API-Potters

1. Estructura general y configuración

Aspectos positivos

- Existe una separación clara entre:
  - `core/dominio` (modelos e interfaces de dominio),
  - `core/aplicacion/casos-uso` (servicios de orquestación y lógica de aplicación),
  - `core/infraestructura/postgres` (acceso a datos),
  - `presentacion` (Fastify, controladores, rutas, esquemas Zod),
  - `common/configuracion.ts` (configuración compartida).
- El archivo `common/configuracion.ts` centraliza la lectura de variables de entorno y el módulo `clientePostgres.ts` utiliza esa configuración para inicializar el `Pool` de PostgreSQL, lo que facilita cambiar credenciales sin tocar el resto del código.

Aspectos a mejorar

- La configuración no tiene validación ni valores por defecto. Se hace directamente:

  ```
  httpPuerto: Number(process.env.PUERTO),
  ```

  Si una variable no está definida, se obtiene `NaN` o `undefined` en tiempo de ejecución sin una señal clara. Sería recomendable:
  - Validar las variables de entorno con Zod o similar.
  - Lanzar un error de configuración temprano si falta algún valor crítico.
- Sería útil agrupar más explícitamente la configuración en un módulo `config/` (por ejemplo, `config/http`, `config/database`) para escalar mejor cuando el proyecto crezca.

2. Capa de dominio

Aspectos positivos

- Se definen interfaces de dominio claras para las entidades principales:
  - `IAsignatura`, `IPrograma`, `IOferta`, `IPeriodoAcademico`, `IPlanEstudio`, etc.
- Existen variantes "relacionadas" cuando se necesita enriquecer el modelo con joins:
  - `IOfertaRelacionada`, `IPeriodoRelacionado`, `IPlanEstudioRelacionado`, lo que ayuda a diferenciar entre datos "propios" de la entidad y vistas más ricas para lectura.
- Se introdujo un error de dominio específico: `ValidacionError`, pensado para representar errores de validación de negocio.

Aspectos a mejorar

- Inconsistencia en nombres de clases:
  - `export class asignatura`, `planEstudio`, `programa` (comienzan en minúscula) frente a `PeriodoAcademico`, `Oferta`.
  - Esto es poco idiomático en TypeScript y complica la lectura. Lo esperable es que todas las clases comiencen en mayúscula (`Asignatura`, `Programa`, `PlanEstudio`).
- Algunas entidades son meros "contenedores de datos" que replican la interfaz sin añadir comportamiento:

```
export class asignatura implements IAsignatura {
  // copia directa de propiedades sin lógica
}
```

  Si no hay lógica de dominio (métodos, invariantes, reglas), es discutible si se justifica una clase. O bien:

  - se les agrega comportamiento de dominio (validaciones, métodos de negocio, fábricas), o

    - se prescinde de la clase y se trabaja con tipos/DTOs.
- En `PeriodoAcademico.ts`:

  ```
  export class PeriodoAcademico implements PeriodoAcademico { ... }
  ```

  La clase se declara implementando a sí misma en lugar de implementar `IPeriodoAcademico`, lo cual es un error conceptual y de tipado.
- Inconsistencias de tipo:
    - `IOferta.cupo: number;` pero en `Oferta`:

      ```
      this.cupo = datosOferta.cupo ?? null;
      ```

      Esto contradice la interfaz y puede generar problemas de tipado o de asunciones en otros puntos.
- El estado del periodo académico (`idEstado`) es un número “mágico” (1, 2, 3) en varias partes. Sería deseable:
    - tener un `enum` o un tipo más expresivo,
    - y encapsular la lógica de transición de estado dentro de la entidad, no en la capa de aplicación.

---

3. Capa de aplicación (Casos de uso)

Aspectos positivos

- Los casos de uso siguen una estructura clara: orquestan repositorios, aplican reglas de negocio y devuelven modelos de dominio o modelos relacionados.
    - `OfertaCasosUso` valida existencia de programa, asignatura y periodo, verifica duplicados y estado del periodo.
    - `PeriodoAcademicoCasosUso` encapsula reglas de solapamiento de fechas y transición de estados.
    - `PlanEstudioCasosUso` realiza comprobaciones de existencia y duplicidad y utiliza `ValidacionError` para errores de negocio.

- El diseño general respeta la idea de que los controladores llaman a casos de uso y estos llaman a repositorios, evitando mezclar SQL o detalles de infraestructura en la capa de presentación.

Aspectos a mejorar

- Fuga de dependencias hacia la capa de presentación:
  - Interfaces como `IAsignaturaCasosUso`, `IOfertaCasosUso`, `IPeriodoAcademicoCasosUso`, `IPlanEstudioCasosUso`, `IProgramaCasosUso` importan DTOs desde `presentacion/esquemas/...`.
  - Esto rompe el principio de independencia: la capa de aplicación no debería depender de la capa de presentación.
- Inconsistencia entre interfaces y clases:
  - `IAsignaturaCasosUso.crearAsignatura(asignatura: AsignaturaDTO)`

    pero `AsignaturaCasosUso.crearAsignatura(datosAsignatura: IAsignatura)`.
  - `IOfertaCasosUso.crearOferta(datosOferta: OfertaDTO)`

    pero `OfertaCasosUso.crearOferta(datosOferta: IOferta)`.
  - `IPlanEstudioCasosUso.actualizarPlanEstudio(..., planEstudio: IPlanEstudio)`

    pero en el controlador se valida con Zod y luego se castea a `IPlanEstudio`.

  Esto genera confusión y debilita el contrato. Lo ideal es:
  - definir DTOs de entrada/salida en la capa de aplicación (o una capa de “contratos”), y
  - mantener aislado el hecho de que en presentación se use Zod.
- Manejo de errores heterogéneo:
  - En `PlanEstudioCasosUso` se usa `ValidacionError`.
  - En `OfertaCasosUso` y `PeriodoAcademicoCasosUso` se usan `Error` genéricos con mensajes específicos.
  - Eso obliga a la capa de presentación a inspeccionar `error.message` mediante substrings para decidir si responder 400, 404 o 422. Es frágil y difícil de mantener.

- Typos y consistencia:
  - En `IPlanEstudioCasosUso` y `PlanEstudioCasosUso` aparece `eliminiarPlanEstudio` (con doble “i”).
  - Nombrar correctamente los métodos es importante para la claridad y para evitar errores en el futuro.

---

4. Capa de infraestructura (repositorios PostgreSQL)

Aspectos positivos

- Todos los repositorios siguen un patrón consistente:
  - reciben interfaces de dominio (`IAsignatura`, `IPrograma`, etc.),
  - generan dinámicamente sentencias de `INSERT` / `UPDATE` en función de las propiedades del objeto,
  - usan `ejecutarConsulta` como punto único de acceso a la base de datos.
- `clientePostgres.ts` configura un `Pool` único y define un parser de tipos para fechas (`types.setTypeParser(1082, val ⇒ val)`), lo que ayuda a homogeneizar el tratamiento de `DATE`.
- En algunos métodos, se usan alias para mapear columnas a nombres de propiedades camelCase:

```
SELECT idperiodo, semestre, fechainicio, fechafin, idestado AS "idEstado"
```

Aspectos a mejorar

- Posibles inconsistencias en nombres de columnas devueltas:
  - Por ejemplo, en `AsignaturaRepositorio.crearAsignatura`, se hace:

    ```
    return respuesta.rows[0].idAsignatura;
    ```

    pero el `INSERT INTO asignatura` genera columnas en minúscula (`idasignatura`), y los resultados de `pg` por defecto vienen en minúscula. Algo similar ocurre en `ProgramaRepositorio` (`idPrograma` vs `idprograma`).

    - Sería más seguro:
        - usar alias en las consultas (`AS "idAsignatura"`), o
        - normalizar todos los accesos a `rows[0].idasignatura` y usar interfaces de tipo para filas.
- Uso de `Object.keys()` + `toLowerCase()` para construir columnas:
    - Se asume que `nombrePrograma` → `nombreprograma` coincide con la columna real.
    - Es una convención válida, pero muy frágil: si en algún momento hay un nombre con mayúsculas intermedias o se introduce una columna con guiones bajos (`nombre_programa`), el patrón se rompe.
    - En operaciones críticas (como fechas en `PeriodoAcademicoRepositorio`), además se escriben fechas sobre `parametros[1]` y `parametros[2]` asumiendo el orden de las keys, lo cual aumenta el riesgo de errores si se añaden campos.
- La lógica de conversión de fechas a `YYYY-MM-DD` está incrustada en los repositorios. Podría extraerse a una pequeña utilidad de fecha para evitar duplicación y mejorar testabilidad.

---

5. Capa de presentación (Fastify: app, controladores y rutas)

Aspectos positivos

- `presentacion/app.ts` concentra el arranque de Fastify, registra los enrutadores con un prefijo común (`/api/Academium`) y aprovecha la centralización de `configuration.httpPuerto` .
- Para cada agregado (asignatura, oferta, periodo, plan de estudio, programa) se sigue un patrón claro:
    - Ruta de construcción (`construirXEnrutador`),
    - Instanciación de repositorio → caso de uso → controlador,
    - Y registro de las rutas.
- Los controladores:

    - devuelven mensajes claros en las respuestas,
    - utilizan Zod para validar el body en las operaciones de creación (y en algunos casos también actualización),
    - encapsulan la lógica de parsing de errores de validación (`ZodError`).

Aspectos a mejorar

- Inconsistencia en validación:
    - En creación, la mayoría usa Zod (`CrearAsignaturaEsquema`, `crearProgramaEsquema`, etc.).
    - En muchas actualizaciones se acepta directamente una interfaz de dominio (`IPrograma`, `IAsignatura`) sin aplicar Zod ni restricciones parciales.
    - Sería deseable:
        - reutilizar los esquemas Zod con `.partial()` para updates,
        - o definir esquemas específicos de actualización, manteniendo la simetría.
- Manejo de errores dependiente de los mensajes:
    - `OfertaControlador` y `PeriodoAcademicoControlador` revisan substrings del `error.message` para decidir si responden 400, 404 o 422.
    - Eso acopla la capa HTTP al texto concreto de los mensajes de las excepciones y es frágil a refactors de texto.
    - Dado que ya existe `ValidacionError`, sería recomendable:
        - extender ese enfoque a otros casos (por ejemplo `NotFoundError`, `ConflictError`),
        - y centralizar la traducción error → código HTTP en una función o hook.
- Nombres de controladores y rutas:
    - En general el patrón es coherente, pero hay detalles como:
        - `construirPlanEstudioControlador` (que en realidad construye el enrutador),

        - varias funciones `gestionAcademicaEnrutador` en archivos distintos.
    - Ayudaría que todos los enrutadores sigan una convención uniforme (`registerXRoutes` o similar).
- En algunas respuestas se mezclan nombres de campos con mayúsculas/minúsculas (`ProgramasEncontrados`, `AsignaturasEncontradas`, etc.). No afecta el funcionamiento, pero una convención consistente en JSON (por ejemplo, `snake_case` o `camelCase`) mejora la claridad de cara a clientes.

---

6. Validación y modelo de errores

Aspectos positivos

- El uso de Zod en los esquemas de `presentacion/esquemas` es sólido:
    - coerción de tipos (`z.coerce.number()`),
    - validaciones de rangos y formatos (fechas AAAA-MM-DD),
    - mensajes de error específicos y en contexto.
- La introducción de `ValidacionError` va en la dirección correcta: separar errores de negocio de errores genéricos.

Aspectos a mejorar

- Mezcla de enfoques:
    - Algunas rutas solo distinguen entre `ZodError` y `Error` genérico.
    - Otras rutas inspeccionan `message` buscando fragmentos de texto.
    - Otras diferencian explícitamente `ValidacionError`.
- No existe un manejador de errores global en Fastify:
    - Cada controlador repite lógica de `try/catch` y formateo de respuesta.
    - Un `setErrorHandler` global podría centralizar la conversión de:
        - `ValidacionError` → 400,
        - `NotFoundError` (si se introduce) → 404,

- `ConflictError` → 409,
- errores inesperados → 500.

---

7. Recomendaciones finales

1. **Mejorar consistencia entre capas**
   - Evitar que la capa de aplicación importe DTOs desde `presentacion/esquemas`.

     Definir modelos de entrada/salida de casos de uso dentro de `core/aplicacion` o en un módulo de contratos.
   - Alinear interfaces y clases de casos de uso para que usen los mismos tipos.
2. **Fortalecer el dominio**
   - Corregir nombres de clases (`Asignatura`, `Programa`, `PlanEstudio`).
   - Usar `IPeriodoAcademico` correctamente en la implementación de `PeriodoAcademico`.
   - Valorar si las clases de dominio deberían incorporar más reglas e invariantes (en lugar de ser solo “copiadoras de propiedades”).
3. **Unificar el manejo de errores**
   - Extender el patrón de `ValidacionError` a otros tipos de errores (por ejemplo: `EntidadNoEncontradaError`, `ReglaNegocioError`, etc.).
   - Evitar basar la lógica HTTP en substrings de `error.message`.
   - Implementar un manejador global de errores en Fastify.
4. **Refinar la infraestructura**
   - Revisar el acceso a campos de `rows[0]` para que coincida con los nombres reales de columnas (o utilizar alias sistemáticamente).
   - Evitar depender de la posición de claves (`Object.keys`) para mapear fechas.
   - Extraer utilidades comunes (formateo de fechas, transformaciones snake/camel) para evitar duplicación de lógica.
5. **Homogeneizar validación en controladores**

- Reutilizar esquemas Zod tanto en creación como en actualización (posiblemente con `.partial()`).
- Mantener un contrato claro de entrada/salida en cada endpoint.