**Paso 1 — Diseño del set de pruebas de API (Petstore)**

**1. Alcance y criterio**

* **API**: Petstore (Swagger UI).
* **Recurso elegido**: Pet. Razonamiento: el recurso **Pet** expone las 4 operaciones principales **POST, GET, PUT, DELETE** sobre el *mismo* dominio funcional, lo que permite **encadenar** pruebas (crear → consultar → actualizar → consultar → eliminar → verificar borrado).
* **Oráculo y aserciones base (aplican a todos los casos)**:
  + **Código HTTP esperado** según el escenario.
  + **Content-Type** application/json.
  + **Tiempo de respuesta** ≤ 2000 ms (criterio objetivo y fácil de medir).
  + **Estructura mínima** del JSON (campos clave presentes).
  + **Consistencia de datos** entre solicitud y respuesta (por ejemplo, id/name devueltos coinciden con lo enviado o con lo guardado previamente).
* **Variables y encadenamiento** (para Postman más adelante):
  + baseUrl
  + petId (generado dinámicamente para evitar colisiones)
  + petName, petStatus (p.ej., available | pending | sold)
  + Guardar petId de la respuesta del **POST** para usarlo en **GET/PUT/DELETE**.
* **Notas de robustez**: si durante la ejecución la API devuelve códigos diferentes a los documentados, **ajustaremos las aserciones** para reflejar el comportamiento real del endpoint observado en Swagger/UI el día de ejecución (en el README dejaremos constancia).

**2. Suite de casos por operación**

Convención de IDs: API-PET-<VERBO>-<n>  
Cada caso incluye: **Objetivo**, **Precondiciones**, **Request** (método + ruta + body si aplica), **Validaciones** (lo que se verificará) y **Por qué** (justificación QA).

**POST /pet — Crear mascota (5 casos)**

**API-PET-POST-01 — Positivo | Crear pet válido (happy path)**

* **Objetivo: validar creación exitosa de una mascota.**
* **Precondiciones: ninguna.**
* **Request: POST /pet.**
* **Datos de prueba: id único; name legible; status en catálogo permitido (p.ej., available).**
* **Validaciones: HTTP 200; id presente; name y status coinciden con lo enviado; Content-Type JSON; tiempo ≤ 2000 ms.**
* **Por qué: asegura el flujo base de creación y sienta datos para el encadenamiento.**

**API-PET-POST-02 — Negativo | JSON malformado / estructura inválida**

* **Objetivo: verificar manejo de request inválido.**
* **Precondiciones: ninguna.**
* **Request: POST /pet.**
* **Datos de prueba: estructura incompleta o mal formada (faltan llaves/campos obligatorios).**
* **Validaciones: error de cliente (400/422); mensaje descriptivo; no se crea recurso.**
* **Por qué: cubre validación sintáctica y evita datos corruptos.**

**API-PET-POST-03 — Negativo | Valor fuera de catálogo en status**

* **Objetivo: validar reglas de dominio (enumeración).**
* **Precondiciones: ninguna.**
* **Request: POST /pet.**
* **Datos de prueba: status con valor no permitido (p.ej., unknown\_status).**
* **Validaciones: 400/422; mensaje de validación; ningún recurso creado.**
* **Por qué: impide persistencia de valores no controlados.**

**API-PET-POST-04 — Negativo | Duplicidad de id**

* **Objetivo: comprobar manejo de colisión de identificadores.**
* **Precondiciones: existe un petId ya creado.**
* **Request: POST /pet.**
* **Datos de prueba: mismo id ya existente; resto de campos válidos.**
* **Validaciones: respuesta de error (409/400 según implementación) o reemplazo no permitido; mensaje claro; no altera el existente.**
* **Por qué: garantiza unicidad de la clave primaria.**

**API-PET-POST-05 — Negativo | Tamaño/extremos de campos (nombre muy largo o vacío)**

* **Objetivo: verificar límites y saneamiento de entradas.**
* **Precondiciones: ninguna.**
* **Request: POST /pet.**
* **Datos de prueba: name vacío o con longitud extrema/caracteres especiales.**
* **Validaciones: 400/422; mensajes de validación; no se crea recurso.**
* **Por qué: cubre límites de campo y posible inyección de entradas anómalas.**

**GET /pet/{petId} — Consultar mascota (5 casos)**

**API-PET-GET-01 — Positivo | Consultar pet existente**

* **Objetivo: recuperar una mascota creada previamente.**
* **Precondiciones: petId válido previamente creado.**
* **Request: GET /pet/{petId}.**
* **Datos de prueba: petId almacenado.**
* **Validaciones: HTTP 200; id = petId; name/status coherentes; Content-Type JSON; tiempo ≤ 2000 ms.**
* **Por qué: verifica lectura e integridad del recurso.**

**API-PET-GET-02 — Negativo | petId inexistente**

* **Objetivo: manejo correcto de no encontrado.**
* **Precondiciones: ninguna.**
* **Request: GET /pet/{petId}.**
* **Datos de prueba: petId alto/aleatorio no existente.**
* **Validaciones: HTTP 404; mensaje “not found”; cuerpo sin datos del recurso.**
* **Por qué: asegura respuestas consistentes para búsquedas inválidas.**

**API-PET-GET-03 — Negativo | petId no numérico / formato inválido**

* **Objetivo: validar tipado del parámetro de ruta.**
* **Precondiciones: ninguna.**
* **Request: GET /pet/{petId}.**
* **Datos de prueba: valor no numérico (p.ej., texto).**
* **Validaciones: 400/404 según validación de servidor; mensaje de error.**
* **Por qué: cubre errores de cliente por formato incorrecto.**

**API-PET-GET-04 — Negativo | petId vacío o nulo**

* **Objetivo: verificar manejo de parámetro faltante.**
* **Precondiciones: ninguna.**
* **Request: GET /pet/ (omisión).**
* **Datos de prueba: parámetro ausente.**
* **Validaciones: 404 (ruta) o 400 (validación); mensaje de error.**
* **Por qué: garantiza contratos claros de path params.**

**API-PET-GET-05 — Negativo | petId con caracteres especiales**

* **Objetivo: testear robustez ante caracteres especiales.**
* **Precondiciones: ninguna.**
* **Request: GET /pet/{petId}.**
* **Datos de prueba: petId con símbolos o espacios.**
* **Validaciones: 400/404; sin fuga de stack traces; mensaje controlado.**
* **Por qué: evita fallos por encoding/decoding en rutas.**

**PUT /pet — Actualizar mascota (5 casos)**

**API-PET-PUT-01 — Positivo | Actualizar nombre y estado**

* **Objetivo: validar actualización exitosa de campos principales.**
* **Precondiciones: petId existente.**
* **Request: PUT /pet.**
* **Datos de prueba: id = petId; name cambiado; status válido.**
* **Validaciones: HTTP 200; respuesta refleja cambios; GET posterior devuelve los nuevos valores.**
* **Por qué: confirma persistencia de cambios y lectura coherente.**

**API-PET-PUT-02 — Negativo | Falta id en body**

* **Objetivo: asegurar obligatoriedad del identificador en actualización.**
* **Precondiciones: ninguna.**
* **Request: PUT /pet.**
* **Datos de prueba: body sin id.**
* **Validaciones: 400/422; mensaje indica id requerido; no se actualiza nada.**
* **Por qué: protege reglas mínimas del contrato PUT.**

**API-PET-PUT-03 — Negativo | status inválido en actualización**

* **Objetivo: validar catálogo de status también en PUT.**
* **Precondiciones: petId existente.**
* **Request: PUT /pet.**
* **Datos de prueba: status fuera de catálogo.**
* **Validaciones: 400/422; mensaje de validación; no se aceptan cambios inválidos.**
* **Por qué: consistencia de reglas de negocio.**

**API-PET-PUT-04 — Negativo | Actualizar petId inexistente**

* **Objetivo: comprobar respuesta cuando se intenta actualizar un recurso que no existe.**
* **Precondiciones: ninguna.**
* **Request: PUT /pet.**
* **Datos de prueba: id que no existe; campos válidos.**
* **Validaciones: 404 o comportamiento documentado (no crear implícitamente); mensaje claro.**
* **Por qué: evita “upsert” no deseado y deja explícito el contrato.**

**API-PET-PUT-05 — No-op / Idempotencia | Enviar mismos datos sin cambios**

* **Objetivo: evaluar comportamiento cuando los valores enviados coinciden con los ya guardados.**
* **Precondiciones: petId existente con los mismos valores.**
* **Request: PUT /pet.**
* **Datos de prueba: id = petId; name/status iguales a los actuales.**
* **Validaciones: HTTP 200; sin cambios reales; datos retornados iguales; tiempos normales.**
* **Por qué: comprueba idempotencia/estabilidad del endpoint ante no-op.**

**DELETE /pet/{petId} — Eliminar mascota (5 casos)**

**API-PET-DELETE-01 — Positivo | Eliminar pet existente**

* **Objetivo: validar borrado exitoso.**
* **Precondiciones: petId existente.**
* **Request: DELETE /pet/{petId}.**
* **Datos de prueba: petId creado anteriormente.**
* **Validaciones: HTTP 200; mensaje de confirmación (si aplica); GET posterior devuelve 404.**
* **Por qué: asegura limpieza de datos y cierre del ciclo CRUD.**

**API-PET-DELETE-02 — Negativo | Eliminar pet inexistente**

* **Objetivo: manejo correcto al intentar borrar un recurso que no existe.**
* **Precondiciones: ninguna.**
* **Request: DELETE /pet/{petId}.**
* **Datos de prueba: petId aleatorio no existente.**
* **Validaciones: 404 (o comportamiento documentado); mensaje claro; no efectos colaterales.**
* **Por qué: consistencia del contrato ante operaciones inválidas.**

**API-PET-DELETE-03 — Negativo | petId no numérico**

* **Objetivo: validar tipado del parámetro en borrado.**
* **Precondiciones: ninguna.**
* **Request: DELETE /pet/{petId}.**
* **Datos de prueba: valor no numérico.**
* **Validaciones: 400/404; mensaje de error; sin cambios en datos.**
* **Por qué: protege al servicio de entradas erróneas.**

**API-PET-DELETE-04 — Negativo | Borrado doble (idempotencia)**

* **Objetivo: verificar respuesta al intentar borrar dos veces el mismo petId.**
* **Precondiciones: se ejecutó DELETE exitoso previamente sobre ese petId.**
* **Request: DELETE /pet/{petId} nuevamente.**
* **Datos de prueba: mismo petId.**
* **Validaciones: 404 o mensaje indicando que ya no existe; sin efectos adicionales.**
* **Por qué: confirma idempotencia/consistencia tras eliminación.**

**API-PET-DELETE-05 — Negativo | petId con caracteres especiales/espacios**

* **Objetivo: robustez ante valores no válidos en path.**
* **Precondiciones: ninguna.**
* **Request: DELETE /pet/{petId}.**
* **Datos de prueba: caracteres especiales/espacios en petId.**
* **Validaciones: 400/404; error controlado; sin stack traces.**
* **Por qué: evita fallas por encoding y entradas peligrosas.**