ChatGPT

Implementación de Login Sin Usuario (Passkeys con WebAuthn)

Introducción

La autenticación mediante **passkeys** (llaves de acceso) utilizando **WebAuthn** permitirá a los usuarios iniciar sesión sin ingresar un nombre de usuario ni contraseña. En lugar de credenciales tradicionales, se usará **autenticación FIDO2/WebAuthn** de alta seguridad, donde el navegador y un autenticador (como una huella digital, Face ID o llave de seguridad) manejan las claves criptográficas. Esto ofrece una experiencia de inicio de sesión **sin usuario y sin contraseña**, mejorando tanto la seguridad como la usabilidad. Con credenciales **residentes** (discoverable credentials), parte de la información del usuario se almacena en el autenticador, logrando un flujo de login más fluido sin pedir identificadores al usuario [1] [2]. En un flujo típico *usernameless*, el usuario simplemente abre la página de login, activa su autenticador (por ejemplo, toca su llave de seguridad o usa la biometría) y **queda autenticado en un solo paso**, sin enviar ninguna contraseña por la red [3]. Este método cumple un doble factor en un solo gesto (posesión del dispositivo + verificación biométrica o PIN), elevando la seguridad.

En el contexto de nuestro proyecto, incorporaremos este nuevo método de autenticación **como complemento** a los métodos existentes (login via NFC y QR) – dado que ya no usamos contraseñas tradicionales, las passkeys serán otra forma conveniente de autenticarse. El inicio de sesión con passkeys podrá convivir con NFC/QR sin conflicto. Los usuarios podrán elegir el método más cómodo según su dispositivo: por ejemplo, **padres y administradores** podrían usar una passkey en sus teléfonos o PCs, mientras que **estudiantes** quizás sigan usando NFC en los quioscos. A continuación, se detallará **paso a paso la implementación** en nuestro stack, incluyendo consideraciones de backend (FastAPI, SQLAlchemy, Redis), frontend (JavaScript puro en nuestras 3 apps), manejo de errores, pruebas y documentación, para asegurar que la integración de WebAuthn sea **perfecta**.

Stack y Consideraciones Generales

Antes de entrar en la implementación, resumamos el entorno técnico para contextualizar la solución:

- **Backend**: FastAPI (Python 3.11+), con SQLAlchemy Async ORM conectado a PostgreSQL. Se emplea Redis + RQ para colas de trabajos background (que también aprovecharemos para manejar desafíos WebAuthn temporalmente). La aplicación corre en **Docker** (Alpine Linux) desplegada en un VPS Ubuntu.
- **Frontend**: JavaScript plano (ES6+) sin framework SPA. Tenemos tres frontends separados:
- `src/kiosk-app/` – Aplicación de kiosco (asistencia en sitio, probablemente usada por estudiantes con NFC).
- `src/web-app/` – Panel web (admin y apoderados).
- `src/teacher-pwa/` – PWA para profesores.
- **Base de Datos**: PostgreSQL, con SQLAlchemy (asyncpg) para acceder. Usamos una URL de conexión tipo `postgresql+asyncpg://user:pass@localhost/school`. Las tablas relevantes incluyen usuarios, sesiones, datos de asistencia, preferencias, etc., a las cuales añadiremos las necesarias para WebAuthn.

- **Otros servicios**: almacenamiento en AWS S3 y notificaciones vía WhatsApp Cloud API y AWS SES, que no se ven afectados directamente por esta implementación.
- **Despliegue**: En un contenedor Docker detrás de un proxy/servidor en el VPS. Es importante asegurarse de configurar correctamente el **dominio (RP ID)** y contar con **HTTPS** en producción, ya que WebAuthn lo exige por seguridad (excepto pruebas en `localhost`). El RP ID será el dominio de la aplicación (por ejemplo, si el panel web está en `miapp.com`, ese será el RP ID).

Nota: Dado que no existe un login de usuario/contraseña convencional, las passkeys serán el primer método de autenticación *moderno* en el sistema web, eliminando la necesidad de pedir identificadores. Sin embargo, aseguraremos que si un dispositivo no soporta WebAuthn, el frontend ofrezca *fallbacks* como un código QR de acceso (o alguna ruta alternativa ya implementada) para no bloquear al usuario [4]. Nuestro objetivo es implementar todo con máxima robustez: manejo de errores claro, guías visuales para el usuario durante el proceso de autenticación, **documentación completa**, y pruebas unitarias/integrales que validen tanto el flujo de registro como de autenticación con passkeys.

Requisitos y Diseño de la Solución

Objetivos funcionales principales:

- **Registro de passkey (ceremonia de registro):** Permitir que un usuario registre una credencial WebAuthn (llave de acceso) en su cuenta. Esto puede suceder en dos escenarios:
- *Usuario existente*: Un administrador, apoderado o profesor que ya tiene cuenta (creada previamente, posiblemente asociada a un QR o NFC) quiere añadir una passkey como método de login adicional.
- *Nuevo usuario*: Alguien sin cuenta previa quiere registrarse usando sólo una passkey (*usernameless signup*). En este caso, el sistema creará la cuenta sobre la marcha tras verificar la credencial. No se le pedirá nombre de usuario ni correo para registrarse, potenciando la privacidad.
- **Autenticación (login) con passkey:** Permitir inicio de sesión mediante la credencial registrada. El flujo de login no solicita usuario previo; el sistema presentará un desafío WebAuthn y cualquier credencial **residente** registrada para este RP en el dispositivo del usuario podrá responder. El autenticador devolverá la información necesaria para que el servidor identifique al usuario sin preguntar un nombre [2].
- **Convivencia con otros métodos:** El nuevo método no reemplazará los existentes. Por tanto, en las pantallas de login ofreceremos la opción “Iniciar sesión con Passkey” junto a (o en lugar de) los botones de login vía NFC o QR según corresponda. Si WebAuthn falla o no está disponible, el usuario podrá recurrir a NFC/QR como fallback.
- **Flujo unificado registro/login:** Idealmente, simplificaremos la UX para que desde un mismo botón el usuario pueda autenticarse con passkey, ya sea registrando una nueva credencial si es la primera vez, o usando una existente si ya la tiene. (Esto puede apoyarse en la capacidad del navegador de decidir crear o usar credencial según corresponda, gracias a las credenciales residentes).
- **Seguridad**: La implementación debe seguir las recomendaciones de FIDO2:
- Usar desafíos (*challenge*) aleatorios de un solo uso para prevenir *replay attacks*.
- Verificar firmemente las respuestas WebAuthn (firmas, contadores de uso, origin, etc.).
- Exigir **verificación de usuario** (biometría/PIN) durante el login, garantizando MFA implícito (evitando autenticadores que no verifiquen al usuario).
- No almacenar información sensible en el *user handle* de WebAuthn (debe ser un identificador aleatorio opaco, sin PII [5]).
- Asegurar que todo el flujo ocurre sobre HTTPS en producción (requisito de WebAuthn; en desarrollo, `localhost` está permitido para pruebas).

- **Compatibilidad**: Soportar autenticadores tanto **plataforma** (ej: Touch ID, Windows Hello, Android Keychain) como **roaming** (YubiKeys, etc.). No restringiremos el tipo de autenticador; de hecho, no especificaremos `authenticatorAttachment` para que el usuario pueda elegir (plataforma o cross-platform). Pero sí requeriremos credenciales **descubribles** (residentes) para habilitar el login sin usuario.
- **Persistencia**: Incorporar nuevas entidades en la base de datos para guardar:
- Las credenciales WebAuthn registradas (clave pública, ID de credencial, contador, tipo, etc.) asociadas a un usuario.
- Un identificador de usuario (*user handle*) aleatorio utilizado en WebAuthn para mapear la credencial al usuario en nuestra BD.
- Posiblemente, *challenges* temporales guardados en Redis u otra estructura volátil para validar las respuestas.
- **UX/UI**: Proporcionar indicaciones en la interfaz:
- Mensajes para guiar al usuario a usar su autenticador (por ejemplo "Por favor, escanea tu huella digital o inserta tu llave de seguridad").
- Manejar casos donde el navegador no soporte WebAuthn (ocultar o deshabilitar la opción, mostrando un aviso para actualizar navegador).
- Informar al usuario si la autenticación falla (ej. "La autenticación con passkey falló, intenta de nuevo o usa un método alternativo").
- **Errores y excepciones**: Manejar todos los errores esperables:
- Usuario cancela la operación o deja expirar el tiempo -> mensaje de reintento.
- Autenticador no soporta credenciales residentes -> ofrecer método alternativo (pues algunos autenticadores antiguos fallarán al requerir credencial residente [4]).
- Fallas en la verificación de firma o desafío -> retornar error 401/400 sin revelar detalles (log interno para debug).
- Varios intentos fallidos -> potencialmente bloquear temporalmente para prevenir abuso (rate limiting).
- **Pruebas**: Incluir pruebas unitarias (de funciones de generación/verificación, manejo de DB, etc.), pruebas de integración (simulando el flujo front-back con datos de ejemplo) e incluso **pruebas end-to-end** (usando herramientas como Playwright con autenticador virtual) para validar el flujo completo en un navegador real. Se definirán criterios claros de éxito para cada prueba (expected outcomes).
- **Documentación y Manual de Usuario**: Al finalizar, proveer:
- Documentación técnica para desarrolladores (cómo está implementado, cómo configurar, agregar nuevos dispositivos, etc.).
- Un **Manual de Usuario** para administradores y usuarios finales explicando cómo usar esta función, idealmente con *capturas de pantalla* de la interfaz en funcionamiento. Esto incluirá pasos de registro de una passkey y pasos para iniciar sesión con ella, con imágenes ilustrativas de los diálogos del navegador y la aplicación.

Aclarado lo anterior, procedemos con la implementación dividida por capas (backend y frontend), seguido de testing y documentación.

Implementación en Backend (FastAPI + SQLAlchemy)

Implementaremos la lógica WebAuthn en el servidor siguiendo las especificaciones FIDO2. Utilizaremos la librería **py_webauthn** de Duo Labs (disponible como paquete `webauthn` en PyPI) para simplificar la generación y verificación de los desafíos WebAuthn. Esta librería provee funciones de alto nivel: `generate_registration_options()`, `verify_registration_response()`, `generate_authentication_options()` y `verify_authentication_response()` [6], que manejan gran parte de la complejidad criptográfica. Internamente, py_webauthn sigue la

especificación, esperando y devolviendo datos en JSON (con campos en base64 URL para los binarios), lo cual se alinea perfecto con FastAPI. Alternativamente, se podría usar la librería **python-fido2** de Yubico o una implementación manual, pero `webauthn` de Duo nos acelera el desarrollo al enfocarse en la parte servidor de WebAuthn [7]. A continuación, detallamos las piezas del backend:

Modelo de Datos para Credenciales WebAuthn

Debemos extender nuestro modelo de datos. Asumiremos que existe una tabla de **usuarios** ya definida (con campos como id, nombre, email quizás, etc.). Añadiremos:

- **user.handle (user_handle)**: un campo nuevo en la tabla de usuarios que contendrá el identificador WebAuthn (*user handle*). Este será un valor binario opaco (podemos almacenarlo como `BYTEA` en PostgreSQL, o como texto base64url en su defecto). Su longitud máxima será 64 bytes según la especificación [8]. Debe generarse aleatoriamente y **no debe contener información personal identificable** [5] (por ejemplo, no usar el email ni el ID numérico tal cual). Sugerimos generarlo como 32 bytes aleatorios por usuario (lo cual da ~43 caracteres base64url) al crear la cuenta. Este valor permanecerá constante por usuario y será el `user.id` que WebAuthn use en las opciones de registro/autenticación. Para usuarios existentes, durante la migración podríamos generar un handle para cada uno.
- **Tabla Credential** (o `user_credentials`): nueva tabla para almacenar las credenciales WebAuthn registradas. Campos principales:
- `credential_id` (texto base64url o bytes): identificador de la credencial (lo que devuelve el autenticador, normalmente 16+ bytes aleatorios [9]). Es único por credencial y la clave primaria de esta tabla.
- `user_id`: referencia/foreign key al usuario dueño de la credencial.
- `public_key`: la clave pública en formato COSE (podemos almacenarla en bytes crudos o como texto base64url). Este es extraído del objeto de atestación durante el registro.
- `sign_count`: un contador entero de uso de la credencial, provisto por el autenticador. Se actualiza en cada autenticación para prevenir reutilización de clones (si el contador disminuye o es incoherente, indica posible duplicación de credencial).
- `transports` (opcional): tipo de transportes soportados (USB, NFC, BLE, interno) si queremos almacenarlos para referencia (viene en la atestación).
- `aa_guid` o `attestation_format` (opcional): detalles de atestación si se quisiera almacenar info del dispositivo. En nuestro caso no es necesario verificar la atestación contra una CA, podemos optar por **attestation="none"** para no recopilar datos del hardware, simplificando la privacidad.
- `created_at`: timestamp de registro.

En SQLAlchemy (async) definiremos estos modelos. Un esquema simplificado podría ser:

```
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True)
    name = Column(String)
    email = Column(String)
    # ... otros campos ...
    handle = Column(LargeBinary(64), unique=True, nullable=False)  #
Identificador WebAuthn (user handle)

class Credential(Base):
```

```
    __tablename__ = "credentials"
    credential_id = Column(String, primary_key=True)  # almacenaremos en
base64url
    user_id = Column(Integer, ForeignKey("users.id"))
    public_key = Column(LargeBinary, nullable=False)  # clave pública COSE
    sign_count = Column(Integer, nullable=False)
    transports = Column(String)  # opcional, coma separada por ejemplo
    # Relationship
    user = relationship("User", back_populates="credentials")
```

Y en `User` añadir `credentials = relationship("Credential", back_populates="user")` para facilitar consultas.

Nota: El campo `handle` del usuario es crítico para login *sin username*: será el dato que nos devuelva el autenticador para decir "este es el usuario X". Debe ser aleatorio y secreto. Podemos generarlo con Python `os.urandom(32)` al crear un usuario. Alternativamente, podríamos usar directamente el UUID del usuario si fuera seguro/aleatorio, pero generalmente un `user_handle` dedicado es preferible [10]. Todas las credenciales de un mismo usuario usarán el mismo `user_handle` al registrarse, de forma que independientemente de qué credencial use en el futuro, se identifique al mismo usuario [11]. En caso de un registro *self-service* (usuario sin cuenta previa), generaremos un handle aleatorio ad-hoc, crearemos la cuenta con ese handle tras registro, y luego usaremos ese mismo handle en adelante.

Dependencias y Configuración WebAuthn

Agregaremos la dependencia `webauthn` a nuestro proyecto Python:

```
pip install webauthn
```

También nos aseguraremos de tener `fastapi` y `uvicorn` actualizados. La librería `webauthn` no maneja almacenamiento, así que necesitará que le proporcionemos ciertas cosas:

- **Relying Party (RP) ID y RP Name**: El ID suele ser el dominio (sin protocolo), por ej. `cole.mivps.com` o similar. El Name es un nombre descriptivo ("MiApp Colegio"). Definiremos esto en la configuración (por ejemplo, variables de entorno `RP_ID` y `RP_NAME`). *Importante*: para desarrollo local podemos usar `localhost` como RP ID [12], pero en producción debe coincidir con el dominio público.
- **Origin**: La URL base desde la que se permite WebAuthn. En producción, algo como `https://cole.mivps.com` (sin slash final). Durante pruebas locales, `http://localhost:8000` (nótese que en localhost se permite HTTP por ser excepción).
- **Options de seguridad**: Requeriremos credenciales residentes y verificación de usuario. Al usar py_webauthn, esto se especifica al generar las opciones:
- `resident_key_requirement="required"` para que **sí o sí** cree credenciales residentes descubribles [13].
- `user_verification="required"` (o "preferred") para que se pida PIN/biometría. Recomendamos "required" para máxima seguridad, salvo que queramos permitir autenticadores sin UV (poco probable, casi todos los passkeys requieren algún factor interno).
- Algoritmos: por defecto la librería soporta ES256, RS256, etc. Podemos dejarlo por defecto; las credenciales típicamente usarán ECDSA-SHA256 que es estándar.

- **Timeouts**: Podemos establecer un timeout (en ms) para las opciones de registro/login, p.ej. 60000 ms (60s) para dar tiempo suficiente.
- **Attestation**: Podemos poner `attestation="none"` en `generate_registration_options` a menos que tengamos un motivo para recopilar la atestación. "none" simplifica el flujo (no necesitamos validar certificados de fabricante). Esto hace que la privacidad aumente, ya que no se recolectan datos identificativos del dispositivo.

La configuración se puede cargar de `.env` usando `python-dotenv` (ya citado en el proyecto). Ejemplo de nuevos valores en `.env`:

```
RP_ID=midominio.com
RP_NAME="Colegio XYZ"
RP_ORIGIN=https://midominio.com  # incluir protocolo
```

En nuestro código, cargaremos estos:

```
RP_ID = os.getenv("RP_ID")
RP_NAME = os.getenv("RP_NAME")
RP_ORIGIN = os.getenv("RP_ORIGIN")
```

Endpoints de Registro (Register Passkey)

Implementaremos dos endpoints FastAPI para el registro de passkeys: uno para iniciar el registro y otro para finalizarlo. Suponiendo que usamos un router `/auth`:

1. **Iniciar Registro** – `POST /auth/passkey/register/start` (o `/options`):
2. **Entrada**: Puede no requerir cuerpo JSON (simplemente quien lo llame). Opcionalmente, si el usuario ya está autenticado por otro método y quiere agregar una passkey, vendría con su sesión; en caso contrario, es un nuevo registro.
3. **Proceso**:
   - Generar un nuevo *challenge* aleatorio y las opciones de registro WebAuthn.
   - Si el usuario ya está logueado (vía dependencia `get_current_user`), usaremos su `user.handle` existente y sus datos para las opciones.
   - Si **no hay usuario logueado** (registro self-service), generaremos un handle aleatorio temporal y también un identificador interno temporal. En WebAuthn necesitamos proveer un `user` en las opciones con:
   - `id`: bytes aleatorios (nuestro user handle temporal).
   - `name`: un string identificador. **Importante**: Aunque no usamos username real, la API WebAuthn exige un `user.name` y `user.displayName`. Podemos usar un placeholder, por ejemplo el mismo ID codificado o `"user_"+8hex`. El objetivo es que sea único pero no identificativo [14]. Podríamos usar el ID aleatorio codificado en base64 para name/displayName para cumplir unicidad [15]. Más adelante, si el usuario completa perfil, podríamos actualizar la credencial con un nombre descriptivo, pero eso es opcional.
   - `displayName`: igual que name (o algo más amigable si lo tuviéramos, pero en registro inicial no).
   - Configurar `authenticatorSelection` en las opciones: `resident_key=Required`, `user_verification=Required` (y si quisiéramos, `authenticator_attachment` podríamos dejarlo None para permitir cualquier).

- Llamar a `webauthn.generate_registration_options()`. Por ejemplo:

```
from webauthn import generate_registration_options,
options_to_json
options = generate_registration_options(
    rp_id=RP_ID,
    rp_name=RP_NAME,
    user_id=user_handle_bytes,
    user_name=user_name_str,
    user_display_name=user_displayname_str,
    attestation="none",
    authenticator_selection=AuthenticatorSelection(
        resident_key=ResidentKeyRequirement.REQUIRED,
        user_verification=UserVerificationRequirement.REQUIRED
    )
)
options_json = options_to_json(options)  # convierte a tipos
serializables
```

  Nota: `AuthenticatorSelection`, `ResidentKeyRequirement`, etc., vienen de `webauthn.helpers.structs` en py_webauthn.
- Guardar en **Redis** el desafío y los datos necesarios para verificar después:
- Podemos usar una clave como `"registration_chal:{challenge}"` o generar un token de sesión. Mejor: asociar al usuario temporal. Dado que en registro no-logueado no tenemos sesión aún, podríamos crear un entry en Redis mapeando el `challenge` (que es único) a los datos: `{challenge: ..., user_handle: ..., user_temp_id: ...}`. Sin embargo, **no exponer directamente el challenge como clave** por seguridad; en su lugar, podríamos generar un UUID asociado a este flujo:
  - Ej: generar `flow_id = uuid4()`, guardar en Redis `regflow:{flow_id}` -> {challenge: b'...', user_handle: b'...', existing_user_id: None/ID}.
  - Incluir ese `flow_id` en las opciones enviadas al cliente como parte de la JSON (por ejemplo, añadimos un campo no estándar `challengeId` o usamos el propio `challenge` como identificador puesto que es único). Como alternativa, usar la propia `challenge` como clave está bien siempre que usemos Redis con expiración corta, ya que es un valor aleatorio de alta entropía. De todas formas, para mantener capas separadas, preferible un `flow_id`.
- Establecer expiración corta en Redis (p.ej. 5 minutos) para ese flujo, así desafíos viejos se descartan [16].
- Devolver al frontend el JSON de opciones (challenge, rp, user, pubKeyCredParams, etc.). **Importante**: `options_to_json` nos da directamente los campos codificados en base64 listo para enviar y usar con librerías front, pero si vamos a manejar manual en JS, tendremos que convertir ciertos campos (ver sección frontend). Podemos enviar tal cual y documentar que el cliente deberá convertir `challenge` y `user.id` desde base64url a ArrayBuffer.

4. **Salida**: JSON con las opciones de registro. Ejemplo (simplificado):

```
{
  "challenge": "GXNY...GAg",
```

```
  "rp": {"name": "Colegio XYZ", "id": "midominio.com"},
  "user": {"id": "Q2tzfkhZ...==", "name": "user_XYZ", "displayName":
"user_XYZ"},
  "pubKeyCredParams": [ { "alg": -7, "type": "public-key" }, ... ],
  "authenticatorSelection": {"residentKey": "required",
"userVerification": "required"},
  "timeout": 60000,
  "attestation": "none",
  "challengeId": "550e8400-e29b-41d4-
a716-446655440000"  // si usamos un flow UUID
}
```

(El campo `challengeId` sería agregado por nosotros si usamos esa estrategia).

5. **Errores**:
    - Si el usuario está logueado y ya tiene una credencial registrada (podríamos permitir múltiples, así que no es error, puede tener varias passkeys).
    - Poco probable falle algo aquí, pero capturar excepciones de la librería `webauthn` por si faltan parámetros, y retornar 500 con mensaje.
    - Si Redis falla al guardar, retornar error 500.

Ejemplo de implementación FastAPI (simplificado):

```python
from fastapi import APIRouter, Depends, HTTPException
from webauthn import generate_registration_options, options_to_json
from webauthn.helpers.structs import AuthenticatorSelection,
ResidentKeyRequirement, UserVerificationRequirement

router = APIRouter()

@router.post("/auth/passkey/register/start")
async def start_passkey_registration(current_user: User =
Depends(get_current_user_optional)):
    # current_user_optional devuelve usuario si hay sesión, o None.
    if current_user:
        user = current_user
        user_handle = user.handle  # bytes
        user_name = f"user_{user.id}"  # o user.email, aunque la
recomendación es no PII
        display_name = user.name or user.email or f"User {user.id}"
    else:
        # Usuario no autenticado: registro nuevo
        user_handle = os.urandom(32)
        user_name = base64url_encode(user_handle)  # definimos una función
util para base64url
        display_name = user_name
        # Podríamos guardar este handle en Redis marcando que es provisional
hasta completar registro
    try:
```

```
        options = generate_registration_options(
            rp_id=RP_ID,
            rp_name=RP_NAME,
            user_id=user_handle,
            user_name=user_name,
            user_display_name=display_name,
            attestation="none",
            authenticator_selection=AuthenticatorSelection(
                resident_key=ResidentKeyRequirement.REQUIRED,
                user_verification=UserVerificationRequirement.REQUIRED
            )
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Error generando
opciones WebAuthn: {e}")
    # Guardar en Redis
    reg_data = {
        "challenge": options.challenge,
        "user_handle": user_handle,
        "existing_user_id": user.id if current_user else None
    }
    flow_id = str(uuid.uuid4())
    redis.set(f"regflow:{flow_id}", pickle.dumps(reg_data), ex=300)  #
expiración 5 min
    # Convertir a JSON serializable
    opts_json = options_to_json(options)
    opts_json["challengeId"] = flow_id
    return opts_json
```

Nota: `options.challenge` es bytes; `options_to_json` lo convierte a base64url en string dentro de la estructura retornada.

1. **Finalizar Registro** – `POST /auth/passkey/register/finish`:
2. **Entrada**: JSON con la respuesta del cliente a la creación de credencial. Esto incluirá:
    - `id`: ID de credencial (texto base64url).
    - `rawId`: ID de credencial en base64url (usualmente igual a id pero en formato de bytes).
    - `type`: "public-key".
    - `response`: objeto con:
    - `attestationObject`: base64url del objeto de atestación (bytes).
    - `clientDataJSON`: base64url del clientData (bytes JSON con el challenge firmado).
    - `transports` (posiblemente, si el navegador los envía).
    - `clientExtensionResults` (posiblemente vacío).
    - `challengeId` (si incluimos ese campo para correlacionar, o podríamos inferirlo del challenge en clientDataJSON).
3. **Proceso**:
    - Recuperar de Redis los datos guardados usando `challengeId` (o si no usamos ID explícito, podríamos extraer el challenge del `clientDataJSON` decodificado para clave de búsqueda, pero es más complejo porque el challenge se codificó y firmó). Usaremos el `challengeId` para facilidad.

- Si no encontramos datos (expiró o no existe): retornar HTTP 400 "El registro ha expirado, intente de nuevo".
- Si encontramos:
- Extraer `expected_challenge` (bytes) y otros datos (user_handle, existing_user_id).
- Llamar a `webauthn.verify_registration_response()`. Esta función verificará la firma de atestación, la integridad de clientDataJSON, etc. Necesita parámetros:
    - `credential: dict` – el objeto credential del cliente (como dict). py_webauthn acepta directamente el dict JSON que enviamos.
    - `expected_challenge`, `expected_origin` (nuestro RP_ORIGIN), `expected_rp_id` (RP_ID).
    - `require_user_verification` (True, ya que pedimos UV).
- La función devuelve un objeto con los datos verificados: `.verified` (bool), y si success, contiene `attestation_info` con la `credential_public_key`, `credential_id` y `sign_count`.
- Evaluar `verified`. Si False, retornar 400 "Datos de registro inválidos".
- Si True:
    - Si `existing_user_id` estaba en los datos:
    - **Usuario existente**: usar ese id para asociar la credencial.
    - Si era None (nuevo registro):
    - Crear un nuevo registro de usuario en la BD:
        - Asignar un ID (se generará auto incremental o según la BD).
        - Asignarle el `user_handle` que habíamos generado (lo tenemos de Redis).
        - Puede que no tengamos nombre/email todavía. Podemos por ahora poner campos vacíos o el displayName provisional. Idealmente, podríamos inmediatamente solicitar al usuario que llene su perfil tras login. En el manual de usuario indicaremos esto.
        - Guardar usuario (flush para obtener su ID).
    - Crear la entrada en tabla `Credential`:
    - `credential_id` = (credential.id del JSON, que es base64url de la clave pública ID). Podemos usar directamente el string del JSON como PK.
    - `user_id` = id del usuario asociado.
    - `public_key` = attestation_info.credential_public_key (py_webauthn nos puede darlo en COSE bytes). Es importante almacenar la clave para futuras verificaciones.
    - `sign_count` = attestation_info.sign_count or 0.
    - `transports` = attestation_info.transports (si disponible).
    - Guardar en BD (commit).
    - (Opcional) Iniciar sesión: Ya que estamos en endpoint de registro, podríamos automáticamente generar una sesión/login para el usuario recién registrado o para el existente. Esto mejora la UX: tras registrar la llave, el usuario queda logueado. Para ello:
    - Crear una sesión (si hay un mecanismo de sesión JWT o cookie). En nuestro proyecto, posiblemente tengamos un sistema de sesión en BD con token. Podemos emitir un token JWT o crear un registro en tabla de sesiones y setear cookie.
    - Devolver HTTP 200 con un mensaje de éxito o con datos del usuario/token. FastAPI puede usar `Response.set_cookie` si usamos cookies de sesión, o simplemente retornar el JWT.
    - Borrar el registro temporal de Redis (ya no se necesita).
- **Errores**:
- Datos faltantes en la solicitud (400).

- Challenge no coincide (lo maneja verify_registration_response si se le pasa mal expected_challenge).
- Excepción de la librería (400/500).
- Si el usuario ya tenía una credencial con el mismo ID (colisión muy improbable a menos que mismo dispositivo registrado dos veces, en tal caso Yubikey normalmente retornaría mismo credential_id; podríamos decidir que no duplique. La BD lo rechazará por PK duplicada, manejar esa excepción para informar "Credencial ya registrada").

Ejemplo de implementación:

```
from webauthn import verify_registration_response
from webauthn.helpers.structs import RegistrationCredential

@router.post("/auth/passkey/register/finish")
async def finish_passkey_registration(response: dict):
    # 'response' es el JSON enviado por el cliente con la credencial
    try:
        flow_id = response.get("challengeId")
        if not flow_id:
            raise HTTPException(status_code=400, detail="Falta challengeId")
        data_raw = redis.get(f"regflow:{flow_id}")
        if not data_raw:
            raise HTTPException(status_code=400, detail="Registro expirado o
inválido")
        reg_data = pickle.loads(data_raw)
        expected_challenge = reg_data["challenge"]
        user_handle = reg_data["user_handle"]
        existing_user_id = reg_data["existing_user_id"]
        # Verificación de la respuesta de registro
        verification = verify_registration_response(
            credential=response,  # dict con id, rawId, response, type
            expected_challenge=expected_challenge,
            expected_rp_id=RP_ID,
            expected_origin=RP_ORIGIN,
            require_user_verification=True
        )
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error verificando
respuesta: {e}")
    if not verification.verified:
        raise HTTPException(status_code=400,
detail="Fallo en verificación de la attestation")
    # Si llegó aquí, registro válido
    # Obtener o crear usuario
    async with async_session() as session:
        if existing_user_id:
            user = await session.get(User, existing_user_id)
            if not user:
                raise HTTPException(status_code=500,
detail="Usuario esperado no encontrado")
        else:
```

```
            # Crear nuevo usuario
            user = User(handle=user_handle)
            session.add(user)
            await session.flush()  # para obtener user.id
        # Almacenar credencial
        cred_id_str = response["id"]  # ID en base64url
        credential = Credential(
            credential_id=cred_id_str,
            user_id=user.id,
            public_key=verification.attestation_info.credential_public_key,
            sign_count=verification.attestation_info.sign_count or 0
        )
        session.add(credential)
        try:
            await session.commit()
        except IntegrityError:
            await session.rollback()
            raise HTTPException(status_code=400, detail="Esta credencial ya
está registrada.")
    # Limpieza
    redis.delete(f"regflow:{flow_id}")
    # Iniciar sesión (crear token/cookie). Supongamos uso de JWT:
    token = create_jwt_for_user(user.id)
    return {"message": "Passkey registrada exitosamente", "token": token}
```

Donde `create_jwt_for_user` es una función hipotética que genera un JWT firmado con el `user.id` (o podríamos setear una cookie de sesión aquí).

Esta ruta ahora permite tanto a usuarios existentes vincular una nueva credencial (no cierra la sesión actual, solo agrega método), como a nuevos usuarios registrarse y quedar logueados.

Endpoints de Autenticación (Login Passkey)

De forma similar, creamos endpoints para el flujo de login WebAuthn:

1. **Iniciar Autenticación** – `POST /auth/passkey/authenticate/start`:
2. **Entrada**: Idealmente nada, o podríamos aceptar algún indicador (pero en usernameless no se provee identificación).
3. **Proceso**:
    - Generar un *challenge* nuevo para login.
    - Preparar opciones de autenticación con `generate_authentication_options()` de py_webauthn.
    - Pasar `rp_id=RP_ID`.
    - Como no tenemos un usuario específico (usernameless), **no proporcionamos** `allowed_credentials` (o ponemos lista vacía). Esto indica al cliente que puede usar **cualquier credencial residente válida para este RP** [17]. El navegador entonces permitirá que el usuario seleccione la credencial deseada si tiene varias, o use la única sin preguntar.
    - `user_verification="required"` (queremos que pida biometría/PIN).
    - Establecer `timeout` como antes.

- *PublicKeyCredentialRequestOptions* resultante contendrá challenge, rpId, etc.
- Guardar en Redis el `challenge` junto con quizás un ID de flujo:
- Similar al registro, generar `flow_id = uuid4()` y guardar `authflow:{flow_id}` -> {challenge: ..., maybe timestamp}.
- (También podríamos almacenar `allowed_credentials` si usáramos, pero aquí no hay).
- Retornar al frontend las opciones de autenticación (JSON). Ej:

```
{
  "challenge": "BASE64URL...",
  "rpId": "midominio.com",
  "allowCredentials": [],   // vacío o ausencia significa
usernameless
  "userVerification": "required",
  "timeout": 60000,
  "challengeId": "...uuid..."
}
```

4. **Errores**: Mismas consideraciones de generación (poco probables). Si falla generar opciones, 500.

Ejemplo:

```
@router.post("/auth/passkey/authenticate/start")
async def start_passkey_authentication():
    try:
        options = generate_authentication_options(
            rp_id=RP_ID,
            user_verification="required"
            # Si quisiéramos permitir credencial no residente (username
input),
            # aquí iría allowed_credentials, pero dejamos vacío.
        )
    except Exception as e:
        raise HTTPException(status_code=500,
detail=f"Error generando reto de autenticación: {e}")
    flow_id = str(uuid.uuid4())
    data = {"challenge": options.challenge}
    redis.set(f"authflow:{flow_id}", pickle.dumps(data), ex=300)
    opts_json = options_to_json(options)
    opts_json["challengeId"] = flow_id
    return opts_json
```

Aquí `options_to_json` convertirá challenge a base64 etc. Si no proporciona `allowed_credentials`, en el JSON podría ni aparecer esa clave. Podemos añadir manualmente `allowCredentials: []` para que el cliente sepa que está vacío (algunos navegadores requieren la clave presente pero vacía para activar modo discoverable; según la spec, si no se provee significa no restringido, lo cual es equivalente a lista vacía).

1. **Finalizar Autenticación** – `POST /auth/passkey/authenticate/finish`:

2. **Entrada**: JSON con la respuesta de `navigator.credentials.get()` del frontend. Contendrá:
    - `id`, `rawId` (ID de credencial usada, en base64url).
    - `type` ("public-key").
    - `response`:
    - `authenticatorData`: base64url (bytes con flags, signCount, etc.).
    - `clientDataJSON`: base64url.
    - `signature`: base64url (firma sobre clientData + authData).
    - `userHandle`: base64url (aquí vendrá el `user_handle` en bytes que almacenamos en el autenticador si es credencial residente).
    - `clientExtensionResults` (si hay).
    - `challengeId` (nuestro ID para correlacionar).
3. **Proceso**:
    - Recuperar de Redis el challenge usando `challengeId`. Si no, error 400 (expirado).
    - Obtener `expected_challenge` (bytes).
    - Con el `credential.id` recibido (identificador de credencial), buscar en la base de datos la credencial correspondiente:
    - Obtener su `public_key` y `sign_count` almacenados.
    - También podemos identificar al usuario asociado a esa credencial aquí.
    - Llamar a `verify_authentication_response()` de py_webauthn con:
    - `credential` (el dict de respuesta),
    - `expected_challenge`, `expected_rp_id`, `expected_origin`,
    - `credential_public_key` (bytes de la pubkey de BD),
    - `credential_current_sign_count` (entero de BD),
    - `require_user_verification=True`.
    - Esta función verificará la firma usando la clave pública dada, comparará el challenge y origin, y validará (y actualizará internamente) el contador de firma.
    - Retorna un objeto con `.verified` y `authentication_info`. Este último incluye `new_sign_count`.
    - Si `verified` es False: retornar 401 "Autenticación fallida".
    - Si True:
    - Actualizar en BD la credencial: set `sign_count = authentication_info.new_sign_count` (importante para detectar duplicados en el futuro).
    - **Identificar al usuario**:
        - Podemos usar el `userHandle` proporcionado en la respuesta. Este debería ser el `user.handle` bytes que pusimos al registrar. Sin embargo, dado que ya buscamos la credencial por ID, ya tenemos `user_id`. Ambas formas son posibles:
        - *Vía credencial*: credential->user_id->usuario.
        - *Vía userHandle*: convertir de base64 a bytes y buscar usuario con ese handle (debe haber uno único). Este enfoque *verifica* adicionalmente consistencia: podríamos asegurar que el userHandle que devolvió coincide con el handle del usuario de la credencial. Por seguridad, podríamos comparar ambos (de BD) y si difieren, es anómalo.
        - Procederemos con `user = credential.user` vía ORM para simplicidad.
    - Crear la sesión de login para ese usuario:
        - Igual que antes, generar JWT o cookie.
        - Si JWT, retornarlo en la respuesta JSON. Si cookie, setear en Response.
    - Retornar 200 OK con un mensaje o token.
    - Borrar el registro de Redis de ese challenge.

4. **Errores**:
    - Credencial no hallada en BD: posible si el usuario usa una credencial de otro sistema o no registrada. En tal caso, retornamos 401 "Credencial no reconocida". (Esto en teoría no debería ocurrir porque para que se active, la credencial debe pertenecer a este RP, pero es posible si la BD se perdió. Mejor manejarlo).
    - Verificación falla (firma incorrecta, challenge mismatch, etc.): 401.
    - Si la `authentication_info.new_sign_count` es menor o igual al almacenado (significa posible clon): se puede invalidar la credencial y alertar. Esto es un edge case de seguridad. Podríamos en ese caso eliminar la credencial y pedir registro de nuevo. Pero inicialmente podemos simplemente loguear una advertencia.
    - Excepciones varias: 500.

Ejemplo de implementación:

```
from webauthn import verify_authentication_response

@router.post("/auth/passkey/authenticate/finish")
async def finish_passkey_authentication(response: dict):
    flow_id = response.get("challengeId")
    if not flow_id:
        raise HTTPException(status_code=400, detail="Falta challengeId")
    data_raw = redis.get(f"authflow:{flow_id}")
    if not data_raw:
        raise HTTPException(status_code=400,
detail="Autenticación expirada o inválida")
    auth_data = pickle.loads(data_raw)
    expected_challenge = auth_data["challenge"]
    cred_id = response.get("id")
    if not cred_id:
        raise HTTPException(status_code=400, detail="Respuesta de credencial
incompleta")
    # Buscar credencial en BD
    async with async_session() as session:
        credential = await session.get(Credential, cred_id)
        if not credential:
            # Credencial no registrada en nuestro sistema
            raise HTTPException(status_code=401, detail="Credencial
desconocida")
        user = await session.get(User, credential.user_id)
    try:
        verification = verify_authentication_response(
            credential=response,
            expected_challenge=expected_challenge,
            expected_rp_id=RP_ID,
            expected_origin=RP_ORIGIN,
            credential_public_key=credential.public_key,
            credential_current_sign_count=credential.sign_count,
            require_user_verification=True
        )
    except Exception as e:
        raise HTTPException(status_code=400, detail=f"Error verificando
```

```
autenticación: {e}")
    if not verification.verified:
        raise HTTPException(status_code=401, detail="Autenticación WebAuthn
no válida")
    # Verificación exitosa
    # Actualizar sign_count
    new_sign_count = verification.authentication_info.new_sign_count
    try:
        async with async_session() as session:
            cred = await session.get(Credential, cred_id)
            cred.sign_count = new_sign_count
            await session.commit()
    except Exception as e:
        # No es crítico si no se puede actualizar inmediatamente, se puede
loguear.
        print("WARN: no se pudo actualizar sign_count:", e)
    # Iniciar sesión para el usuario
    token = create_jwt_for_user(user.id)
    redis.delete(f"authflow:{flow_id}")
    return {"message": "Autenticación exitosa", "token": token}
```

Tras esto, el frontend recibiría el token y podría usarlo (o si manejamos cookies de sesión, habríamos seteado la cookie en la respuesta HTTP).

Manejo de Errores y Fallbacks en Backend

Algunos casos especiales y cómo los manejaremos:

- **Autenticador sin soporte de resident keys**: Si el usuario intenta registrar en un dispositivo antiguo (por ejemplo, una Yubikey U2F muy vieja o ciertos navegadores), al llamar a `navigator.credentials.create` con `residentKey: "required"`, **el navegador lanzará un error** (NotSupportedError) indicando que no se pudo crear credencial [4]. Este error en realidad ocurre en frontend antes de llegar al servidor. No obstante, en backend podemos detectar intentos repetidos fallidos. La recomendación de UX es: si detectamos este error en frontend, informar al usuario "Tu autenticador no soporta passkeys. Intenta con otro dispositivo o método" y quizás ofrecer un *fallback* (por ejemplo, permitir un registro *con username* solo en ese caso, aunque eso reintroduce username... Alternativamente, el fallback es usar login via QR/NFC). Dado que no implementaremos credencial no-residente en este sistema (para mantenerlo passwordless total), simplemente documentaremos este caso y aconsejaremos al usuario usar otro método. En backend, si por algún motivo recibimos una respuesta sin `userHandle` (lo que indicaría credencial no residente, que no esperamos al requerir required), podríamos rechazarla por política.
- **Challenge expirado**: Ya cubierto, devolvemos 400 y en frontend reiniciamos el flujo.
- **Falla de verificación de firma**: 401 no autorizado. Esto podría significar firma alterada o credencial no válida.
- **Credential cloned (sign count)**: Si el contador de la credencial recibido es menor que el almacenado, indica posible clonación (alguien duplicó la llave privada). En ese caso, `verify_authentication_response` suele marcar `verified=False` automáticamente si `require_user_verification=True` y clon (porque el flag UV or sign count check falharía). Pero si no, podemos manualmente comprobarlo. Si ocurre, podríamos invalidar esa credencial

(borrarla de BD) y forzar re-registro. Por simplicidad, loguearemos una advertencia y actualizaremos el contador de todos modos para evitar bloqueos posteriores.
- **Intentos múltiples**: Podríamos limitar via Redis, p. ej., contando intentos fallidos por IP o por credencial, para evitar fuerza bruta. Sin password de por medio, es difícil hacer fuerza bruta ya que se necesita la llave física para generar firmas; por lo tanto, los ataques de *online guessing* no aplican, pero podríamos tener un mal usuario haciendo spam al endpoint. Podemos aplicar rate limit a `/authenticate/start` y `/register/start` a X por minuto por IP.
- **Excepciones internas**: Cualquier excepción no manejada, devolvemos 500 genérico para no filtrar info sensible.

Consideraciones de seguridad adicionales:

- **HTTPS**: Asegurar que el despliegue esté bajo TLS. WebAuthn falla si la página no es segura. En nuestro VPS con Docker, probablemente esté detrás de un Nginx con SSL. Verificar configuraciones de *CORS* y cookies seguras si aplican.
- **Origin**: Debemos configurar correctamente `RP_ID` y `RP_ORIGIN`. Si el panel web es accesible tanto por nombre de dominio como por IP, el RP_ID debe ser exactamente el dominio que salga en la barra del navegador. No usar comodines.
- **Reproducción de challenges**: Los challenges son de un solo uso. Nos aseguramos de quitarlos de Redis tras uso, y la librería verifica que el challenge en `clientDataJSON` coincida.
- **Algoritmos**: Aceptamos ECDSA (alg -7) y tal vez RSA (alg -257) que son por defecto. Esto cubre prácticamente todos los autenticadores [12].
- **Tamaño de clave**: No restringimos, pero típicamente ES256.
- **Protección de replays**: Gracias al sign_count y challenge único, estamos cubiertos.
- **Sesiones**: Tras login, mantenemos la sesión activa según las políticas ya existentes (p. ej., expirar al cabo de X horas o con JWT exp).
- **Eliminar credenciales**: Deberíamos también implementar funcionalidad para que un usuario o admin pueda **remover una passkey registrada** (p. ej., perdió su dispositivo). Esto sería otro endpoint (DELETE /auth/passkey, quizás). Lo mencionamos para completitud aunque no fue requerido explícitamente. En documentación anotaremos que la eliminación es posible manualmente removiendo el registro en BD si fuera necesario, hasta implementar endpoint.

En este punto, el backend soporta el registro y autenticación vía passkeys de manera segura y alineada con las mejores prácticas. Hemos marcado *Resident Keys* = *Required* y *User Verification* = *Required* en la configuración [12] para asegurar credenciales descubiertas y fuertemente verificadas. Estas opciones se basan en recomendaciones comunes en implementaciones de passkeys [12].

Por último, antes de pasar al frontend, es importante recalcar que nuestra implementación es *stateless* respecto a la credencial: cualquier dispositivo con una credencial registrada para nuestro RP podrá iniciar sesión sin username. **¿Cómo identifica el servidor al usuario correcto?** Por el *user handle* devuelto en la autenticación, que coincide con el almacenado en la cuenta [18]. Al registrar, guardamos esa asociación *user handle -> user id*; al autenticar, cuando `navigator.credentials.get` entrega `userHandle`, el servidor la usa para determinar la cuenta y concluir quién inició sesión [18]. Todo esto ocurre automáticamente con credenciales residentes, cumpliendo el objetivo de no pedir nombre de usuario.

Implementación en Frontend (Vanilla JavaScript)

Ahora abordamos los cambios necesarios en nuestros frontends JavaScript sin framework. Tendremos que integrar el uso de la **Web Authentication API** del navegador (`navigator.credentials`) en las aplicaciones web pertinentes.

Dado que tenemos tres aplicaciones separadas, analizaremos cada una brevemente:

- **Kiosk App (kiosk-app)**: Esta app probablemente corre en dispositivos kiosco fijos para registrar asistencia con NFC, posiblemente sin un proceso de login tradicional por usuario, o con un login de administrador para configurar. Si no hay un formulario de login de usuario en kiosco (porque los estudiantes solo escanean tarjetas NFC), quizá **no necesitemos integrar passkeys en el kiosk**. A menos que quisiéramos que un administrador desbloquee el kiosco con su passkey, pero no parece el caso. Por ahora, podemos dejar el kiosk sin cambios relativos a passkeys.
- **Web Admin/Parent Panel (web-app)**: Aquí seguramente sí existe una pantalla de login para administradores y/o apoderados. Actualmente quizás se usa un método especial (¿login por QR? no está claro). Introduciremos una opción clara de "Iniciar sesión con Passkey". También posibilitaremos que, una vez logueados, puedan registrar una passkey en su perfil (para uso futuro).
- **Teacher PWA (teacher-pwa)**: Los profesores al abrir la PWA en sus móviles es muy posible que necesiten autenticarse. Si hasta ahora no tenían método de contraseña, tal vez se les generaba un QR o token para ingreso. Passkeys son ideales para profesores porque pueden usar la biometría del teléfono. Integraremos similar al panel web: opción de login con passkey y registro de passkey.

En resumen, enfocaremos la implementación en **pantallas de autenticación** (login) y en alguna sección de **perfil/ajustes** para registro de nuevas passkeys.

Detección de soporte y UI adaptativa

Primero, debemos detectar si el navegador soporta WebAuthn (todas las versiones modernas de Chrome, Firefox, Safari, Edge lo soportan a 2025). La API base es `window.PublicKeyCredential`. En nuestros scripts de frontend, haremos:

```
if (!window.PublicKeyCredential) {
    // Navegador no soporta WebAuthn (muy raro en 2025, pero IE u otros).
    // Ocultar o deshabilitar la opción de passkey login:
    document.getElementById('passkeyLoginBtn').style.display = 'none';
    // O mostrar un mensaje: "Tu navegador no soporta passkeys. Actualiza o
usa otro método."
}
```

Esto se hace al cargar la página de login. Asimismo, podríamos inspeccionar capacidades: `PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable()` devuelve una Promise que indica si hay algún autenticador de plataforma disponible (por ej, si el dispositivo tiene biométrico configurado). Pero para mostrar el botón no es imprescindible; con la simple detección de API es suficiente, ya que incluso si no hay autenticador, al hacer click el navegador ofrecerá opciones (por ejemplo, conectar una llave USB).

Flujo de Login con Passkey (frontend)

En la página de login del panel web y PWA, agregaremos un botón o enlace: **"Iniciar sesión con Passkey"**. En HTML puede ser un botón con id `passkeyLoginBtn`.

Cuando el usuario pulsa ese botón: 1. Llamamos vía `fetch` al endpoint backend `/auth/passkey/authenticate/start` para obtener las opciones de autenticación. Esto será una petición POST (posiblemente sin cuerpo).

```
const resp = await fetch('/auth/passkey/authenticate/start', { method:
'POST' });
if (!resp.ok) { alert("Error iniciando autenticación WebAuthn"); return; }
const options = await resp.json();
```

2. Recibiremos un JSON con `challenge` (en base64url), `rpId`, `timeout`, `userVerification` y tal vez `allowCredentials` vacío, más nuestro `challengeId`. Antes de invocar la API WebAuthn, debemos transformar algunos campos: - `challenge` y cualquier otro campo que deba ser ArrayBuffer: * `challenge` viene como string base64url; convertirlo a ArrayBuffer. * `allowCredentials`: en este caso vacío, no hay que convertir nada. (Si hubiese, cada cred.id base64 -> ArrayBuffer). * `userVerification` es string "required" que se puede dejar igual o cambiar a `'required'` (string) ya está correcto. - Utilidades para base64url <-> ArrayBuffer: Podemos implementar dos funciones JavaScript:

```
function base64urlToArrayBuffer(base64url) {
    // Añadir padding '=' si falta
    let base64 = base64url.replace(/-/g, '+').replace(/_/g, '/');
    const pad = base64.length % 4;
    if (pad) base64 += '='.repeat(4 - pad);
    const binary = atob(base64);
    const len = binary.length;
    const bytes = new Uint8Array(len);
    for (let i = 0; i < len; i++) {
        bytes[i] = binary.charCodeAt(i);
    }
    return bytes.buffer;
}
function arrayBufferToBase64url(buffer) {
    const bytes = new Uint8Array(buffer);
    let binary = "";
    bytes.forEach(b => binary += String.fromCharCode(b));
    let base64 = btoa(binary);
    base64 = base64.replace(/\+/g, '-').replace(/\//g, '_').replace(/=+$/,
'');
    return base64;
}
```

- Aplicamos:

```
options.challenge = base64urlToArrayBuffer(options.challenge);
// Si existiera options.allowCredentials:
// options.allowCredentials.forEach(c => c.id =
base64urlToArrayBuffer(c.id));
```

- Ahora `options` está listo para la API. 3. Llamar a `navigator.credentials.get({ publicKey: options })`. Esto abrirá el diálogo de autenticación en el navegador. El navegador buscará una credencial para `rpId`. Si solo hay una, posiblemente simplemente pida biometría; si hay varias, mostrará una lista al usuario para elegir cuenta. *(Por ejemplo, en Chrome si el usuario tiene 2 passkeys almacenadas para este sitio - quizá dos cuentas distintas - le aparecerá una ventana para seleccionar cuál usar* [19] *.)*

Mientras tanto, podemos mostrar en la UI un mensaje tipo "Esperando autenticación..." o un spinner, ya que el usuario tiene que interactuar con el dispositivo. Por ejemplo, deshabilitar el botón para que no puedan clickear de nuevo.

Cuando la Promise de `navigator.credentials.get` se resuelva: - Si el usuario autenticó satisfactoriamente, recibiremos un objeto `PublicKeyCredential`. - Si falla (usuario canceló o timeout), se lanzará una excepción. Debemos capturarla. - Código:

```
let credential;
try {
    credential = await navigator.credentials.get({ publicKey: options });
} catch (err) {
    if (err.name === 'NotAllowedError') {
        // El usuario canceló o expiró el tiempo
        alert("Autenticación cancelada o expirada. Inténtalo de nuevo.");
        // Re-habilitar botón, etc.
        return;
    } else {
        console.error("WebAuthn error", err);
        alert("Error en autenticación: " + err.message);
        return;
    }
}
```

4. Formatear los datos de `credential` para enviar al servidor: - Un objeto `PublicKeyCredential` tiene propiedades: - `id` (string base64url de cred ID). - `rawId` (ArrayBuffer del ID). - `response` (AuthAssertion, con ArrayBuffers). - `type`. - Preparar JSON:

```
const clientDataJSON =
arrayBufferToBase64url(credential.response.clientDataJSON);
const authenticatorData =
arrayBufferToBase64url(credential.response.authenticatorData);
const signature = arrayBufferToBase64url(credential.response.signature);
const userHandle = credential.response.userHandle
                   ? arrayBufferToBase64url(credential.response.userHandle)
                   : null;
const authPayload = {
    id: credential.id,
    rawId: arrayBufferToBase64url(credential.rawId),
    type: credential.type,
    response: {
        clientDataJSON,
```

```
        authenticatorData,
        signature,
        userHandle
    },
    clientExtensionResults: credential.getClientExtensionResults(),
    challengeId: options.challengeId  // incluir nuestro flow id
};
```

Aquí `userHandle` puede ser null si la credencial no era residente, pero en nuestro caso siempre debe venir (al menos en Chrome y otros, `userHandle` debería estar presente con el bytes del user id registrado [20]). 5. Enviar este JSON al endpoint `/auth/passkey/authenticate/finish` vía fetch:

```
const verifyResp = await fetch('/auth/passkey/authenticate/finish', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(authPayload)
});
if (!verifyResp.ok) {
    const err = await verifyResp.json();
    alert("Error: " + err.detail || "Falló la autenticación.");
    return;
}
const data = await verifyResp.json();
// Si usamos JWT:
const token = data.token;
// Guardar token en localStorage o cookie, e.j:
localStorage.setItem('authToken', token);
// Redirigir a la página principal logueada:
window.location.href = "/dashboard.html";
```

Si nuestro backend setea una cookie de sesión en la respuesta, en lugar de procesar JSON, usaríamos `credentials: 'include'` en fetch para que la cookie se guarde, y simplemente comprobar el status.

- Errores posibles:
  - 400/401 devuelto: mostramos mensaje. Por ejemplo credencial no reconocida.
  - Network error: alertar "No se pudo conectar, verifica tu conexión."
- Una vez logueado con éxito, el usuario estará en la aplicación (dashboard). Podemos mostrar un mensaje de bienvenida etc. Si era un nuevo registro completado en el mismo flujo (caso de registro implícito), también estaría logueado ya.

UX Considerations: La primera vez, si el usuario es *nuevo* y acaba de crearse la cuenta via passkey, quizás querramos detectar eso para guiarlo a completar su perfil (por ej, ingresar nombre, email). Podríamos pasar un indicador en la respuesta de login. Esto es más de flujo de negocio, pero mencionarlo: en `finish_passkey_registration` podríamos devolver algo como `{"newUser": true, "token": "...", "userId": 123}` para que el frontend sepa que es recién creado y redirija a una página "Complete su registro". Esto se documentará en el manual.

Procedamos ahora con el **registro de credencial (passkey) desde el frontend** para usuarios logueados que quieran añadir otra:

Flujo de Registro de Passkey (frontend)

En la interfaz de perfil o ajustes del usuario (tanto en panel web como PWA), agregaremos una sección **"Registrar nueva Passkey"** donde, estando logueado, puedan añadir otra credencial (por ejemplo, un usuario quiere registrar tanto la huella del laptop como una llave USB). Debemos:

- Proveer un botón "Agregar Passkey" (`registerPasskeyBtn`).
- Al hacer click, proceso similar:
- Llamar a `POST /auth/passkey/register/start` (esta vez el usuario está autenticado, el endpoint usará su user_handle existente).
- Recibir opciones de registro (challenge, user, etc). Convertir `challenge` (y `user.id`) de base64 a ArrayBuffer.
    - **Nota**: Con py_webauthn, `options_to_json` habrá enviado `user.id` como base64. Debemos convertirlo:

      ```
      options.challenge = base64urlToArrayBuffer(options.challenge);
      options.user.id = base64urlToArrayBuffer(options.user.id);
      ```

      (No tocamos user.name/displayName).
- Llamar `navigator.credentials.create({ publicKey: options })`.
    - Esto abrirá el diálogo de crear credencial en el navegador. El usuario verá algo como *"Crear llave de acceso para Sitio: midominio.com"* y posiblemente se le pida autenticar (PIN/biometría) para atestación. Por ejemplo, **Chrome** pedirá tocar la llave o lector [21], y quizás confirmar la creación de la credencial. El texto dirá que se almacenará un registro de la visita en el autenticador (eso es el residuo de la credencial residente) [21]. *(Ver imagen de ejemplo abajo donde Chrome solicita tocar la llave de seguridad para crear la passkey: la notificación menciona que se guardará un registro en la llave.)*
- Capturar resultados:
    - Si el usuario canceló (NotAllowedError): notificar "Registro cancelado".
    - Si éxito, tendremos un `PublicKeyCredential` con:
    - `id` (string base64 de cred ID).
    - `rawId` (ArrayBuffer).
    - `response.attestationObject` (ArrayBuffer), `response.clientDataJSON` (ArrayBuffer).
    - Formar JSON similar a antes:

      ```
      const attestationObject =
      arrayBufferToBase64url(credential.response.attestationObject);
      const clientDataJSON =
      arrayBufferToBase64url(credential.response.clientDataJSON);
      const rawId = arrayBufferToBase64url(credential.rawId);
      const regPayload = {
          id: credential.id,
          rawId,
          type: credential.type,
          response: { attestationObject, clientDataJSON },
          clientExtensionResults:
      ```

```
    credential.getClientExtensionResults(),
        challengeId: options.challengeId
};
```

(No hay `userHandle` en este caso; la info de usuario va dentro de attestation).

- Enviar via fetch a `/auth/passkey/register/finish`.
    - Si devuelve ok:
    - Mostrar "Passkey registrada con éxito." En caso de que no inicie sesión nuevo (aquí ya estaba logueado), quizás actualizar una lista de credenciales en la UI si la tenemos.
    - Puede que el backend no devuelva nada útil aparte de 200. Podemos simplemente notificar.
    - Si error:
    - Ejemplo: 400 si la verificación falló. Mostrar "No se pudo registrar la passkey."
    - Si 400 "Credencial ya registrada": indicar "Esta llave ya estaba registrada a tu cuenta."
- (Opcional) Actualizar UI: por ejemplo, si tenemos una lista de métodos de 2FA, agregar "Passkey - registrada el {fecha}". Esto requeriría un endpoint para listar credenciales o incluir en la respuesta.

Interfaz de Usuario durante registro: Debemos guiar al usuario: - Al hacer click en "Agregar Passkey", mostrar un modal o texto "Por favor, sigue las instrucciones del navegador para registrar tu nueva llave de acceso". - **Ejemplo de indicación**: "Si usas un dispositivo con sensor biométrico, es posible que aparezca una ventana pidiéndote que uses tu huella digital o rostro. Si usas una llave de seguridad física, insértala y tócala cuando se te indique."
(Imagen de referencia: el navegador mostrando "Toque su llave de seguridad") - Tras completar, un mensaje de éxito.

Ejemplo de código JavaScript para registro:

```
document.getElementById('registerPasskeyBtn').addEventListener('click',
async () => {
    if (!window.PublicKeyCredential) {
        alert("Passkeys no soportadas en este navegador.");
        return;
    }
    try {
        const resp = await fetch('/auth/passkey/register/start', { method:
'POST' });
        if (!resp.ok) {
            let err = await resp.json();
            throw new Error(err.detail || "Error iniciando registro");
        }
        const options = await resp.json();
        // Convertir challenge y user.id a ArrayBuffer
        options.challenge = base64urlToArrayBuffer(options.challenge);
        options.user.id = base64urlToArrayBuffer(options.user.id);
        // Llamar WebAuthn create
        const credential = await navigator.credentials.create({ publicKey:
```

```
options });
        if (!credential) {
            throw new Error("Credencial no creada.");
        }
        // Preparar datos para enviar
        const attObj =
arrayBufferToBase64url(credential.response.attestationObject);
        const clientDataJSON =
arrayBufferToBase64url(credential.response.clientDataJSON);
        const rawId = arrayBufferToBase64url(credential.rawId);
        const regPayload = {
            id: credential.id,
            rawId: rawId,
            type: credential.type,
            response: { attestationObject: attObj, clientDataJSON:
clientDataJSON },
            clientExtensionResults: credential.getClientExtensionResults(),
            challengeId: options.challengeId
        };
        const verifyResp = await fetch('/auth/passkey/register/finish', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(regPayload)
        });
        if (!verifyResp.ok) {
            let err = await verifyResp.json();
            alert("Error registrando passkey: " + (err.detail ||
verifyResp.status));
            return;
        }
        alert("¡Passkey registrada exitosamente!");
        // Aquí podríamos actualizar la UI para reflejar la nueva credencial
    } catch (err) {
        if (err.name === 'NotAllowedError') {
            alert("Registro de passkey cancelado.");
        } else {
            console.error(err);
            alert("No se pudo completar el registro de passkey. " +
err.message);
        }
    }
});
```

Este código cubre desde obtener las opciones hasta manejar errores comunes.

Manejo de errores en frontend (resumen): - **NotAllowedError**: ocurre tanto en `create()` como `get()` si el usuario cancela o el tiempo expira. Vamos a interceptarlo y dar un aviso amable en vez de saturar la consola. No lo trataremos como un fallo crítico, solo informativo. - **SecurityError/ NotSupportedError**: podría ocurrir en `create()` si `residentKey: required` pero el authenticator no lo soporta. Esto se manifiesta como `NotAllowedError` en muchos casos. Si viéramos `NotSupportedError`, mensaje "Tu dispositivo no soporta la creación de passkeys

requeridas. Intenta con otro.". - **InvalidStateError**: ocurre si se llama a `create()` sin cerrar un previo, etc. Evitaremos llamadas duplicadas deshabilitando botones mientras esperamos. - **Network errors**: fetch fallido -> "No se pudo comunicar con el servidor". - **HTTP errores**: si backend dice 400/401, usamos `err.detail` para mostrar.

También manejamos estados de la UI: - Deshabilitar el botón durante la operación para evitar doble click. - Podríamos mostrar un **loader overlay** durante la espera de interacción (ej. un mensaje "Esperando tu autenticación..."). - En caso de cancelación o error, ocultar el loader y re-habilitar controles.

Integración con la Teacher PWA

Para la PWA de profesores, la lógica es muy similar. Si la PWA es una página web instalada, la API WebAuthn funciona igual (Safari iOS soporta WebAuthn también en contextos PWA ahora). Debemos asegurarnos de usar HTTPS y que la PWA esté en el mismo dominio o asociado (posiblemente via `webcredentials` association for iOS, pero creo que Safari 15+ ya soporta WebAuthn en PWA instaladas con la condición de asociar dominio – se podría documentar, pero está avanzando su soporte).

En todo caso, replicaremos el flujo: - Pantalla de login de PWA: mismo botón, mismo código. - Pantalla de perfil PWA: mismo registro de passkey.

Podemos abstraer el código JS en un módulo compartido, pero como son apps separadas sin bundler, quizás duplicar un poco el código en cada. O hostigar un script común desde CDN (no hay librería necesitándose aparte de nuestra lógica).

Nota sobre Librerías: Existe la opción de usar la librería **@simplewebauthn/browser** para simplificar conversiones y llamadas. Dado que no usamos frameworks, podríamos incluirla via `<script>` (es un paquete UMD también). SimpleWebAuthn/browser puede hacer `startAuthentication(options)` que maneja base64 conv internamente, y `startRegistration(options)` similar. Esto reduce errores de conversión manual. Sin embargo, para evitar una dependencia extra (aunque pequeña), continuaremos con la implementación manual descrita, que es didáctica y bajo control.

Ejemplos de Interacción (para Manual de Usuario)

Vale la pena describir la interacción típica para clarificar, con base en lo implementado:

- **Login con Passkey – Ejemplo**:
- Usuario abre la página de login del panel admin. Ve opciones “Iniciar sesión con passkey” y quizás “Iniciar sesión escaneando QR” (lo que exista).
- Usuario hace clic en **"Iniciar sesión con passkey"**.
- El navegador muestra un diálogo solicitando usar un autenticador. Por ejemplo, en Windows 11 con Chrome podría mostrar opciones de **Windows Hello** (huella, rostro) o llaves de seguridad disponibles [22] [23]. En macOS, Safari/Chrome podrían preguntar si usar Touch ID o una llave externa.

  Ilustración: al usuario se le puede presentar una ventana emergente del navegador similar a la siguiente, donde el navegador le pide que elija o use su método de autenticación (ya sea tocar la llave de seguridad USB o usar el sensor integrado).

- El usuario interactúa: si es huella digital, la coloca; si es FaceID, mira a la cámara; si es llave USB, la inserta y toca el botón. En ese momento, el navegador puede indicar algo como “Toca tu llave de seguridad para completar” [21] .

- Si todo va bien, en segundos el servidor valida la respuesta y la aplicación redirige al **dashboard**. El usuario ya está autenticado sin haber tipeado nada.

- Si el usuario no tenía cuenta y este flujo creó una cuenta nueva, podría ser llevado a una pantalla para completar su nombre o asociar su correo, ya dentro del sistema.

- **Registro de nueva Passkey – Ejemplo**:

- Usuario ya logueado (profesor o admin) navega a *Perfil* > *Seguridad* (o donde coloquemos la opción) y hace clic en **"Agregar nueva Passkey"**.
- Aparece un aviso en la página: *"Para registrar una nueva llave de acceso, haz clic en continuar y sigue las instrucciones..."*.

- Al confirmar, el navegador le muestra un diálogo para crear la credencial. Por ejemplo, en Chrome un modal que dice *“Crear una llave de acceso para midominio.com”*. Si el usuario tiene varios autenticadores, primero elegirá uno (ej: una ventana *“Elegir autenticador”* puede aparecer si hay llaves USB conectadas además del Touch ID integrado) [24] . Supongamos que elige su llave USB.

  Luego, el navegador le pedirá acción: "*Touch your security key*" (Toca tu llave) como en la imagen a continuación, avisándole que se guardará un registro en su llave (lo que corresponde a la credencial residente).

- El usuario toca la llave; si no tenía PIN, es posible que se le pida establecer uno en ese momento (Chrome puede pedir “Configura un PIN para tu llave” como paso previo [25]). Si ya tenía PIN, se lo pedirá y luego confirmará tocar de nuevo [26] .

- Tras completarse, el navegador indica éxito. En la aplicación, mostramos "*Passkey registrada exitosamente*".

- A partir de entonces, en la próxima pantalla de login, el usuario podrá usar cualquiera de sus passkeys registradas (por ejemplo, tanto su laptop con Windows Hello como su llave USB); el navegador le presentará la lista si hay varias [19] . Por ejemplo, puede verse un diálogo *"Seleccione qué credencial usar para iniciar sesión"* con dos opciones identificadas por el *displayName* de cada una [19] .

- **Caso de múltiples cuentas en un dispositivo**: Si dos usuarios distintos usan la misma máquina con la misma autenticación de plataforma (ej: dos cuentas en la misma Mac con Touch ID registradas en el mismo sitio), al usar login sin username, el navegador mostrará una lista de cuentas para elegir. En esa lista aparecerá el *displayName* que se usó al registrar la credencial. Por eso es importante que no sea algo como un GUID ilegible. En nuestra implementación inicial, un nuevo registro usó un nombre placeholder (base64 del id). Eso puede resultar confuso si luego aparece en esta lista [27] . **Mitigación**: Podemos mejorar poniendo un nombre amigable post-registro. Por ejemplo, después de que un nuevo usuario se registre con passkey, podríamos ofrecerle establecer un nombre público para su passkey. O automáticamente, si tenemos el nombre real (en caso de usuario existente, usamos su nombre para displayName en registro).

Esto ayudará a que en la lista el usuario reconozca su cuenta. *(Ejemplo: en la imagen se ve una cuenta "Sam Bellen" vs otra "SomeOtherUser", claramente uno tiene un nombre descriptivo).*

En todo caso, si solo un usuario usa el dispositivo, esto no es problema: se seleccionará automáticamente su credencial única.

Resumen de cambios en cada app:

- **Login Page (web-app & teacher-pwa)**:
- Añadir botón "Iniciar con Passkey".
- Incluir script con la lógica anterior (`fetch -> navigator.credentials -> fetch`).
- Ocultar método alternativo si no soportado.
- Mantener métodos existentes (ej: botón "Iniciar con QR" si existe).

- Posiblemente estilizar el botón con el ícono de una llave o huella para indicar passkey.

- **Profile/Security Page (web-app & teacher-pwa)**:

- Añadir sección "Tus Passkeys registradas:" listándolas (opcional, se puede hacer más adelante).
- Botón "Agregar Passkey".
- Botón "Eliminar" junto a cada, si implementamos eliminación.
- Script con lógica de registro.

- Mensajes de feedback.

- **Kiosk-app**:

- Sin cambios (a menos que decidamos implementar login de admin con passkey para desbloquear kiosk, lo cual sería similar a web-app login, pero eso no estaba en requerimientos).

- **Móvil vs Desktop**:

- Asegurar que en móvil iOS Safari PWA funcione: Safari presentará la autenticación de plataforma (FaceID o TouchID) en un *sheet* modal. No requiere cambios de código.
- En Android Chrome, si el user tiene sincronización de Passkeys (Google Password Manager), al tocar *login con passkey*, Chrome puede ofrecer la selección de credencial desde su gestor (a veces aparece como un *autofill* UI).
- Probar ambos entornos (lo haremos en pruebas).

Pruebas Unitarias e Integración (Backend)

Dado lo crítico de esta funcionalidad, escribiremos una batería de pruebas. Separaremos en niveles:

Pruebas Unitarias (Backend)

Objetivo: Verificar que las funciones de generación y verificación se comportan correctamente bajo diferentes condiciones, aislando la lógica de terceros lo más posible. También probar funciones utilitarias como conversión base64 y manejo de respuestas.

- **Generación de opciones**: Simular llamadas a `start_passkey_registration` y `start_passkey_authentication`:

- Sin usuario logueado y con usuario logueado.
- Comprobar que el JSON devuelto contiene campos correctos:
  - `challenge` es string base64url de 16+ bytes.
  - `user.id` (en registro) es base64url de longitud <=64 bytes.
  - `user.name` y `displayName` no estén vacíos (deben haberse rellenado).
  - `residentKey` = "required".
  - `allowCredentials` ausente o vacía en auth.
- También comprobar que se guardó en Redis la info correspondiente (podemos mockear Redis en test o usar fakeredis).
- Verificar que dos llamadas generan challenges distintos.
- **Verificación de registro (happy path)**:
- Esto es más complejo de test unitario puro porque requiere generar una respuesta de atestación válida. Podemos simularlo usando la propia librería `webauthn` con un test vector:
  - Duo Labs en su repo puede tener ejemplos, o podríamos usar `webauthn_rp` test tools. Alternativamente, usar la librería Yubico `python-fido2` para crear un *virtual authenticator*.
  - Por simplicidad, podríamos **mockear** `verify_registration_response` para que devuelva un objeto `verified=True` con fake `credential_public_key` y `credential_id`. Así probamos la lógica alrededor (guardar en BD, crear usuario).
  - O usar credenciales de ejemplo: por ejemplo, Yubico documentation proporciona attestation dumps. Sin embargo, verificarlas requeriría también code de atestación or trust anchors, que es mucho para test unit.
- Lo más pragmático: monkeypatch `webauthn.verify_registration_response` en test para simular:

```
fake_att_info = SimpleNamespace(credential_public_key=b"\x01\x02",
credential_id=b"abcd", sign_count=0)
monkeypatch.setattr(webauthn, "verify_registration_response", lambda
**kwargs: SimpleNamespace(verified=True,
attestation_info=fake_att_info))
```

  Luego llamar a nuestra `finish_passkey_registration` con un fake request JSON (valores no importan porque lo hemos monkeypatched).
- Verificar que después de la llamada:
  - Si existing_user_id=None, se creó un nuevo User en la BD (user.handle coincide con el guardado en Redis).
  - Se creó Credential con id "abcd" etc.
  - Si existing_user_id estaba en Redis, no creó user nuevo sino usó el existente.
  - El endpoint devuelve 200 y token.
- Probar también el caso credencial duplicada: Insertar una credencial en BD antes con el mismo ID, luego simular otra vez registro con mismo ID -> debería dar 400 y no crear duplicados.
- **Verificación de autenticación (happy path)**:
- Similar approach: monkeypatch `verify_authentication_response`:
  - Simular `verified=True`, `authentication_info.new_sign_count = old_count+1`.
  - Insertar en BD previamente un Credential con known `credential_id` and `sign_count`.
  - Llamar a finish_passkey_authentication con JSON donde `id` coincide con ese credential_id, `userHandle` coincide con user.handle en base64 (podemos generarlo).
  - Verificar:

    - Retorna 200 con token.
    - Credencial en BD actualizó sign_count.
    - Si credential no existía -> retorna 401.
    - Si verify_auth_response devuelve verified=False -> 401.
- Probar error de challenge:
    - Llamar finish_passkey_auth sin haber llamado start (Redis vacío) -> 400.
    - Llamar con challengeId incorrecto -> 400.
- **Utils**: Podemos probar nuestra función de generación de user_handle:
- Comprobar que `os.urandom(32)` genera bytes len 32, no repetidos en llamadas múltiples.
- Comprobar base64url encoding/decoding es correcto (tomar un buffer conocido, codificar y decodificar, comparar con original).
- **Seguridad**: Si implementamos un check para origin mismatches:
- Podríamos testear que `expected_origin` es correcto (tomar uno distinto y simular verify -> debería fallar).
- Esto en verdad lo hace la librería, así que confiar en ella.
- **Edge cases**:
- Intentar registro sin challengeId -> 400.
- Intentar autenticación con cred id desconocido -> 401.
- Multi-cred: registrar dos credenciales para un user y luego autenticar con cualquiera, pero esto es más e2e.

Estas pruebas se ejecutarán con FastAPI TestClient o AsyncClient. Simularemos Redis quizá con un dict global en tests para no requerir un server real (monkeypatch redis.set/get). Similar con DB usando una DB sqlite en memoria para pruebas.

Pruebas de Integración (Sistema)

Estas pruebas involucran múltiples pasos pero aún sin navegador real, usando HTTP a nuestra API:

- **Flujo completo registro nuevo usuario** (semi-integration):
- Simular front-end llamando a `/register/start`, luego tomar la respuesta:
    - Extraer `challenge` y `user.id`.
    - Usar la librería Yubico `Fido2` para simular un registro:
    - Se puede usar `fido2.client.Fido2Client` en modo virtual con SoftAuthenticator.
    - Esto se complica, quizás hay una forma más manual:
        - Con py_webauthn, tal vez se pueda usar la clase `RegistrationCredential` para construir manualmente la respuesta: `RegistrationCredential.from_json({...})` quizás, pero sin la firma válida no pasará verify.
    - Dado lo complejo, podríamos cheat: monkeypatch verify as hicimos en unit.
- Mejor enfoque integrador: **No** comprobar cripto sino el *orquestamiento*:
    - Monkeypatch verify fns para devolver verified.
    - Llamar a start, luego llamar a finish con un payload construido a partir de start (challengeId) y valores dummy (ya que verify está parcheado, no importa).
    - Verificar que el usuario fue creado en DB etc, token devuelto.
- Esto prueba la secuencia de llamadas y la interacción con Redis y DB sin un cliente real.
- **Flujo login existente**:
- Crear un usuario y cred en DB manualmente (para simular uno registrado).
- Llamar /auth/start -> obtener challenge.
- Simular /auth/finish con cred id correspondiente y userHandle base64 (tomado del user.handle).
- Verificar respuesta 200 (monkeypatch verify_auth to skip signature).

- Asegurar que después de login, la cred sign_count actualizó.
- **Prueba de múltiples credenciales**:
- Registrar 2 credenciales para un mismo usuario (llamar /register/finish dos veces con monkeypatch).
- Luego login (monkeypatch verify_auth so both would verify).
- Aquí realmente lo mismo, pero al menos ver que no hay conflictos guardando dos cred con distinto id pero mismo user.
- Listar cred (si implementáramos listing endpoint).
- **Concurrent requests**:
- No es trivial simular concurrency en tests, pero podríamos asegurarnos de locks:
    - E.g. iniciar un registro, no terminarlo, iniciar otro registro (tal vez para otro user) y ver que ambos challenges se mantienen separados en Redis.
    - Probar que dos login flows paralelos (two challenge in Redis) work (simulate by calling /auth/start twice and then finishing each).
- Estas pruebas de condición de carrera se controlan en gran parte porque challengeId separa flujos.

Pruebas End-to-End (E2E)

Esto implica usar un **navegador real** automatizado para reproducir el flujo exactamente como un usuario:

Herramienta propuesta: **Playwright** (o Puppeteer). Playwright tiene soporte para *Virtual Authenticator* a través del CDP (Chrome DevTools Protocol). Podemos programar un test E2E que:

- Inicia la aplicación (por ejemplo, lanzar FastAPI server en local).
- En Playwright:
- Crear contexto de navegador.
- **Configurar un autenticador virtual**: Playwright permite:

```
await context.addInitScript(() => {
    // This doesn't add authenticator. Instead, use CDP:
});
await context.grantPermissions(['publickeyCredentials'], { origin:
'https://localhost:8000' });
const auth = await context.newCDPSession(page);
await auth.send('WebAuthn.enable');
await auth.send('WebAuthn.addVirtualAuthenticator', { options: {
     protocol: 'ctap2', transport: 'usb', hasResidentKey: true,
hasUserVerification: true, isUserVerified: true
}});
```

This returns an authenticatorId used internally. Then later we might handle calls, but Playwright can actually intercept navigator.credentials calls, or we might need to fill them manually: There's an easier approach: use **passkeys E2E testing libraries**. The Corbado blog [28] and others show how to simulate.
- Or simpler: have Playwright run non-headless and pause for human input (not fully automated).
- Due to time, we might not fully automate cryptographic steps. Instead, we could do a semi-manual test:

- Run the server, open a real browser to the app, test with a real platform authenticator (Windows Hello, etc.) to ensure everything flows. This is more QA than automated test.

However, given the user asked por *pruebas e2e (e12)*, interpretamos que se desea al menos planificar su uso. Podríamos escribir un test like:

```
test('Usernameless login flow works', async ({ page }) => {
  // Navigate to login
  await page.goto('https://localhost:8000/login');
  // Ensure passkey button is visible
  await expect(page.locator('#passkeyLoginBtn')).toBeVisible();
  // Setup virtual authenticator
  const authenticator = await page.context().newCDPSession(page);
  await authenticator.send('WebAuthn.enable');
  const authenticatorId = (await
authenticator.send('WebAuthn.addVirtualAuthenticator', { options:
{...} })).authenticatorId;
  // Create credential in the virtual authenticator for user
  // ... Use WebAuthn.addCredential CDP command with a known credential for a
user handle
  // For brevity, assume we precomputed a credential for user handle X
  // Actually, a simpler approach is to let the test create a credential by
going through registration first:
  // Click "Register new passkey" in UI (if we had sign up flow on login
page).
  // But our UI might not have sign-up separate; hmm.
  // We might cheat by directly adding a credential via CDP:
  const credential = {
    // Create a new credential with userHandle = some bytes, privateKey =
random, signCount = 0, etc.
  };
  await authenticator.send('WebAuthn.addCredential', { authenticatorId,
credential });
  // Now simulate login:
  await page.click('#passkeyLoginBtn');
  // The browser should auto-select the credential since allowCredentials
empty.
  // Virtual authenticator will respond with our credential's assertion.
  // Wait for navigation or success indicator:
  await page.waitForURL('**/dashboard');
  // Assert user is logged in by checking presence of logout button etc.
});
```

Setting up the credential data for `WebAuthn.addCredential` requires generating a valid key pair and the correct RP ID hash. This is doable using Node crypto libraries or precomputed test vectors. This goes deep, but there are resources (like the *hoop.dev* article or authgear).

Alternatively, we could rely on *Conditional UI* where the browser can pick credential, but virtual authenticator likely requires explicit credentials.

Given complexity, another approach: **Nightwatch** or **Selenium** with a real browser and actual OS prompts might not be automatable fully.

Thus, focusing on Playwright with virtual authenticator is best for a headless solution. The references [29] [30] confirm it's possible. We will plan tests accordingly, even si no implementamos them now.

Pruebas de UI/UX Manuales

Además de pruebas automatizadas, detallaremos pruebas manuales que se deben realizar: - Probar login con: - Chrome en Windows (usar Windows Hello, y usar llave externa). - Safari en Mac (Touch ID). - Safari en iPhone (FaceID) tanto en navegador como PWA instalada. - Edge en Windows. - Firefox (Firefox soporta WebAuthn pero quizá con menos features; probar un YubiKey). - Probar flujos de error manualmente: - Cancelar el prompt en registro y login, verificar mensajes. - Probar con un YubiKey sin PIN e intentar registrar: el flujo de asignar PIN. - Probar con dispositivo no soportado (difícil hoy, quizá un YubiKey U2F via Firefox?). - Desconectar la red después de obtener challenge, ver manejo de error. - Probar interoperabilidad: - Registrar passkey en Chrome desktop, luego intentar login en Safari iPhone (allí se puede usar la función de scan QR para usar la credencial del teléfono si la cuenta está guardada, iOS tiene *Passkey sharing* via QR-Bluetooth; eso es avanzado, pero podríamos documentar que por ahora login passkey asume usar el mismo dispositivo). - Realizar login en dispositivo distinto al de registro: Esto requiere que el usuario registre passkeys en cada dispositivo, o use una passkey sincronizada (Google/Apple implementan sincronización: ex. Passkey en iCloud Keychain se sincroniza entre iPhone y Mac). Sería bueno probarlo: registrar en iPhone, luego Chrome en Mac debería ofrecer esa passkey via iCloud if the AppleID is same and site is on associated domains. Probablemente fuera de alcance probar todos, pero hay que tenerlo en cuenta.

Documentación y Manual de Usuario

Una parte esencial es la **documentación** tanto técnica como para el usuario final:

Documentación Técnica (para desarrolladores)

Debemos agregar/actualizar la documentación del proyecto (README o Wiki interno) con:

- **Configuración**: Explicar las variables nuevas: `RP_ID`, `RP_NAME`, `RP_ORIGIN`. Indicar que en producción `RP_ID` debe ser el dominio público exacto. Si se despliega en varios entornos (staging, etc.), ajustar esos valores.
- **Dependencias**: Mencionar la adición de la librería py_webauthn y su versión. Cómo instalar (ya en requirements.txt).
- **Migraciones DB**: Describir la nueva tabla Credential y el nuevo campo `User.handle`. Proporcionar un script Alembic de migración:

```
ALTER TABLE users ADD COLUMN handle BYTEA;
UPDATE users SET handle = <random bytes> WHERE handle IS NULL;
ALTER TABLE users ALTER COLUMN handle SET NOT NULL;
ALTER TABLE users ADD CONSTRAINT users_handle_unique UNIQUE(handle);
CREATE TABLE credentials (
    credential_id VARCHAR PRIMARY KEY,
    user_id INT REFERENCES users(id),
    public_key BYTEA NOT NULL,
    sign_count INT NOT NULL,
```

```
    transports VARCHAR
);
```

(El update generando random bytes puede hacerse con una función PL/pgSQL o en Python migración).

- **API**: Documentar los nuevos endpoints:
- `POST /auth/passkey/register/start` – inicia registro, requerimientos (autenticado opcional).
- `POST /auth/passkey/register/finish` – finaliza, qué espera (ejemplo JSON).
- `POST /auth/passkey/authenticate/start` – inicia login.
- `POST /auth/passkey/authenticate/finish` – finaliza login. Incluir ejemplos de requests y responses. *Ejemplo:*

```
$ curl -X POST https://midominio.com/auth/passkey/authenticate/start -d
'' -c cookies.txt
{ "challenge": "45j...-A", "rpId": "midominio.com", "timeout": 60000,
"userVerification": "required", "challengeId": "uuid-1234" }

# (Luego en el cliente se usa navigator.credentials.get, etc.)

$ curl -X POST https://midominio.com/auth/passkey/authenticate/finish \
       -H "Content-Type: application/json" \
       -d '{"id":"...", "rawId":"...", "type":"public-key", "response":
{...}, "challengeId":"uuid-1234"}' -b cookies.txt -c cookies.txt
{ "message": "Autenticación exitosa" }
```

- **Flujo de registro usernameless**: Describir cómo un nuevo usuario puede registrarse sin proporcionar datos, y que luego se le asignará un identificador interno. Comentar que si se quiere luego asociar un correo/username para reconocimiento, se puede (nuestro sistema quizá no requiera).
- **Sesiones**: Explicar cómo se maneja la sesión tras login (si JWT en localStorage o cookie). Esto para integridad de documentación.
- **Seguridad**: Listar las consideraciones ya mencionadas (challenge, sign_count, etc.). Reforzar que se debe usar HTTPS y un dominio estable, etc. Señalar cumplimiento de lineamientos FIDO: credenciales no almacenan PII, todo OK.
- **Límites conocidos**: Mencionar que si un usuario quiere usar passkey en varios dispositivos, deberá registrar una en cada uno (a menos que use plataformas con sincronización de passkeys, lo cual ocurre automáticamente en Apple/Google ecosystems). Si pierde acceso a su dispositivo y no tiene otro método registrado, necesitará asistencia (recuperación de cuenta). Esto abre un tema: *Account Recovery*. Podríamos decir: "Actualmente, si pierde su dispositivo con la passkey, un administrador debe generar un QR alternativo o registrar un nuevo método para el usuario manualmente. Considere implementar flujos de recuperación (p.ej., código por email o similar) [31] [32] ." Esto es un aspecto más allá, pero importante documentarlo.
- **Testing**: Incluir instrucciones para correr las pruebas unitarias (`pytest`) y cualquier configuración (puede necesitar variables de entorno dummy para RP_ID, etc. en CI). Mencionar uso de `pytest` markers para tests e2e (que quizás requieren Chrome).
- **Referencias**: Podemos citar recursos útiles para entender WebAuthn (por ejemplo, link a MDN or Yubico dev guide) para futuros devs que lean doc:
- Indicar que esta implementación sigue la idea de *usernameless authentication* donde el `user.id` es el `user_handle` aleatorio y es fundamental [14] .

- Apuntar a la documentación de Yubico donde se explica user handle y credenciales residentes [10].

Manual de Usuario (con pantallas)

El manual de usuario debe explicar a los usuarios finales (probablemente admins/profes, dado que estudiantes usan kiosco) cómo usar esta nueva funcionalidad. Incluirá:

- **Qué es una passkey**: Explicación breve en términos simples ("una forma segura de iniciar sesión sin contraseña, usando la huella, rostro o un dispositivo de seguridad").
- **Requisitos**: Un dispositivo compatible (PC o smartphone moderno) y un navegador actualizado. Para usar huella o FaceID, que el dispositivo tenga esos sensores configurados. Si no, que disponga de una llave de seguridad externa.
- **Registro de una passkey (como usuario existente)**:
- Instrucciones paso a paso:
    - Ir a perfil > Seguridad.
    - Clic en "Agregar Passkey".
    - Seguir el prompt: el navegador te pedirá crear la credencial. Incluye capturas:
    - *Captura 1:* La pantalla de nuestra app con el botón "Agregar Passkey" resaltado.
    - *Captura 2:* El diálogo del navegador pidiéndole que use su autenticador (por ejemplo, un recuadro de Windows Hello o un cuadro de diálogo de Chrome como se mostró antes). E.g., en Windows: *"Use Windows Hello o una llave de seguridad"*. En Mac: *"¿Desea guardar una llave de acceso para midominio.com?"*.
    - *Captura 3:* Mensaje de éxito en nuestra app tras registrar (ej. un toast "Passkey registrada").
    - Nota: Si el dispositivo le pide configurar PIN o similar, hacer eso (ese paso es propio del dispositivo).
- Explicar que puede registrar varias passkeys (por ejemplo, una del teléfono y otra de la PC).
- Recomendar registrar al menos 2 métodos (por si pierde uno, tenga otro).
- **Inicio de sesión con passkey**:
- Mostrar la pantalla de login con el botón "Iniciar con passkey". *Captura 4:* Pantalla de login con el botón marcado.
- Explicar que al hacer clic, *puede* que:
    - Directamente se abra un diálogo de huella/rostro (si solo hay un usuario/credencial).
    - O le pida elegir entre cuentas si hay varias (mostrar ejemplo de lista de cuentas si posible). *Captura 5:* Un ejemplo de selector de cuenta (como la imagen de Auth0 con "Select an account", podríamos reetiquetar mentalmente).
- Luego, una vez que use su método (huella etc.), debería entrar directamente sin más pasos. *Captura 6:* Pantalla principal tras login, indicando "Hola [Nombre]".
- Mencionar fallback: Si no funciona (ej: el dispositivo no ofrece la opción), pueden usar el método alternativo (QR/NFC).
- También indicar: "Si su navegador le ofrece guardar la passkey en la nube (ej: *"Guardar llave de acceso con su cuenta Google/Apple"*), puede aceptarlo para facilitar uso en otros dispositivos." (Esto pasa en Chrome con cuenta Google, y en Safari con iCloud).
- **Eliminación de passkey**:
- Si implementamos el eliminar, mostrar cómo: en perfil, lista de passkeys con botón eliminar. Advertir que eliminarla significa que ya no podrá iniciar con ese dispositivo hasta registrar de nuevo.
- Captura si fuera implementado.
- **Soporte y resolución de problemas**:

- FAQ:
    - *¿Qué hago si cambio de navegador o dispositivo?* – Respuesta: Debe registrar una nueva passkey en el nuevo dispositivo (la passkey anterior no "salta" de un dispositivo a otro salvo que use cuentas en la nube). Ej: si registró en Chrome Windows y ahora quiere usar Safari en iPhone, deberá registrar en iPhone también.
    - *¿Qué pasa si pierdo mi dispositivo o me roban la llave de seguridad?* – Respuesta: Contactar a TI (o admin) para que eliminen esa credencial de su cuenta y registrar otra. Ofrecer métodos de emergencia (quizá login via código SMS si existiera, pero nuestro sistema no tiene implementado, así que sería via admin).
    - *El navegador me dice "No se pudo usar passkey"* – Sugerir verificar que el dispositivo tenga un método de desbloqueo configurado (PIN/huella). En Windows Hello, por ej, se debe tener PIN o huella enrollada.
    - *Compatibilidad* – Todos los navegadores modernos soportan. IE no soporta (pero nadie debería usar IE a esta altura).
    - *Privacidad* – Aclarar que la passkey no comparte datos personales con el servidor, solo una clave pública [33], y que el huella/FaceID nunca sale del dispositivo.
    - *¿Puedo usar la misma passkey para varias cuentas?* – Cada passkey se asocia a una cuenta interna. Si es el mismo sitio, usualmente se tiene una por cuenta. Si un usuario tiene dos cuentas distintas en nuestro sistema (poco común, pero si fuera), podría registrar dos passkeys o la misma en teorìa podría usarse para ambas si el authenticator lo soporta con diferentes user handle, pero es mejor no complicarlo. Asumimos 1 persona = 1 cuenta, así que no aplica.
    - *¿Puedo usar la passkey de Google/Apple que ya uso en otros sitios?* – Sí, al momento de iniciar sesión, tu navegador te puede sugerir usar una passkey existente (ej: en Safari aparecerá tu llavero de iCloud si tienes una passkey guardada para este sitio), o en Chrome con cuenta Google similar. Depende de si ya registraste para este sitio; la primera vez siempre debes registrar.
- **Screenshots**: Hay que preparar varias. Dado que no podemos obtener directamente de la app que aún no existe, podemos usar representaciones equivalentes:
- Podemos capturar de un entorno de prueba. Alternativamente, usar imágenes de documentación WebAuthn (como ya obtuvimos) para ilustrar los diálogos del navegador:
    - The Auth0 blog images (we have a couple above) are useful: *Ejemplo de prompt seleccionando autenticador (lo usaríamos para ilustrar punto en login).* Ejemplo de prompt de tocar llave (ilustrar registro). * Ejemplo de selección de credencial (ilustrar multi-cuenta login).
    - Las capturas de nuestra UI (botones) tendremos que crearlas manualmente una vez implementado. Podemos indicar en el manual con placeholders en esta entrega, pero posteriormente el agente desarrollador deberá reemplazarlas con reales de la aplicación funcionando.
- **Paso a paso con imágenes**: El manual tendrá secciones "Cómo registrarse con passkey" y "Cómo iniciar sesión", enumerando los pasos con imágenes integradas.

Al inicio del manual clarificaríamos el público objetivo (administradores, profesores, apoderados). Usar lenguaje sencillo.

Consideraciones de implementación del manual:

El agente (desarrollador) deberá crear estas capturas en entorno de staging. Sugerimos usar por ejemplo: - Un navegador Chrome para capturar el flujo de registro (como la imagen que mostramos de "Touch your security key", se puede recrear). - Un iPhone para capturar FaceID prompt (si fuera posible)

– aunque iOS prompts no se capturan fácilmente, podríamos omitir si complicado. - O usar imágenes de marketing FIDO (posiblemente evitamos por licencias, mejor propias).

Finalmente, asegurarnos que toda esta funcionalidad se integra sin romper nada existente. Esto requiere probar los flujos antiguos de login (NFC/QR) para verificar que la introducción de estos endpoints y quizás UI nuevas no interfieren. Por ejemplo, si antes al entrar al panel admin se esperaba un QR code scan, ahora con passkey añadido, debemos decidir si ambos métodos aparecen simultáneamente o excluyentes. Podemos mostrar ambas opciones: "Escanear QR" y "Passkey". Documentar para el equipo de UX que la pantalla de login necesita esos dos botones claramente.

En resumen, con esta implementación detallada cubrimos desde el almacenamiento de credenciales seguras en backend, hasta la interacción del usuario en frontend, con gestión de errores y soporte multiplataforma. Estamos apoyándonos en fuentes autorizadas (Yubico, Duo Labs, Auth0) que describen cómo las credenciales residentes permiten omitir el ingreso de username [2] y cómo un *user handle* aleatorio mapea la credencial al usuario [18]. También hemos seguido consejos de configuración estándar (residentKey=Required, UV=Preferred/Required) [12] y recordado puntos de producción (usar Redis para desafíos y HTTPS obligatorio) [16].

A continuación consolidamos algunos de esos puntos citados en la documentación para respaldo:

- *Cita (Yubico) - Beneficio de credenciales residentes: "WebAuthn… permite un login sin contraseña y sin nombre de usuario. … el autenticador almacena la clave privada y devuelve el user handle en la autenticación, permitiendo al RP identificar al usuario sin que este ingrese un username."* [2].
- *Cita (Auth0) - Experiencia fluida: "Las credenciales residentes permiten una experiencia de login más fluida, eliminando la necesidad de que el usuario introduzca un nombre de usuario"*, gracias a guardar datos del usuario en el autenticador [1].
- *Cita (StackOverflow) - Implementación usernameless: "Para un uso completamente 'usernameless' de WebAuthn… simplemente genera valores ficticios para los tres campos requeridos en* `user` *al llamar a* `navigator.credentials.create()` *… usa un ID aleatorio generado por el servidor como* `user.id` *(y codificaciones de ese para name/displayName). … Cuando el usuario vuelva para login, invoca* `navigator.credentials.get()` *con* `allowCredentials: []` *vacío. Esto permitirá al usuario seleccionar la credencial discoverable guardada anteriormente."* [14] [17].
- *Cita (Yubico) - User Handle en BD: "El user handle es requerido para autenticación sin usuario con credenciales discoverable… al loguear, el autenticador retorna el user handle junto con la firma, y el servidor lo utiliza para buscar al usuario en su base de datos, concluyendo quién inició sesión (sin haber requerido username ni password luego del registro)."* [10] [18].
- *Cita (Demo PasskeyAuth) - Configuración recomendada: "ID de Relying Party: localhost (en dev); Resident Keys: Required (habilita credenciales discoverable); User Verification: Preferred; … En producción usar almacenamiento apropiado de usuarios y sesiones, considerar Redis para desafíos en entornos con instancias múltiples, y asegurar HTTPS"*. [34] [16].

Con esta base teórica y la guía práctica escrita, el desarrollador tiene un **prompt completo** para implementar la autenticación WebAuthn (passkeys) en el proyecto. Incluimos manejo de errores, fallbacks, ejemplos de código y pruebas, así como indicaciones para la documentación y manual de usuario con sus screenshots ilustrativos. Siguiendo estos lineamientos, la implementación deberá quedar **perfectamente integrada y funcional** en el ecosistema de la aplicación.

---

1 4 13 19 21 24 25 26 A Look at WebAuthn Resident Credentials
https://auth0.com/blog/a-look-at-webauthn-resident-credentials/

2 3 Resident Keys
https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/Resident_Keys.html

5 8 10 11 18 33 User Handle
https://developers.yubico.com/WebAuthn/WebAuthn_Developer_Guide/User_Handle.html

6 Overview — py_webauthn v2.2.0 documentation
https://duo-labs.github.io/py_webauthn/overview.html

7 py_webauthn — py_webauthn v2.2.0 documentation
https://duo-labs.github.io/py_webauthn/

9 Server Configuration — webauthn-rp documentation
https://webauthn-rp.readthedocs.io/en/latest/getting-started/server-configuration.html

12 16 34 git.zi.fi
https://git.zi.fi/LeoVasanko/passkey-auth/raw/commit/a0da799c9e09a9d0a82673019b0494b5ca37c275/README.md

14 15 17 20 27 31 32 WebAuthn: how to get rid of the username requirement? - Stack Overflow
https://stackoverflow.com/questions/73562080/webauthn-how-to-get-rid-of-the-username-requirement

22 23 WebAuthn Public Key Credential / User-Agent Hints
https://www.corbado.com/blog/webauthn-public-key-credential-hints

28 29 Passkeys E2E Playwright Testing via WebAuthn Virtual Authenticator
https://www.corbado.com/blog/passkeys-e2e-playwright-testing-webauthn-virtual-authenticator

30 A Practical Guide: Automating Passkey Testing with Playwright and ...
https://www.oursky.com/blogs/a-practical-guide-automating-passkey-testing-with-playwright-and-authgear