**🔹 Guía Paso a Paso con Prompts para Construir el Backend de Zentro**

**Fase 1: Los Cimientos (Configuración y Base de Datos)**

📌 Estos pasos se hacen una sola vez y son la base de todo.

**Paso 1.1 – requirements.txt**

**Prompt:**

Genera el archivo requirements.txt para un backend en **FastAPI** con PostgreSQL y SQLAlchemy, usando autenticación JWT y Pydantic. Asegúrate de incluir librerías comunes como fastapi, uvicorn, sqlalchemy, alembic, python-jose, passlib[bcrypt], pydantic[email], python-dotenv, y cualquier otra necesaria para que el proyecto funcione correctamente. Optimiza el archivo para producción y desarrollo.

**Paso 1.2 – core/config.py**

**Prompt:**

Genera el archivo core/config.py para un proyecto FastAPI.

* Usa pydantic.BaseSettings para cargar variables de entorno.
* Variables mínimas: DATABASE\_URL, SECRET\_KEY, ALGORITHM, ACCESS\_TOKEN\_EXPIRE\_MINUTES.
* Asegúrate de que sea extensible para añadir más configuraciones después.
* Usa buenas prácticas de tipado y comentarios.

**Paso 1.3 – db/session.py**

**Prompt:**

Genera el archivo db/session.py.

* Configura la conexión con **SQLAlchemy** usando la DATABASE\_URL desde config.py.
* Crea un engine y un SessionLocal.
* Implementa la función get\_db() como dependencia de FastAPI, usando yield.
* Usa buenas prácticas para manejo de sesiones y errores.

**Paso 1.4 – db/base\_class.py**

**Prompt:**

Genera el archivo db/base\_class.py.

* Define la clase base Base usando declarative\_base() de SQLAlchemy.
* Incluye una columna id genérica como clave primaria Integer, autoincrement, primary\_key=True.
* Prepara este archivo para que todos los modelos lo hereden.

**Paso 1.5 – core/security.py**

**Prompt:**

Genera el archivo core/security.py.

* Implementa funciones para:
  1. Hash de contraseñas (get\_password\_hash).
  2. Verificación de contraseñas (verify\_password).
  3. Creación de tokens JWT (create\_access\_token).
* Usa passlib[bcrypt] y python-jose.
* Obtén la configuración (SECRET\_KEY, ALGORITHM, ACCESS\_TOKEN\_EXPIRE\_MINUTES) desde config.py.
* Código seguro, limpio y bien comentado.

**Paso 1.6 – core/dependencies.py**

**Prompt:**

Genera el archivo core/dependencies.py.

* Crea la dependencia get\_db() que devuelve una sesión de base de datos.
* Implementa una función get\_current\_user() que:
  + Obtenga el token JWT desde el encabezado Authorization.
  + Lo valide con core/security.py.
  + Busque el usuario en la base de datos.
  + Retorne un objeto User autenticado.
* Asegúrate de manejar errores con HTTPException.

**Fase 2: Usuarios, Roles y Permisos (Corazón del Backend)**

📌 Este módulo es clave. Una vez dominado, se repite el patrón en los demás.

**Paso 2.1 – models/user.py**

**Prompt:**

Genera el modelo SQLAlchemy User en models/user.py.

* Hereda de Base.
* Campos mínimos: id, email, username, hashed\_password, is\_active, role\_id.
* Relación con Role.
* Usa tipado y buenas prácticas.

**Paso 2.2 – models/role.py**

**Prompt:**

Genera el modelo SQLAlchemy Role en models/role.py.

* Campos: id, name, description.
* Relación con User y Permission (muchos a muchos).
* Define correctamente la tabla intermedia role\_permissions.

**Paso 2.3 – models/permission.py**

**Prompt:**

Genera el modelo SQLAlchemy Permission en models/permission.py.

* Campos: id, name, description.
* Relación muchos a muchos con Role.
* Usa la tabla intermedia role\_permissions.

**Paso 2.4 – schemas/user\_schema.py**

**Prompt:**

Genera schemas/user\_schema.py con modelos Pydantic.

* UserBase (email, username).
* UserCreate (hereda de UserBase, añade password).
* UserRead (id, email, username, role).
* UserUpdate (parcial).

**Paso 2.5 – schemas/role\_schema.py**

**Prompt:**

Genera schemas/role\_schema.py.

* RoleBase (name, description).
* RoleCreate.
* RoleRead (incluye lista de permisos).
* RoleUpdate.

**Paso 2.6 – schemas/permission\_schema.py**

**Prompt:**

Genera schemas/permission\_schema.py.

* PermissionBase (name, description).
* PermissionCreate.
* PermissionRead.
* PermissionUpdate.

**Paso 2.7 – schemas/auth\_schema.py**

**Prompt:**

Genera schemas/auth\_schema.py.

* Modelo para LoginRequest (username/email + password).
* Modelo para TokenResponse (access\_token, token\_type).
* Usa validaciones de email.

**Paso 2.8 – dao/user\_dao.py**

**Prompt:**

Genera dao/user\_dao.py.

* CRUD básico para User.
* Métodos: get\_by\_id, get\_by\_email, create, update, delete.
* Manejo de sesión SQLAlchemy.

**Paso 2.9 – dao/role\_dao.py**

**Prompt:**

Genera dao/role\_dao.py.

* CRUD para Role.
* Incluye manejo de permisos asociados.

**Paso 2.10 – dao/permission\_dao.py**

**Prompt:**

Genera dao/permission\_dao.py.

* CRUD básico para Permission.

**Paso 2.11 – services/user\_service.py**

**Prompt:**

Genera services/user\_service.py.

* Funciones: create\_user, authenticate\_user, get\_user, update\_user.
* Usa user\_dao.
* En create\_user encripta contraseña.
* En authenticate\_user valida credenciales y devuelve usuario.

**Paso 2.12 – services/role\_service.py**

**Prompt:**

Genera services/role\_service.py.

* Funciones: create\_role, assign\_permissions, get\_roles.
* Usa role\_dao.

**Paso 2.13 – services/permission\_service.py**

**Prompt:**

Genera services/permission\_service.py.

* Funciones CRUD para permisos.
* Usa permission\_dao.

**Paso 2.14 – api/v1/auth.py**

**Prompt:**

Genera el archivo api/v1/auth.py.

* Endpoint POST /login que use user\_service.authenticate\_user.
* Devuelve TokenResponse.
* Usa seguridad con JWT.

**Paso 2.15 – api/v1/users.py**

**Prompt:**

Genera el archivo api/v1/users.py.

* CRUD de usuarios.
* Protege rutas con get\_current\_user.
* Usa user\_service.

**Paso 2.16 – api/v1/roles.py**

**Prompt:**

Genera api/v1/roles.py.

* Endpoints CRUD para roles.
* Endpoint para asignar permisos a un rol.
* Usa role\_service.

**Paso 2.17 – api/v1/permissions.py**

**Prompt:**

Genera api/v1/permissions.py.

* CRUD de permisos.
* Usa permission\_service.

**Paso 2.18 – db/init\_db.py**

**Prompt:**

Genera db/init\_db.py.

* Inserta roles básicos (admin, trainer, client).
* Inserta permisos básicos (manage\_users, manage\_classes, etc.).
* Crea un usuario admin por defecto.

**Paso 2.19 – api/v1/api.py**

**Prompt:**

Genera api/v1/api.py.

* Router principal de la API.
* Incluye routers de auth, users, roles, permissions.

**Paso 2.20 – main.py**

**Prompt:**

Genera main.py.

* Inicializa FastAPI.
* Incluye api\_router.
* Configura CORS.
* Expón /docs y /redoc.

**Fase 3 — Módulos funcionales (prompts listos para copiar/pegar)**

Nota: sustituye <modulo> por el nombre (ej. clients, trainers) y ajusta nombres de campos cuando sea necesario.

**Prompts generales reutilizables (utils y patrones comunes)**

**Prompt: util/pagination.py**

Genera app/core/pagination.py.

* Implementa parámetros comunes de paginación (limit, offset, page, page\_size) como Pydantic BaseModel.
* Añade una función utilitaria paginate(query, limit, offset) que reciba un query de SQLAlchemy y devuelva (items, total, limit, offset).
* Incluye ejemplos de uso en comentarios.

**Prompt: util/filters.py**

Genera app/core/filters.py.

* Implementa helpers para aplicar filtros dinámicos (texto, fecha entre, rango numérico) a queries de SQLAlchemy.
* Incluye función apply\_search(query, model, fields, q) y apply\_date\_range(query, column, start, end).

**Prompt: base/crud\_base.py**

Genera app/dao/crud\_base.py.

* Clase genérica CRUDBase(Model, SchemaCreate, SchemaUpdate) con métodos get, get\_multi, create, update, remove.
* Implementar paginación y optional eager loading.
* Documentar cómo heredar para casos especiales.

**Módulo: Clients (clientes)**

**1) models/client.py**

Genera models/client.py.

* Modelo SQLAlchemy Client que herede de Base.
* Campos: id, first\_name, last\_name, email (unique), phone, birthdate, address, joined\_at (timestamp), is\_active, user\_id (FK a User, nullable) y photo\_url.
* Añadir created\_at y updated\_at (timestamps automáticos).
* Indexes en email y user\_id.
* Relaciones: user (si aplica), memberships (relación a Membership), attendances.

**2) schemas/client\_schema.py**

Genera schemas/client\_schema.py.

* ClientBase (first\_name, last\_name, email, phone, birthdate, address).
* ClientCreate (hereda ClientBase y requiere email).
* ClientRead (id, joined\_at, is\_active, photo\_url).
* ClientUpdate (parcial).
* Validaciones (email format, phone pattern, birthdate no futura).

**3) dao/client\_dao.py**

Genera dao/client\_dao.py.

* Usa CRUDBase o implementa métodos: get\_by\_id, get\_by\_email, get\_multi (con paginación y búsqueda por nombre/email), create, update, soft\_delete.
* get\_with\_memberships(client\_id) que haga eager load.

**4) services/client\_service.py**

Genera services/client\_service.py.

* Lógica: creación (verificar email único), actualizar (validaciones), activar/desactivar cuenta, asignar user account si aplica, get\_profile(client\_id) que consolide datos (memberships, last attendance).
* Usar client\_dao.
* Exponer search\_clients(query, page, page\_size) para frontend.

**5) api/v1/clients.py**

Genera api/v1/clients.py.

* Rutas: GET /clients (paginado, filtro por q), GET /clients/{id}, POST /clients, PUT /clients/{id}, DELETE /clients/{id} (soft delete).
* Proteger según permisos: crear/editar sólo para roles con manage\_clients. GET para staff autenticado o dueño.
* Validar y devolver ClientRead.
* Soportar subir photo (endpoint POST /clients/{id}/photo o multipart).

**6) tests (clients)**

Genera tests en tests/test\_clients.py.

* Unit tests para client\_dao y client\_service (crear, buscar, actualizar).
* Integration tests para endpoints (POST /clients, GET /clients).

**7) migration & seed**

Genera Alembic migration para la tabla clients y un script de db/init\_db.py para crear 5 clientes de ejemplo.

**Módulo: Trainers (entrenadores)**

**1) models/trainer.py**

Genera models/trainer.py.

* Campos: id, first\_name, last\_name, email unique, phone, bio, specialties (array/text), certifications (json/text), is\_active, user\_id FK.
* Relaciones: classes (GymClass), routines\_assigned.

**2) schemas/trainer\_schema.py**

Genera schemas/trainer\_schema.py con TrainerBase, TrainerCreate, TrainerRead, TrainerUpdate. Validaciones similares a client.

**3) dao/trainer\_dao.py**

Genera dao/trainer\_dao.py con CRUD, get\_by\_specialty, get\_available\_for(date\_range).

**4) services/trainer\_service.py**

Genera services/trainer\_service.py.

* Lógica: asignar a clases, marcar disponibilidad, validaciones de solapamiento de horarios, obtener perfil público (bio + clases).

**5) api/v1/trainers.py**

Genera api/v1/trainers.py.

* Endpoints CRUD y GET /trainers/{id}/schedule.
* Endpoint POST /trainers/{id}/certificates para subir archivos (pdf/images).

**6) tests + migration**

Tests unitarios e integración; Alembic para tabla trainers.

**Módulo: Memberships (membresías)**

**1) models/membership.py**

Genera models/membership.py.

* Campos: id, client\_id FK, type (enum: monthly, yearly, dropin, classpack), status (active, expired, paused), start\_date, end\_date, price, auto\_renew (bool), remaining\_sessions (nullable).
* Relaciones: client, payments.

**2) schemas/membership\_schema.py**

Genera schemas/membership\_schema.py con MembershipCreate (type, start\_date, end\_date, price, client\_id), MembershipRead, MembershipUpdate.

**3) dao/membership\_dao.py**

Genera dao/membership\_dao.py.

* CRUD, get\_active\_by\_client(client\_id), decrement\_session(client\_id).

**4) services/membership\_service.py**

Genera services/membership\_service.py.

* Reglas: creación con verificación de solapamiento, activación automática, expiración por cron job (cron job se crea en Fase 4), consume\_session(client\_id) con transacción (evitar race conditions).
* Integrar con payments (placeholder).

**5) api/v1/memberships.py**

Genera api/v1/memberships.py.

* Endpoints para CRUD, POST /memberships/{id}/pause, POST /memberships/{id}/resume, GET /clients/{id}/memberships.

**6) tests + migration**

Tests de negocio (consumo de sesión, expiración). Alembic migration.

**Módulo: Classes (clases/gimnasio)**

**1) models/gym\_class.py**

Genera models/gym\_class.py.

* Campos: id, title, description, capacity, start\_time (datetime), end\_time, trainer\_id FK, room (string), status (scheduled, cancelled, completed).
* Relaciones: trainer, attendances (many-to-many con clients través attendance table), memberships\_allowed (opcional).

**2) schemas/class\_schema.py**

Genera schemas/class\_schema.py con ClassCreate, ClassRead, ClassUpdate, ClassList (para calendario).

**3) dao/class\_dao.py**

Genera dao/class\_dao.py.

* CRUD, get\_by\_date\_range, get\_available\_classes, book\_class(client\_id, class\_id) (operación transaccional que verifica cupo y membresía).

**4) services/class\_service.py**

Genera services/class\_service.py.

* Lógica: crear clases, asignar trainer, reservar cupos (book/unbook), enviar notificaciones de cupo lleno, reglas de cancelación/penalización.
* book\_class debe decrementar disponibilidad y crear attendance.

**5) api/v1/classes.py**

Genera api/v1/classes.py.

* Endpoints: GET /classes (filtros por fecha/trainer/type), POST /classes (staff), POST /classes/{id}/book (cliente), POST /classes/{id}/cancel-booking.
* Uso de WebSocket opcional para notificar plazas.

**6) attendance model**

Genera models/attendance.py (tabla intermedia). Campos: id, class\_id, client\_id, status (booked, attended, no\_show), booked\_at, attended\_at.

**7) tests + migration**

Tests transaccionales y Alembic.

**Módulo: Routines (rutinas)**

**1) models/routine.py**

Genera models/routine.py.

* Campos: id, title, description, exercises (JSON list con sets/reps/tempo), difficulty, duration\_minutes, created\_by (trainer\_id), is\_public.
* Relaciones: trainer y assigned\_to\_clients (many-to-many via routine\_assignments).

**2) schemas/routine\_schema.py**

Genera schemas/routine\_schema.py con RoutineCreate, RoutineRead, RoutineAssign (client\_id + start/end).

**3) dao/routine\_dao.py**

Genera dao/routine\_dao.py con CRUD y get\_templates, assign\_routine(client\_id, routine\_id, schedule).

**4) services/routine\_service.py**

Genera services/routine\_service.py.

* Lógica: clonar plantilla, asignar y versionar rutinas, marcar completadas, track de progreso.

**5) api/v1/routines.py**

Genera api/v1/routines.py con endpoints para templates públicos, asignaciones, historial de progreso.

**6) tests + migration**

Tests para asignación y versiones.

**Módulo: Nutrition (nutrición / planes alimenticios)**

**1) models/nutrition.py**

Genera models/nutrition.py.

* Campos: id, title, calories, macros (json: protein/fat/carbs), meals (json list), created\_by (trainer or nutritionist), is\_template.

**2) schemas/nutrition\_schema.py**

Genera schemas/nutrition\_schema.py (create/read/update/assign).

**3) dao/nutrition\_dao.py**

Genera dao/nutrition\_dao.py.

**4) services/nutrition\_service.py**

Genera services/nutrition\_service.py.

* Lógica: crear plan, asignar a cliente, historial, reemplazo automático basado en objetivos.

**5) api/v1/nutrition.py**

Genera api/v1/nutrition.py con endpoints CRUD y POST /nutrition/{id}/assign.

**Módulo: Store (tienda / productos)**

**1) models/product.py**

Genera models/product.py.

* Campos: id, sku, name, description, price, stock, category, images (json/list), is\_active.
* Añadir inventory\_transactions (movimientos stock).

**2) schemas/product\_schema.py**

Genera schemas/product\_schema.py.

**3) dao/store\_dao.py**

Genera dao/store\_dao.py.

* CRUD, decrement\_stock(product\_id, qty) transaccional.

**4) services/store\_service.py**

Genera services/store\_service.py.

* Lógica: venta (crear order placeholder), gestionar stock, notificaciones bajo stock, integrar con gateway de pagos (planificar en Fase 4).

**5) api/v1/store.py**

Genera api/v1/store.py con endpoints público GET /products, POST /orders (placeholder), POST /products/{id}/images.

**6) tests + migration**

Tests y Alembic.

**Módulo: Incidents (incidencias)**

**1) models/incident.py**

Genera models/incident.py.

* Campos: id, title, description, reported\_by (user\_id/client\_id), status (open, in\_progress, resolved), priority, assigned\_to (staff), created\_at, resolved\_at, attachments (json).

**2) schemas/incident\_schema.py**

Genera schemas/incident\_schema.py.

**3) dao/incident\_dao.py**

Genera dao/incident\_dao.py con CRUD y assign\_incident.

**4) services/incident\_service.py**

Genera services/incident\_service.py.

* Lógica: escalado automático por prioridad, SLA checks (notificar si excede tiempo).

**5) api/v1/incidents.py**

Genera api/v1/incidents.py con endpoints para reportar incidencias, listarlas, asignar y cerrar.

**Módulo: Reception (recepción / check-in)**

**1) models/reception.py**

Genera models/reception.py.

* Campos: id, client\_id, checkin\_time, checkout\_time, method (qr, manual), notes, frontdesk\_user\_id.

**2) schemas/reception\_schema.py**

Genera schemas/reception\_schema.py.

**3) dao/reception\_dao.py**

Genera dao/reception\_dao.py con check\_in(client\_id) y check\_out(client\_id).

**4) services/reception\_service.py**

Genera services/reception\_service.py.

* Lógica: validar membresía activa al check-in, penalizaciones por no presentarse a clase (se integra con class\_service).

**5) api/v1/reception.py**

Genera api/v1/reception.py con endpoints POST /reception/checkin y POST /reception/checkout, GET /reception/today.

**Prompts extras por módulo (útiles)**

**Prompt: uploads/file\_store.py**

Genera app/core/file\_store.py que gestione uploads locales (y switch a S3). Debe exponer save\_file(file, dest) y get\_file\_url(path). Documentar seguridad (validar extensiones).

**Prompt: permissions integration**

Genera snippet y ejemplo de uso: en cada api/v1/<modulo>.py añadir Depends(has\_permission('manage\_<module>')). Crea helper core/permissions\_checker.py con has\_permission(permission\_name) que use get\_current\_user().

**Prompt: soft deletes & audit**

Genera mixin app/db/mixins.py con SoftDeleteMixin y AuditMixin (created\_by, updated\_by, deleted\_at). Indica cómo aplicarlo en modelos.

**Pruebas y seeds por módulo (prompt único)**

**Prompt: tests\_and\_seed\_all\_modules**

Genera tests/ skeleton para cada módulo (clients, trainers, memberships, classes, routines, nutrition, store, incidents, reception) con fixtures para DB (use sqlite in-memory), y un script scripts/seed\_dev\_data.py que cree: 3 trainers, 10 clients, 3 memberships types, 8 classes en calendario y 5 products. Documentar cómo ejecutar tests (pytest) y seed.

**Inclusión en router principal (prompt)**

**Prompt: api/v1/api.py (actualizar)**

Actualiza api/v1/api.py para incluir routers: clients, trainers, memberships, classes, routines, nutrition, store, incidents, reception. Asegúrate de montar con prefijado prefix="/api/v1" y tags por módulo.

**Migraciones y Alembic (prompt)**

**Prompt: alembic\_setup\_and\_first\_migration**

Genera instrucciones y archivos base para Alembic: alembic.ini, env.py configurado para detectar app.db.base\_class.Base.metadata. Crea la primera migration que incluye todas las tablas de la Fase 3. Añade ejemplo de comando para generar migrations automáticamente: alembic revision --autogenerate -m "Initial modules".

**Entregable final de la Fase 3**

He incluido **todos** los prompts esenciales para cada archivo del árbol de módulos (models, schemas, dao, services, api, tests, migrations, seeds y utilidades comunes). Si quieres, ahora puedo:

* A) Generar **todos** los archivos de la Fase 3 (código real) si copias/pegas **uno de los prompts** aquí y me pides que lo genere (yo crearé el contenido del archivo solicitado).
* B) Entregar la **checklist** compacta de Fase 3 en formato que puedas marcar (si prefieres un check-list imprimible).

**Fase 4 — Cross-cutting, pruebas y despliegue (prompts listos)**

**4.0 — Resumen de objetivos de la fase**

**Prompt:**

Entrega un plan de Fase 4 para el proyecto Zentro que cubra: contenedorización (Docker + Docker Compose), migraciones (Alembic), CI/CD (GitHub Actions que construya, pruebe y despliegue), pruebas automáticas (unit/integration/e2e), observabilidad (metrics, logs, tracing), gestión de secretos y seguridad (HTTPS, rotación de secretos), backups y DR (pg\_dump, PITR), escalado y configuración de producción (uvicorn/gunicorn, workers), y scripts de despliegue/rollback. El plan debe incluir comandos exactos, nombres de archivos a crear, y ejemplos de configuración mínima para cada herramienta.

**4.1 — Migraciones y gestión del esquema (Alembic)**

**4.1.1 — Configurar Alembic (env.py, alembic.ini)**

**Prompt:**

Genera los archivos y la configuración inicial de Alembic para app:

* alembic.ini con conexión controlada por env var (DATABASE\_URL).
* alembic/env.py que importe app.db.base\_class.Base.metadata (usar path relativo from app.db.base\_class import Base) y configure run\_migrations\_online() con engine\_from\_config.
* Un Makefile o scripts en pyproject con comandos: alembic revision --autogenerate -m "msg", alembic upgrade head.
* Añade en README sección corta: cómo generar revision y aplicarla en CI.  
  Incluye ejemplos concretos de comandos y cómo ejecutar en local y en CI.

**4.1.2 — Primer migration «Initial»**

**Prompt:**

Crea la primera migration autogenerada versions/xxxx\_initial.py incluyendo todas las tablas de Fase 1-3. Debe contener upgrade() y downgrade() y usar op.create\_table() para cada modelo. Proporciona el comando exacto para generarla (alembic revision --autogenerate -m "Initial models") y la instrucción para aplicarla (alembic upgrade head).

**4.2 — Contenedores (Docker & Docker Compose)**

**4.2.1 — Dockerfile de producción**

**Prompt:**

Genera un Dockerfile optimizado para producción para FastAPI:

* Multi-stage build: base python slim -> install deps -> copy -> final stage con sólo runtime.
* Usa gunicorn + uvicorn.workers.UvicornWorker en el CMD (no uvicorn.run() dentro de app). Ejemplo de CMD:

gunicorn -k uvicorn.workers.UvicornWorker app.main:app \

--workers 4 --bind 0.0.0.0:8000 --timeout 120

* Exponer puerto 8000, variables de entorno para DB y settings.
* Buenas prácticas: no ejecutar como root (user non-root), reducir capas, usar .dockerignore.  
  (Basado en guías oficiales y prácticas recomendadas). [fastapi.tiangolo.com+1](https://fastapi.tiangolo.com/deployment/docker/?utm_source=chatgpt.com)

**4.2.2 — docker-compose para desarrollo y stack mínimo producción**

**Prompt:**

Genera docker-compose.yml con servicios: web (la app), db (postgres), redis (cache/celery), prometheus y grafana (stack observability opcional). Incluir volúmenes para postgres y configuración mínima de red. Añade un docker-compose.override.yml para desarrollo con hot-reload (uvicorn --reload). Proporciona comandos docker-compose up --build y docker-compose -f docker-compose.prod.yml up -d para producción.

**4.3 — CI / CD (GitHub Actions)**

**4.3.1 — Workflow: build → test → image → push → deploy**

**Prompt:**

Genera un workflow de GitHub Actions .github/workflows/ci\_cd.yml que haga:

1. Trigger: push a main y pull\_request.
2. Jobs:
   * lint-and-test: setup Python, install deps, run pytest -q --cov, flake8/ruff.
   * build-and-push: build Docker image, tag (commit sha) y push a GHCR o Docker Hub (usar secrets: REGISTRY, REGISTRY\_USER, REGISTRY\_TOKEN).
   * deploy: despliegue por SSH a servidor o despliegue a un proveedor (ej.: Docker Compose en server remoto o usar kubectl para K8s).
3. Guardar artifacts (coverage) y publicar reportes (summary).  
   Incluye snippet para manejar secrets y ejemplo de secrets a configurar en el repo. Provee un ejemplo de job para desplegar via SSH usando appleboy/ssh-action.

Referencia de ejemplo/plantilla para CI/CD. [PyImageSearch](https://pyimagesearch.com/2024/11/11/fastapi-with-github-actions-and-ghcr-continuous-delivery-made-simple/?utm_source=chatgpt.com)[Medium](https://medium.com/%40saveriomazza/setting-up-a-ci-cd-pipeline-for-dockerized-fastapi-service-on-github-actions-d1c333d20880?utm_source=chatgpt.com)

**4.4 — Pruebas: unitarias, integración y E2E**

**4.4.1 — Estrategia de pruebas**

**Prompt:**

Documenta y genera scripts/tests:

* pytest para unit tests (tests/unit/) y integration tests (tests/integration/) usando pytest-asyncio.
* Fixtures: db\_session en SQLite in-memory o contenedor Postgres para integración con docker-compose.
* E2E: usar pytest + httpx.AsyncClient para probar endpoints completos; opcional: Playwright/Selenium para pruebas UI.
* Añadir coverage y codecov job en CI.
* Añadir ejemplos mínimos de tests para user\_service.create\_user() y api/v1/auth login.

**4.4.2 — Test DB & Migrations en CI**

**Prompt:**

En el workflow CI, añade steps para crear una base de datos Postgres (service) o usar sqlite in-memory, aplicar migrations (alembic upgrade head) y luego ejecutar tests de integración. Proporciona el YAML snippet que aplica migraciones antes de ejecutar pytest.

**4.5 — Observabilidad (métricas, logs, tracing)**

**4.5.1 — Métricas Prometheus**

**Prompt:**

Genera un ejemplo de integración con prometheus-fastapi-instrumentator:

* Código para exponer /metrics y añadir middleware para medir latencias y contadores por endpoint.
* Añade configuración en docker-compose para Prometheus scrape (job) apuntando al web:8000/metrics.
* Incluye prometheus.yml mínimo y ejemplos de dashboards básicos en Grafana. [GitHub](https://github.com/trallnag/prometheus-fastapi-instrumentator?utm_source=chatgpt.com)[kubernetestraining.io](https://kubernetestraining.io/blog/fastapi-prometheus-monitoring?utm_source=chatgpt.com)

**4.5.2 — Logging estructurado & centralizado**

**Prompt:**

Genera app/core/logging.py que configure:

* Logging estructurado (ej. structlog o python-json-logger).
* Logging a stdout (para contenedores) y a archivo rotativo opcional.
* Ejemplo de integración con Loki (Grafana Loki) o con un collector (Fluentd/Vector) vía docker compose.

**4.5.3 — Error tracking (Sentry)**

**Prompt:**

Genera snippet core/sentry.py con inicialización condicional (SENTRY\_DSN env var). Añade middleware que capture excepciones y trace\_id. Indica cómo añadir SENTRY\_DSN en secrets y cómo ver errores en Sentry. (Recomendado para errores en producción.)

**4.5.4 — Tracing (OpenTelemetry)**

**Prompt:**

Genera un ejemplo mínimo para instrumentar con OpenTelemetry (traces) y exportar a Jaeger/OTLP. Incluye init snippet y middleware para correlación entre logs y traces.

**4.6 — Seguridad, secretos y HTTPS**

**4.6.1 — Gestión de secretos**

**Prompt:**

Documenta y genera ejemplos para manejar secretos:

* Locally: .env (no commitear) y python-dotenv en dev.
* CI/Prod: usar GitHub Secrets o un Secrets Manager (HashiCorp Vault, AWS Secrets Manager).
* Añadir snippet para core/config.py que priorice variables de entorno y provea validación.
* Indicar rotación periódica de claves y cómo invalidar tokens.

**4.6.2 — HTTPS / certificados**

**Prompt:**

Genera instrucciones para poner HTTPS en producción:

* Opciones: usar un reverse proxy (Nginx) con Certbot (Let's Encrypt) o usar plataforma que provea TLS (Cloud Run, App Platform).
* Proporciona snippet de configuración Nginx para proxy\_pass al contenedor en 8000 y renovar certificados con Certbot.

**4.6.3 — Hardening**

**Prompt:**

Lista de chequeo y acciones automáticas:

* Revisar dependencias con pip-audit/safety.
* Ejecutar scanner de vulnerabilidades de imágenes (Grype/Syft) en CI.
* Deshabilitar endpoints sensibles en prod y proteger docs (o proteger /docs con auth).
* Revisión de CORS y rate limiting (implementar slowapi o middleware similar).

**4.7 — Backups y Disaster Recovery (Postgres)**

**4.7.1 — Backups regulares (pg\_dump) y restores**

**Prompt:**

Genera scripts en scripts/db\_backup.sh y scripts/db\_restore.sh:

* db\_backup.sh: PGPASSWORD=$PGPASSWORD pg\_dump -Fc -h $DB\_HOST -U $DB\_USER -d $DB\_NAME -f /backups/db-$(date +%F-%H%M).dump y rotación de backups por retenzione.
* db\_restore.sh: pg\_restore -h $DB\_HOST -U $DB\_USER -d $DB\_NAME --clean --no-owner /backups/<file>.
* Añadir ejemplo de cron job o GitHub Actions scheduled workflow para backups automáticos.
* Recomendación: probar restores periódicamente. [PostgreSQL](https://www.postgresql.org/docs/current/app-pgdump.html?utm_source=chatgpt.com)[Crunchy Data](https://www.crunchydata.com/blog/introduction-to-postgres-backups?utm_source=chatgpt.com)

**4.7.2 — PITR & WAL archiving**

**Prompt:**

Genera guía y scripts para habilitar WAL archiving en Postgres (PITR):

* Instrucciones de postgresql.conf (wal\_level=replica, archive\_mode=on, archive\_command con script que suba WAL a storage seguro).
* Script de archivado y cómo restaurar a un punto en el tiempo.
* Nota sobre elegir herramienta (Barman/pgBackRest) para entornos producción. [PostgreSQL](https://www.postgresql.org/docs/current/continuous-archiving.html?utm_source=chatgpt.com)[Percona](https://www.percona.com/blog/postgresql-backup-strategy-enterprise-grade-environment/?utm_source=chatgpt.com)

**4.8 — Escalado y rendimiento**

**4.8.1 — Uvicorn/Gunicorn tuning**

**Prompt:**

Genera recomendaciones y script run\_gunicorn.sh:

* Reglas: workers = (2 x $CPU) + 1 como baseline; ajustar threads, timeout y keepalive.
* Ejemplo de comando (como en Dockerfile): gunicorn -k uvicorn.workers.UvicornWorker app.main:app --workers 4 --bind 0.0.0.0:8000 --timeout 120. [fastapi.tiangolo.com](https://fastapi.tiangolo.com/deployment/server-workers/?utm_source=chatgpt.com)

**4.8.2 — Caching y cola de tareas**

**Prompt:**

Genera guía para integrar Redis (cache) y Celery (o RQ) para tareas asíncronas:

* docker-compose service para redis.
* Ejemplo de task en tasks/email.py y cómo ejecutar worker (celery -A app.worker worker --loglevel=info).
* Uso de cache TTL y invalidaciones.

**4.9 — Healthchecks, readiness/liveness probes y rollbacks**

**4.9.1 — Health endpoints**

**Prompt:**

Genera endpoint GET /health que verifique: conexión DB, redis (opcional) y retorna 200 y {"status":"ok","db":true}. Incluir GET /ready y GET /live. Añadir tests.

**4.9.2 — Rollback plan**

**Prompt:**

Documenta un plan de rollback para despliegues:

* Mantener última imagen estable etiquetada stable.
* Script deploy.sh --rollback que vuelva a stable y aplique docker-compose up -d --no-deps --scale web=1 web=stable.
* Notas sobre DB migrations: si la migration no es reversible, bloquear rollback o aplicar migration compensatoria.

**4.10 — Infraestructura como código y despliegue avanzado**

**4.10.1 — Terraform (opcional)**

**Prompt:**

Genera ejemplo mínimo de terraform para levantar: VPC, instancia (or managed service), security groups, y una base RDS/managed Postgres. Proporciona estructura y variables. Indicar cómo integrarlo con CI.

**4.10.2 — Kubernetes (opcional)**

**Prompt:**

Genera manifests básicos para K8s: Deployment, Service, Ingress con TLS, ConfigMap, Secret (sugerencia de usar External Secrets), HorizontalPodAutoscaler y Probe config para liveness/readiness. Añade un helm chart skeleton opcional.

**4.11 — Documentación y runbook operativo**

**4.11.1 — Documentar runbooks**

**Prompt:**

Genera un docs/OPERATIONAL.md (runbook) con:

* Cómo desplegar (comandos), cómo restaurar DB, cómo acceder a logs, how-to rollback, checklist pre-despliegue, runbook de incidentes (steps para P0, P1).
* Incluir playbooks para escalado, seguridad y recuperación.

**4.11.2 — API docs & versioning**

**Prompt:**

Recomienda política de versionado (v1, v2) y cómo mantener compatibilidad. Añade ejemplo de cómo proteger /docs con auth en producción (HTTP Basic o token).

**4.12 — Checklist final y prioridades inmediatas**

**Prompt:**

Genera una checklist priorizada (tareas en orden) para Fase 4 que pueda seguir un equipo:

1. Configurar Alembic y generar primera migration.
2. Crear Dockerfile y .dockerignore.
3. Crear workflow CI para tests y build de image.
4. Configurar Prometheus scrapping y /metrics.
5. Establecer backups básicos (pg\_dump + retention).
6. Preparar runbook y health endpoints.
7. Añadir Sentry y logging estructurado.
8. Preparar deploy a entorno staging.  
   Entregar comandos exactos para ejecutar cada item.

**Fase 5 — Operaciones y mejoras (Prompts listos)**

Objetivo de la Fase 5: poner Zentro en un modo operativo sólido y productivo: monitorización completa, logging avanzado y centralizado, caching y optimizaciones para baja latencia, SRE/operaciones (SLOs, runbooks, on-call), seguridad operativa, pruebas de rendimiento y mejora continua.

**A — Monitorización y Alerting**

**Prompt: metrics/collector\_setup**

Genera un módulo app/observability/metrics.py para exponer métricas de aplicación usando prometheus-fastapi-instrumentator o prometheus\_client.

* Exponer /metrics.
* Métricas obligatorias: request\_count (por método/endpoint/status), request\_latency\_seconds (histograma), db\_query\_duration\_seconds, cache\_hit\_ratio, background\_tasks\_queue\_length.
* Instrumenta middlewares para medir latencia por ruta y método.
* Añade docstring con ejemplos para consultas PromQL básicas (p95 latency, errors/sec).

**Prompt: prometheus/config**

Genera monitoring/prometheus.yml con job que haga scrape a web:8000/metrics, redis:9121 (si aplica) y postgres\_exporter. Incluye reglas de alerting básicas (p.ej: high error rate >5% durante 5m, p95 latency > 1s durante 5m, db connections ~ > 90%) y un rules file ejemplo.

**Prompt: grafana/dashboards**

Genera 3 dashboards JSON para Grafana:

1. **App Overview**: requests/sec, p50/p95/p99 latency, error rate, active requests, CPU/memory.
2. **DB & Cache**: DB connections, slow queries, cache hit ratio, eviction rate.
3. **Background workers**: queue length, task durations, failed tasks.  
   Añade paneles con ejemplos de PromQL para cada métrica.

**Prompt: alerting/policies**

Genera docs/ALERTING.md con políticas: severities (P0/P1/P2), canales (Slack, PagerDuty, correo), criterios de alerta y runbook resumen por alerta (qué chequear primero: health endpoints, logs, recent deploys). Añade ejemplos de reglas en Prometheus Alertmanager format.

**Prompt: synthetic\_checks**

Genera scripts/synthetic\_check.sh y monitoring/synthetic.yml para pruebas sintéticas (health + login + create client + book class) ejecutadas cada 5m por un synthetic runner (puede ser Grafana Synthetic, UptimeRobot o cronjob). Incluir ejemplo de fallo y alerta asociada.

**B — Logging avanzado y centralizado**

**Prompt: logging/structured**

Genera app/core/logging.py que configure logging estructurado usando python-json-logger (o structlog) y que:

* Añada request\_id/trace\_id a cada log (middleware que inyecte o propague X-Request-ID).
* Loguee request/response summary (method, path, status, duration, user\_id si aplica) en nivel INFO.
* Loguee stack traces en nivel ERROR con campos estructurados.
* Salida por stdout (formato JSON) para consumo por un collector.

**Prompt: log-forwarder-config**

Genera monitoring/log\_forwarder/values.yaml ejemplo para Vector/Fluentd/FluentBit que:

* Recoja logs de containers (stdout), los transforme (parse JSON), agregue metadata (pod/container labels), y los envíe a Loki/Elasticsearch/Cloud provider.
* Incluir reglas de retención y parsing para stack traces.

**Prompt: alerts-from-logs**

Genera ejemplos de alertas basadas en logs (p.ej. X enrros del tipo "database connection refused" en 1m) y un script scripts/query\_logs.sh que use API de Loki/ES para buscar patrones y generar reporte.

**C — Tracing y correlación (OpenTelemetry)**

**Prompt: tracing/opentelemetry\_init**

Genera app/observability/tracing.py con inicialización de OpenTelemetry: OTLP exporter (env var OTEL\_EXPORTER\_ENDPOINT), instrumentación de FastAPI, SQLAlchemy and Celery workers.

* Asegura que trace\_id se propague a logs y métricas.
* Añade ejemplo de span custom en service (p.ej. with tracer.start\_as\_current\_span("book\_class")).

**Prompt: jaeger/docker**

Genera monitoring/jaeger-docker-compose.yml con Jaeger (collector, query, ui) y ejemplo de cómo apuntar el OTLP exporter. Incluir instrucciones para ver traces y correlacionarlos con logs.

**D — Caching y optimizaciones de rendimiento**

**Prompt: cache/redis\_integration**

Genera app/core/cache.py con integración Redis: funciones get\_cache(key), set\_cache(key, value, ttl), cache\_memoize(ttl) decorator.

* Ejemplos de uso para endpoints pesados: listado de clases, trainer availability.
* Métricas para cache hit/miss e invalidación.

**Prompt: cache/invalidation\_patterns**

Genera docs/CACHE\_PATTERNS.md con patrones: cache-aside, write-through, time-based invalidation, tag-based invalidation. Ejemplos concretos para classes, products y routines.

**Prompt: cdn/static\_assets**

Genera docs/CDN\_SETUP.md con pasos para ubicar imágenes/public assets en CDN (CloudFront/Cloudflare): generación de URLs firmadas para assets privados y cache headers óptimos (Cache-Control, ETag, immutable).

**Prompt: db/query\_optimization**

Genera scripts/db\_profiler.py que conecte a Postgres e identifique queries lentas (pg\_stat\_statements) y genere un reporte con candidate indexes. Añade checklist de optimización (indexes compuestos, EXPLAIN ANALYZE, partitioning, VACUUM/ANALYZE).

**E — Background jobs, rate-limiting y resiliencia**

**Prompt: tasks/celery\_setup**

Genera app/worker/celery\_app.py con configuración Celery (broker=redis), ejemplo de task send\_email y tasks/process\_membership\_expiration. Incluir retry strategy (exponential backoff), idempotency keys y monitoring of failed tasks.

**Prompt: rate\_limit/middleware**

Genera app/core/rate\_limiter.py usando slowapi o middleware custom con Redis: rate limits por IP y por user, headers Retry-After, y manejo de status 429. Incluye tests de comportamiento.

**Prompt: circuit\_breaker**

Genera app/core/circuit\_breaker.py con patrón de circuit breaker simple (fallback, threshold, cool-down) para llamadas externas (pago gateway). Ejemplo de integración con services/store\_service.

**F — SLOs, SLAs, runbooks y on-call**

**Prompt: slos\_and\_slas**

Genera docs/SLOS\_AND\_SLAS.md con: SLI propuestas (success rate, latency p95, availability), SLO targets (ej: availability 99.9 monthly, p95 < 500ms), y cómo medirlos. Añade indicadores de error budget y acciones cuando se consuma.

**Prompt: runbook/p0\_p1**

Genera docs/RUNBOOK.md con playbooks accionables para P0 (app down) y P1 (payment failure): pasos a seguir, checklists (verificar health, últimas deploys, métricas clave, rollbacks), comandos útiles (docker-compose logs, psql quick queries), y contactos on-call.

**Prompt: oncall\_rotation**

Genera plantilla docs/ONCALL.md con política de rotation (ej: 1 semana por ingeniero), escalación a 2do nivel, contacto de emergencia, cómo hacer postmortem y how-to for handover.

**G — Seguridad operativa y hardening continuo**

**Prompt: waf\_and\_protection**

Genera docs/WAF\_AND\_SECURITY.md con configuración sugerida para WAF (Cloudflare WAF rules o ModSecurity), reglas básicas (SQLi, XSS, rate limit, bruteforce login), y ejemplos para bloquear patrones de ataque. Añadir pasos para whitelisting y tests.

**Prompt: dependency\_scanning**

Genera scripts/ci\_security\_scan.sh que ejecute pip-audit y safety en CI, fail build en caso de CVE crítico, y genere report automatizado. Incluir ejemplo de configuración para GitHub Actions.

**Prompt: secret\_detection**

Genera docs/SECRET\_MANAGEMENT.md con reglas: no guardar secretos en repo, usar GitHub Secrets / Vault, config para git-secrets pre-commit hook y procedimientos para rotación de claves.

**H — Database maintenance & operational tasks**

**Prompt: db/maintenance\_tasks**

Genera scripts/db\_maintenance.sh con tasks: VACUUM (VERBOSE, ANALYZE), REINDEX, pg\_stat\_statements rotation, and nightly routine. Añade crontab example and runbook for when autovacuum lag detected.

**Prompt: db/partitioning\_strategy**

Genera docs/DB\_PARTITIONING.md con criterios para particionar (attendances by month, events by date), cómo crear partitions in Postgres (declarative partitioning), and migration strategy for existing data.

**Prompt: read\_replica\_setup**

Genera docs/READ\_REPLICA.md con steps to set up read-replicas (streaming replication), promote replica, connection pooling considerations (pgbouncer), and how to route read-only queries.

**I — Performance testing & chaos**

**Prompt: perf/load\_test**

Genera load\_tests/locustfile.py (o k6 script) que simule: login flow, book class, list classes, create client. Incluir target RPS, ramp-up scenarios, and thresholds (p95 < X). Add instructions to run in docker-compose to run against staging.

**Prompt: chaos/chaoskit**

Genera docs/CHAOS\_TESTING.md with small chaos experiments: kill worker, saturate DB CPU, introduce latency to external API. Include safety rules (do not run on production without approval), rollback steps and incident notification plan.

**J — Observability maturity: dashboards & KPIs**

**Prompt: dashboards/kpi\_list**

Genera docs/KPIS.md listing core KPIs to monitor weekly/monthly (MAU, bookings/day, revenue/day, conversion funnel, average class fill-rate, avg. membership churn). For each KPI include data source and suggested Grafana panel.

**Prompt: dashboard\_templates**

Genera Grafana JSON templates for SLO dashboard (error budget burn-down), and a Postgres performance dashboard (top queries by total\_time, avg\_time, calls).

**K — Developer experience & feature flags**

**Prompt: feature\_flags**

Genera app/core/feature\_flags.py (simple implementation backed by Redis or LaunchDarkly). Include toggle examples: FEATURE\_BOOKING\_V2. Add migration strategy for flags (default off) and tests.

**Prompt: ci\_release\_notes**

Genera scripts/generate\_release\_notes.py that compiles commits since last tag into a changelog and attaches it to the release workflow. Add integration example with GitHub Releases in CI.

**L — Retention, cost control & data governance**

**Prompt: retention/policies**

Genera docs/RETENTION.md with retention policies: logs (30d hot, 365d cold), metrics (90d), traces (30d), backups (90d offsite). Add cost estimation notes and cold/archival storage steps (S3 Glacier).

**Prompt: pii\_handling**

Genera docs/PII\_COMPLIANCE.md with guidance: what fields are PII, encryption at rest/in transit, access controls (RBAC), data deletion workflow (subject access requests) and example SQL to anonymize old data.

**M — Checklist final y prioridades inmediatas**

**Prompt: phase5/checklist**

Genera docs/PHASE5\_CHECKLIST.md priorizado:

1. Exponer /metrics y configurar Prometheus scrape + basic alerts.
2. Logging estructurado y forwarder (Vector/FluentBit → Loki/ES).
3. Tracing básico (OpenTelemetry → Jaeger/OTLP).
4. Cache Redis + pattern implemented on hot endpoints.
5. Background tasks con retries/monitoring.
6. SLOs definidos y dashboard SLO.
7. Runbooks P0/P1 y on-call rota.
8. Performance tests & baseline.
9. Security scans in CI.
10. Chaos small experiment in staging.  
    Para cada item añade comandos concretos a ejecutar, owners (role), y criterios de “done”.

**Fases / áreas adicionales (opcional, recomendadas según crecimiento)**

Si quieres ir más allá y preparar la compañía para escalar, puedes añadir fases extra enfocadas en producto, cumplimiento y crecimiento:

* **Fase 6 — Crecimiento & Producto**
  + Instrumentación analítica avanzada (product analytics: Amplitude/Heap), A/B testing, funnels, onboarding optimizado, email/marketing automation, CRM.
* **Fase 7 — Pagos, legal y finanzas**
  + Integración de pasarelas de pago (PCI compliance), facturación recurrente, impuestos, políticas de privacidad, términos de servicio, contabilidad y reconciliación.
* **Fase 8 — Mobile & SDKs**
  + Apps móviles nativas (iOS/Android) o wrappers, SDKs para integraciones, push notifications.
* **Fase 9 — Internacionalización**
  + Localización (i18n), formatos regionales, multi-moneda, adaptación legal/regulatoria por país.
* **Fase 10 — Data & ML**
  + Almacenamiento analítico (data warehouse), pipelines ETL, modelos ML para recomendaciones (rutinas, productos), predicción de churn.
* **Fase 11 — Partnerships & Ecosystem**
  + Marketplace, integraciones con terceros (calendarios, pagos, wearables), APIs públicas y documentación para partners.
* **Fase 12 — Crecimiento organizacional**
  + Hiring, estructura de equipos, procesos de ventas/CS, soporte al cliente, SLAs comerciales.