-2486a.up.railway.app',
124 ]
125
126 # Logging
127 LOGGING = {
128     'version': 1,
129     'handlers': {
130         'console': {
131             'class': 'logging.StreamHandler',
132             'stream': sys.stdout,
133         },
134     },
135     'loggers': {
136         'django': {
137             'handlers': ['console'],
138             'level': 'DEBUG',
139             'propagate': True,
140         },
141     },
142 }
143
144 # ✂️ Métodos permitidos explícitamente
145 CORS_ALLOW_METHODS = [
146     "DELETE",
147     "GET",
148     "OPTIONS",
149     "PATCH",
150     "POST",
151     "PUT",
152 ]
153
154
155 # dummy change to force redeploy
156
```

Figura 30: Fragmento de la configuración de CORS en `settings.py` para permitir el acceso desde el dominio de Vercel.

Manejo de logs para control de errores: El registro de eventos y errores (logging) es una práctica esencial en el desarrollo de software para monitorear el comportamiento de la aplicación y detectar posibles fallos. En este proyecto se implementó un sistema de logs tanto en el backend como en el frontend.

Por ejemplo, en el archivo `NotificacionesAgricultorList.js` del frontend, se utilizan mensajes de log para rastrear la carga de datos y detectar errores en la gestión de notificaciones para los agricultores. Esto facilita el diagnóstico de problemas y mejora la mantenibilidad del código, permitiendo una rápida respuesta ante cualquier inconveniente en producción.

```

8
9  useEffect(() => {
10    const fetchNotificaciones = async () => {
11      try {
12        const response = await axios.get(`${import.meta.env.VITE_API_URL}/notificaciones/`, {
13          headers: {
14            Authorization: `Bearer ${token}`,
15          },
16        });
17        setNotificaciones(response.data);
18        console.log("📢 Notificaciones recibidas:", response.data);
19      } catch (error) {
20        console.error("❌ Error al obtener notificaciones:", error);
21      }
22    };
23  });

```

Figura 31: Ejemplo de uso de logs en `NotificacionesAgricultorList.js` para el control y monitoreo de errores en el frontend.

3. Recomendaciones

En esta sección se presentan una serie de recomendaciones y buenas prácticas derivadas del desarrollo del proyecto *AgroRegistro*, enfocadas en aspectos de seguridad, funcionalidad, experiencia de usuario y escalabilidad futura. Estas consideraciones buscan no solo mejorar la calidad del sistema actual, sino también sentar las bases para su evolución y mantenimiento a largo plazo.

3.1. Buenas prácticas y seguridad

- **Manejo seguro de tokens y autenticación:** Se recomienda asegurar el almacenamiento de tokens JWT en el frontend, evitando vulnerabilidades como el acceso por scripts maliciosos mediante almacenamiento en `httpOnly` cookies o técnicas similares. Además, validar la expiración y renovación de tokens es fundamental para evitar accesos no autorizados.
- **Control de acceso y roles:** Implementar un sistema robusto de roles y permisos, tanto en backend como frontend, garantizando que cada usuario solo pueda acceder y modificar los recursos que le corresponden. Esto ayuda a prevenir manipulaciones indebidas de datos.
- **Configuración CORS estricta:** Ajustar la configuración de CORS para permitir únicamente los orígenes confiables, limitando el riesgo de ataques Cross-Site Request Forgery (CSRF) o acceso desde sitios externos no autorizados.
- **Manejo de errores y registros:** Utilizar sistemas de logging centralizados para monitorear errores y comportamientos anómalos, facilitando la detección temprana de fallos o intentos de intrusión. En el proyecto se usaron logs externos para mejorar esta trazabilidad.

3.2. Mejoras funcionales y experiencia de usuario

- **Visualización del carrito en tiempo real:** Actualmente, aunque el sistema permite agregar productos al carrito, no se refleja visualmente de forma dinámica para el usuario. Se recomienda implementar una actualización en tiempo real del estado del carrito con notificaciones visuales, mejorando la interacción y claridad del proceso de compra.
- **Funcionalidad completa de solicitudes de pedidos:** Si bien el panel de agricultor permite responder solicitudes, la creación de dichas solicitudes por parte del cliente no está implementada. Este flujo debe ser desarrollado para cerrar el ciclo de interacción y completar la funcionalidad principal del sistema.
- **Mejora en el diseño y estilos:** Se sugiere incorporar librerías de diseño como *Bootstrap* o *Material-UI* para mejorar la apariencia, consistencia y usabilidad de la interfaz, facilitando también la adaptación responsiva en distintos dispositivos.

3.3. Consejos para escalabilidad y funcionalidades futuras

- **Implementación de métodos de pago:** Para convertir la plataforma en una aplicación completamente funcional y comercial, es recomendable integrar pasarelas de pago seguras, como Stripe o PayPal, que permitan realizar transacciones dentro del sistema.
- **Segmentación por grupos o intereses:** Añadir funcionalidades que permitan agrupar usuarios o productos según intereses o categorías facilitará la personalización de la experiencia y la gestión eficiente del catálogo y las ofertas.
- **Optimización del backend y base de datos:** Considerar la adopción de técnicas de cacheo, paginación de datos y optimización de consultas para mejorar el rendimiento ante un volumen creciente de usuarios y transacciones.

- **Pruebas automatizadas y CI/CD:** Implementar pruebas unitarias y de integración, así como pipelines de integración continua y despliegue continuo (CI/CD), para garantizar la calidad y estabilidad del sistema ante futuros cambios.

En resumen, las recomendaciones aquí expuestas buscan fortalecer la arquitectura, seguridad y experiencia del proyecto *AgroRegistro*, preparando el sistema para su evolución hacia una plataforma más robusta, segura y amigable para sus usuarios.

4. Conclusiones

El desarrollo del proyecto **AgroRegistro** ha representado un valioso ejercicio de integración y aplicación de diversas tecnologías modernas en el ámbito del desarrollo web. A lo largo del proceso, se logró construir un sistema funcional que conecta eficientemente un backend robusto, basado en Django REST Framework, con un frontend interactivo desarrollado en React, asegurando una experiencia de usuario fluida y responsive.

Uno de los principales logros fue la correcta implementación de la comunicación asincrónica entre ambas capas mediante AJAX, lo que permitió un manejo dinámico de los datos en tiempo real, fundamental para funcionalidades como la gestión del carrito y la interacción con las ofertas de los agricultores. Asimismo, la integración y despliegue en plataformas en la nube como Railway y Vercel facilitó la puesta en marcha de la aplicación, garantizando accesibilidad y escalabilidad.

Sin embargo, el proyecto también presentó desafíos significativos, especialmente en la conexión y gestión de la base de datos Supabase, cuya compatibilidad requirió soluciones específicas y ajustes en la configuración para asegurar la estabilidad del sistema. Además, la seguridad en la autenticación y manejo de tokens JWT demandó atención cuidadosa para proteger la integridad y privacidad de los usuarios.

Es importante reconocer que, aunque se lograron los objetivos principales, existen áreas de mejora para futuros desarrollos. La implementación de funcionalidades críticas como la visualización dinámica de adiciones al carrito o la capacidad de realizar solicitudes de pedidos desde el panel del cliente están pendientes y son fundamentales para una experiencia completa. De igual modo, el estilizado de la interfaz podría beneficiarse del uso de frameworks CSS como Bootstrap para mejorar la usabilidad y diseño visual.

Finalmente, para que **AgroRegistro** alcance un nivel profesional y escalable, se recomienda incorporar métodos de pago seguros, implementar sistemas de agrupación por intereses y fortalecer la gestión de notificaciones y comunicación interna.

En conclusión, este proyecto ha sido un paso significativo hacia la construcción de soluciones digitales integrales para la gestión agrícola, demostrando la importancia de la planificación, el trabajo interdisciplinario y la adaptabilidad tecnológica para enfrentar los retos actuales.

Evaluación del Trabajo en Equipo

En esta sección se presentan las autoevaluaciones y evaluaciones cruzadas correspondientes a los miembros del equipo de desarrollo.

Rúbrica sugerida por el profesor para la calificación

	4 - Excelente	3 - Bueno	2 - Regular	0.5 - eficiente	Autocalificación Estudiante	Calificación Profesor
Producción de videos	Muy claro, coherente y bien estructurado. Alta calidad (audio, video, edición). Comunica totalmente el propósito del video.	Claro en general. Buena calidad con detalles menores. Comunica la mayoría.	Poco claro o con errores leves. Calidad media, edición básica. Comunica parcialmente.	Confuso o con errores graves. Baja calidad o sin edición. No comunica el objetivo.	3	
Backend en Producción	100% funcional y sin errores. Desplegado correctamente y accesible. Seguridad y validación sólidas.	Funciona con pequeños errores. Desplegado con leves fallos. Buen manejo general.	Funciona parcialmente. Despliegue incompleto o poco estable. Validaciones básicas.	No funciona. No desplegado. Sin validaciones ni manejo de errores.	3	
Frontend en Producción	Interfaz atractiva, intuitiva y responsiva. Todas las funcionalidades operativas. Totalmente integrado y funcional.	Buena usabilidad con pocos detalles. Algunas fallas menores. Integración parcial, funciona bien.	Interfaz básica y poco intuitiva. Funciones básicas, faltan otras. Integración básica o incompleta.	Interfaz confusa o sin diseño. Fallos importantes o no funcional. No hay integración.	2	
Backup de BD	Backup completo y verificado. Documentado claramente. Proceso automatizado y funcional.	Backup completo sin verificación. Documentado con pocos detalles. Proceso semiautomatizado.	Backup incompleto. Documentación mínima. Manual pero funcional.	No se generó backup. Sin documentación. No funcional.	2	
Github Backend	Muy clara y profesional. Commits frecuentes, claros y con mensajes descriptivos. Uso correcto de ramas y merges.	Bien organizada. Mensajes adecuados. Uso parcial de ramas.	Algo desorganizada. Pocos commits o mensajes vagos. Uso mínimo de ramas.	Confusa o caótica. Sin estructura ni control. Todo en main/master sin control.	3	
Github Frontend	Muy bien estructurado. Detallado y constante. Buen uso de versiones y ramas.	Estructura adecuada. Constante pero sin detalle. Uso limitado de control.	Organización básica. Escaso o mal documentado. Versión única y cambios directos.	Desorganizado. Desordenado o ausente. Sin control de versiones.	3	
Informe Final	Muy completo, análisis crítico y detallado. Excelente redacción, sin errores. Incluye capturas, enlaces, resultados, etc. Se autocalifica.	Completo con buen análisis. Pocos errores menores. Incluye algunas evidencias.	Superficial pero cumple. Algunos errores importantes. Pocas evidencias.	Incompleto o sin análisis. Muchos errores. No incluye evidencias.	3	
				TOTAL	19	

Autoevaluación escrita

Darío Cornejo Hurtado: **Responsabilidad:** Cumplí con todas las tareas asignadas, tanto en el backend como en la conexión con el frontend. Subí a tiempo el código a GitHub y me aseguré de que la base de datos estuviera poblada correctamente.

Proactividad: Implementé características adicionales no pedidas como el sistema de notificaciones al agricultor y estructura modular de modelos.

Aporte al grupo: Coordiné el despliegue en Railway y Vercel, asistí con bugs en el frontend, orienté a mi compañero sobre la lógica de roles y ayudé a estructurar el informe.

Evaluación de compañeros: Realicé la calificación completa a mis compañeros según lo requerido.

Nota individual considerada: 20/20, Nota a mi compañero considerada: 20/20

Diego Cervantes Apaza: **Responsabilidad:** Cumplí con todas las tareas asignadas, principalmente en el frontend y me aseguré de que la base de datos estuviera correctamente implementada.

Proactividad: Implementé manejo inteligente del carrito con lógica de stock.

Aporte al grupo: Me encargué de la conexión correcta con Supabase y asistí con bugs en el frontend y ayudé a estructurar el informe.

Evaluación de compañeros: Realicé la calificación completa a mis compañeros según lo requerido.

Nota individual considerada: 20/20, Nota a mi compañero considerada: 20/20

5. Anexos

5.1. Respaldo de Base de Datos

5.10a. ¿Por qué realizar un respaldo?: Realizar un respaldo (backup) de la base de datos es una práctica fundamental para cualquier proyecto que maneje información importante. El respaldo permite proteger los datos ante posibles fallos, pérdidas accidentales, errores humanos o problemas técnicos. Contar con copias actualizadas de la base de datos asegura que el proyecto pueda recuperarse rápidamente y continuar funcionando sin pérdida significativa de información.

5.10b. Método y procedimiento utilizado: Para el proyecto **AgroRegistro**, se decidió realizar el respaldo exportando cada tabla de la base de datos en formato CSV. Este método es sencillo, portátil y compatible con múltiples herramientas y sistemas, facilitando la restauración o análisis posterior.

El procedimiento realizado fue el siguiente:

1. Se ingresó a la plataforma de administración de la base de datos (Supabase).

2. Para cada tabla principal, se utilizó la opción de exportar datos en formato Excel.
3. Los archivos Excel se convirtieron a formato CSV para cumplir con los requisitos del curso.
4. Se almacenaron todos los archivos CSV en una carpeta organizada llamada **BACKUPS CSV**.
5. Finalmente, estos archivos se subieron al repositorio del proyecto para mantenerlos accesibles y versionados.

Nombre del archivo	Contenido principal
api_agricultor.csv	Información personal y de contacto de los agricultores registrados.
api_carritoitem.csv	Detalles de los productos agregados al carrito de cada cliente.
api_categoria.csv	Categorías bajo las cuales se clasifican los productos.
api_cliente.csv	Datos personales y de contacto de los clientes registrados.
api_notificacionagricultor.csv	Notificaciones enviadas a los agricultores sobre solicitudes y mensajes.
api_oferta.csv	Ofertas creadas por los agricultores con precios y detalles de productos.
api_producto.csv	Detalles de los productos disponibles en la plataforma.
api_respuestasolicitud.csv	Respuestas proporcionadas por los agricultores a solicitudes de clientes.
api_solicitudproducto.csv	Solicitudes realizadas por los clientes para productos específicos.
api_usuario.csv	Información general de los usuarios del sistema (roles, credenciales, etc.).
auth_permission.csv	Permisos definidos para los roles de usuarios.
django_content_type.csv	Información técnica interna del framework Django para los modelos.

Cuadro I: Archivos CSV de respaldo y su contenido principal.

5.10c. *Archivos de respaldo y su contenido principal:*

5.10d. *Ubicación de los archivos:* Todos los archivos de respaldo están organizados y almacenados en la carpeta llamada **BACKUPS CSV** dentro del repositorio principal del proyecto. Esta estructura facilita su acceso y mantenimiento, asegurando que se pueda localizar rápidamente en caso de ser necesario restaurar o analizar la base de datos.

Se recomienda realizar respaldos periódicos y mantener actualizados estos archivos para preservar la integridad y disponibilidad de la información del proyecto.

Referencias

1. Django Software Foundation. (2024). *Django Documentation*. Disponible en: <https://docs.djangoproject.com/en/4.2/>
2. Django REST framework. (2024). *DRF Documentation*. Disponible en: <https://www.django-rest-framework.org/>
3. Supabase Inc. (2024). *Supabase Docs*. Disponible en: <https://supabase.com/docs>
4. Vercel Inc. (2024). *Vercel Deployment Platform*. Disponible en: <https://vercel.com/>
5. Railway. (2024). *Railway Docs - Deploying backend apps*. Disponible en: <https://docs.railway.app/>
6. React – Meta Platforms Inc. (2024). *React Documentation*. Disponible en: <https://react.dev/>
7. Mozilla. (2024). *MDN Web Docs*. Disponible en: <https://developer.mozilla.org/>
8. GitHub Inc. (2024). *GitHub Documentation*. Disponible en: <https://docs.github.com/>
9. PostgreSQL Global Development Group. (2024). *PostgreSQL Documentation*. Disponible en: <https://www.postgresql.org/docs/>
10. Stack Overflow. (2024). *Stack Overflow - Community Discussions*. Disponible en: <https://stackoverflow.com/>