**Prueba Técnica PHP**

# INSTRUCCIONES GENERALES

Deberás desarrollar una aplicación en PHP sin utilizar frameworks, con la única excepción de Doctrine para la gestión de base de datos.

El objetivo de esta prueba es evaluar tus habilidades en modelado de dominio, diseño limpio, persistencia de datos y pruebas automatizadas.

Tu solución debe incluir:

* Entidades correctamente diseñadas.
* Uso de Value Objects para asegurar consistencia en datos clave.
* Repositorios implementados a través de interfaces.
* Un caso de uso separado del controlador.
* Pruebas unitarias y de integración utilizando PHPUnit.
* Persistencia de datos en MySQL utilizando Doctrine.
* Despliegue del entorno con Docker.
* Implementación de Domain-Driven Design (DDD) para estructurar el dominio de la aplicación.
* Aplicación del patrón Ports and Adapters, asegurando que la lógica de negocio esté desacoplada de la infraestructura.

Las pruebas que no cumplan estos requisitos serán descartadas.

# REQUISITOS TÉCNICOS

1. Modelado de Dominio

* Implementa la entidad `User` con atributos clave como:
* `id` (debe ser un Value Object que represente una identidad única).
* `name` (debe ser un Value Object que valide longitud mínima y caracteres permitidos).
* `email` (debe ser un Value Object con validación de formato).
* `password` (debe ser un Value Object con gestión de hash y validación).
* `createdAt` (fecha de creación automática).
* La entidad debe ser inmutable en lo posible.

2. Repositorio de Usuarios

* Implementa un repositorio `UserRepositoryInterface` con los siguientes métodos:
* `save(User $user): void`
* `findById(UserId $id): ?User`
* `delete(UserId $id): void`
* Implementa una clase `DoctrineUserRepository` para manejar la persistencia de datos en MySQL con Doctrine.

3. Caso de Uso: Registro de Usuario

* Crea un caso de uso (`RegisterUserUseCase`) que permita registrar un nuevo usuario.
* El caso de uso debe recibir los datos a través de un DTO (`RegisterUserRequest`).
* Validar que el email no esté en uso antes de registrar al usuario.
* Si el usuario se registra correctamente, debe dispararse un evento de dominio `UserRegisteredEvent`.

4. Eventos de Dominio

* Al registrar un usuario, se debe disparar un evento `UserRegisteredEvent`.
* Implementa un manejador de eventos que escuche `UserRegisteredEvent` y ejecute una acción simulada, como enviar un email de bienvenida.

5. Controlador y Formato de Respuesta

* Implementa un controlador (`RegisterUserController`) que reciba una solicitud HTTP y llame al caso de uso.
* La respuesta del caso de uso debe entregarse en formato JSON utilizando un DTO (`UserResponseDTO`).

6. Seguridad y Validaciones

* La contraseña debe cumplir con una política estricta:
* Mínimo 8 caracteres.
* Al menos una letra mayúscula, un número y un carácter especial.
* Si se recibe un email inválido, debe lanzarse una `InvalidEmailException`.
* Si la contraseña no cumple los requisitos, `WeakPasswordException`.
* Si se intenta registrar un usuario con un email ya existente, `UserAlreadyExistsException`.

7. Pruebas Automatizadas

* Implementa pruebas unitarias para:
* La entidad `User`.
* Los Value Objects (`UserId`, `Email`, `Name`, `Password`).
* El caso de uso `RegisterUserUseCase`.
* Implementa una prueba de integración para verificar que el repositorio funciona correctamente con Doctrine y MySQL.

8. Configuración con Docker

* Todo el sistema debe ser desplegable mediante Docker.
* Crea un `docker-compose.yml` que incluya:
* Un servicio para PHP con Doctrine y Composer instalados.
* Un servicio para la base de datos MySQL.
* Proporciona un `Makefile` o script para inicializar el entorno rápidamente.

# CRITERIOS DE EVALUACIÓN

Se valorará especialmente:

* Uso de Value Objects para encapsular datos sensibles.
* Implementación de repositorios con interfaces.
* Diseño limpio siguiendo principios de DDD y Clean Architecture.
* Uso adecuado de tests unitarios y de integración con PHPUnit.
* Persistencia de datos correctamente implementada con Doctrine y MySQL.
* Controladores desacoplados que actúan como adaptadores.
* Configuración de Docker funcional y correctamente documentada.

# ENTREGA

1. Sube tu código a un repositorio público en GitHub.
2. Incluye un archivo README.md explicando cómo ejecutar el proyecto.
3. Comparte el enlace con nosotros.

**Explicación del Makefile:**

* **Variables:**
  + COMPOSE\_FILE: Especifica el nombre de tu archivo docker-compose.yml.
  + CONTAINER\_NAME: Especifica el nombre del contenedor PHP (ajusta si es necesario).
* **Comandos:**
  + build: Construye las imágenes Docker.
  + up: Inicia los contenedores en modo "detached" (en segundo plano).
  + down: Detiene y elimina los contenedores.
  + exec: Abre una sesión de bash dentro del contenedor PHP.
  + logs: Muestra los registros del contenedor PHP y los sigue en tiempo real.
  + rebuild: Detiene, reconstruye e inicia los contenedores.
  + restart: Reinicia el contenedor PHP.
  + clean: Detiene los contenedores y limpia las imágenes, contenedores y volúmenes no utilizados.
  + run: Ejecuta el script run.sh.
* .PHONY: Declara los objetivos como "phony" (no representan archivos reales).

**Explicación del run.sh:**

* Ejecuta comandos dentro del contenedor php-service.
* En el ejemplo, ejecuta composer install y php artisan migrate (si usas Laravel).
* Puedes agregar otros comandos que necesites ejecutar al iniciar el entorno.

**Cómo Usar:**

1. Guarda el Makefile y el run.sh en el mismo directorio que tu docker-compose.yml.
2. Abre una terminal en ese directorio.
3. Ejecuta los comandos make <objetivo>:
   * make build: Para construir las imágenes.
   * make up: Para iniciar los contenedores.
   * make down: Para detener los contenedores.
   * make exec: Para acceder al contenedor.
   * make logs: Para ver los registros.
   * make rebuild: Para reconstruir el entorno.
   * make restart: Para reiniciar el contenedor.
   * make clean: Para limpiar el entorno.
   * make run: Para ejecutar el script run.sh