

PWA z JavaScript

Projekt

Igor Barcik 131780

2025-05-23

Budget Manager PWA - Raport projektu

1. Wprowadzenie

Aplikacja “Budget Manager PWA” ([link do git](#)) to progresywna aplikacja internetowa (PWA) służąca do zarządzania budżetem osobistym. Umożliwia śledzenie wydatków i przychodów, kategoryzowanie transakcji, generowanie raportów oraz ustawianie powiadomień związanych z budżetem. Aplikacja działa zarówno online, jak i offline, dzięki wykorzystaniu nowoczesnych technologii webowych.

1.1 Cel projektu

Celem projektu było stworzenie aplikacji PWA, która:

- Umożliwia śledzenie wydatków i przychodów użytkownika
- Działa zarówno online, jak i offline
- Przechowuje dane w Local Storage oraz MongoDB
- Oferuje funkcje budżetowania, raportowania i analizy finansowej

1.2 Wykorzystane technologie

Frontend:

- Vue 3 z Composition API
- TypeScript
- Tailwind CSS
- Vite (bundler)
- Pinia (zarządzanie stanem)
- Vue Router

Backend:

- Koa.js
- TypeScript
- MongoDB

PWA:

- Service Worker
- Web App Manifest
- IndexedDB (poprzez abstrakcję)

2. Architektura aplikacji

2.1 Struktura projektu

Projekt jest podzielony na dwie główne części:

Frontend

```
frontend/
├── public/                # Statyczne pliki i manifest PWA
│   ├── manifest.json    # Konfiguracja PWA
│   ├── sw.js            # Service Worker
│   └── icons/           # Ikony dla PWA
├── src/
│   ├── components/      # Komponenty Vue
│   ├── pages/           # Widoki aplikacji
│   ├── stores/          # Pinia stores
│   ├── lib/             # Funkcje pomocnicze
│   │   └── pwa-utils.ts  # Narzędzia związane z PWA
│   ├── main.ts          # Główny plik aplikacji
│   └── styles/          # Style CSS
```

Backend

```
backend/
├── src/
│   └── app.ts            # Główny plik aplikacji
```

```

├── controllers/    # Kontrolery REST API
├── middleware/     # Middleware (np. autoryzacja)
├── services/       # Serwisy (np. baza danych)
└── scripts/
    └── generate-vapid-keys.js # Skrypt do generowania kluczy VAPID

```

2.2 Przepływ danych

1. Dane użytkownika są przechowywane lokalnie w Local Storage dla dostępu offline
2. Przy połączeniu z internetem dane są synchronizowane z bazą MongoDB na serwerze
3. Service Worker zapisuje dane do cache, umożliwiając działanie aplikacji offline
4. Background Sync API jest używane do synchronizacji danych, gdy połączenie internetowe zostanie przywrócone

3. Implementacja PWA

3.1 Manifest aplikacji

Manifest definiuje jak aplikacja ma się zachowywać po zainstalowaniu na urządzeniu użytkownika:

```

{
  "name": "Budget Manager PWA",
  "short_name": "Budget",
  "description": "Budget Manager application for tracking expenses and income",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#3b82f6",
  "icons": [
    {
      "src": "/icons/icon-192px.jpg",
      "sizes": "192x192",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "/icons/icon-512px.jpg",
      "sizes": "512x512",
      "type": "image/png",
      "purpose": "any maskable"
    }
  ],
  "shortcuts": [
    {
      "name": "Add Transaction",
      "short_name": "Add",
      "description": "Add a new transaction",
      "url": "/add",
      "icons": [{ "src": "/icons/add-icon-96x96.png", "sizes": "96x96" }]
    },
    {
      "name": "View Reports",
      "short_name": "Reports",
      "description": "View budget reports",
      "url": "/reports",
      "icons": [{ "src": "/icons/report-icon-96x96.png", "sizes": "96x96" }]
    }
  ]
}

```

3.2 Service Worker

Service Worker obsługuje:

- Cache'owanie zasobów dla dostępu offline
- Synchronizację w tle (background sync)

Kluczowe fragmenty implementacji:

```
// Instalacja Service Worker'a
self.addEventListener("install", (event) => {
  console.log("Service worker install event");
  event.waitUntil(
    caches
      .open(CACHE_NAME)
      .then((cache) => {
        return cache.addAll(precacheResources);
      })
      .then(() => {
        return self.skipWaiting();
      }),
  );
});

// Strategia fetch: cache-first, następnie sieć
self.addEventListener("fetch", (event) => {
  // Pomijanie żądań cross-origin i API
  if (
    !event.request.url.startsWith(self.location.origin) ||
    event.request.url.includes("/api/")
  ) {
    return;
  }

  event.respondWith(
    caches.match(event.request).then((cachedResponse) => {
      if (cachedResponse) {
        return cachedResponse;
      }

      return fetch(event.request)
        .then((response) => {
          // Cache'owanie odpowiedzi
          if (
            response &&
            response.status === 200 &&
            response.type === "basic"
          ) {
            const responseToCache = response.clone();
            caches.open(CACHE_NAME).then((cache) => {
              cache.put(event.request, responseToCache);
            });
          }
          return response;
        })
        .catch(() => {
          // Fallback w przypadku braku sieci
          if (event.request.mode === "navigate") {
            return caches.match("/");
          }
          return null;
        });
    });
  );
});

// Obsługa background sync
self.addEventListener("sync", (event) => {
  if (event.tag === "sync-transactions") {
    event.waitUntil(syncTransactions());
  } else if (event.tag === "sync-categories") {
    event.waitUntil(syncCategories());
  }
});
```

3.3 Rejestracja Service Worker

Rejestracja Service Worker'a w aplikacji:

```
export async function registerServiceWorker() {
  if ("serviceWorker" in navigator) {
    try {
      const registration = await navigator.serviceWorker.register("/sw.js", {
        scope: "/",
      });

      if (registration.installing) {
        console.log("Service worker installing");
      } else if (registration.waiting) {
        console.log("Service worker installed");
      } else if (registration.active) {
        console.log("Service worker active");
      }

      // Upewniamy się, że service worker jest gotowy przed próbą subskrypcji
      await navigator.serviceWorker.ready;

      return registration;
    } catch (error) {
      console.error("Registration failed with error:", error);
      return null;
    }
  }
  return null;
}
```

3.4 Powiadomienia Push

Powiadomienia Push są kluczowym elementem aplikacji PWA, umożliwiającym informowanie użytkownika o ważnych zdarzeniach, nawet gdy aplikacja nie jest aktywna w przeglądarce. W Budget Manager PWA, powiadomienia push są wykorzystywane do informowania użytkowników o nowych transakcjach dodanych do ich budżetu.

3.4.1 Klucze VAPID Do bezpiecznego wysyłania powiadomień push wykorzystywany jest protokół VAPID (Voluntary Application Server Identification). Klucze VAPID (publiczny i prywatny) są generowane za pomocą skryptu i przechowywane jako zmienne środowiskowe na serwerze.

Skrypt backend/scripts/generate-vapid-keys.js generuje te klucze:

```
# backend/scripts/generate-vapid-keys.js
const webpush = require('web-push');
const fs = require('fs');

const vapidKeys = webpush.generateVAPIDKeys();

console.log('Public Key: ', vapidKeys.publicKey);
console.log('Private Key: ', vapidKeys.privateKey);

// Zapisz klucze do pliku .env lub użyj bezpośrednio
fs.appendFileSync('.env.example', `\nVAPID_PUBLIC_KEY=${vapidKeys.publicKey}`);
fs.appendFileSync('.env.example', `VAPID_PRIVATE_KEY=${vapidKeys.privateKey}\n`);
console.log('VAPID keys appended to .env.example');
```

Klucz publiczny VAPID jest udostępniany frontendowi, aby mógł zasubskrybować usługę push. Klucze są ładowane na serwerze przy starcie aplikacji.

3.4.2 Implementacja Frontend Subskrypcja powiadomień:

Logika subskrypcji znajduje się w frontend/src/lib/pwa-utils.ts. Użytkownik może włączyć lub wyłączyć powiadomienia na stronie ustawień (frontend/src/pages/Settings.vue). Proces obejmuje:

1. Sprawdzenie wsparcia dla Service Worker i Push API.
2. Poproszenie użytkownika o zgodę na otrzymywanie powiadomień.

3. Pobranie klucza publicznego VAPID z backendu (/api/push/vapid-public-key).
4. Subskrypcja usługi push za pomocą `registration.pushManager.subscribe()`.
5. Wysłanie obiektu subskrypcji na backend (/api/push/subscribe) w celu zapisania.

Fragment kodu z `pwa-utils.ts` odpowiedzialny za subskrypcję:

```
// frontend/src/lib/pwa-utils.ts
// ... (uproszczony przykład)
async function subscribeToPushNotifications(registration: ServiceWorkerRegistration) {
  try {
    const vapidPublicKeyResponse = await fetch('/api/push/vapid-public-key', { headers: { 'Accept':
      ↪ 'text/plain' } });
    if (!vapidPublicKeyResponse.ok) {
      throw new Error(`Failed to get VAPID public key: ${vapidPublicKeyResponse.statusText}`);
    }
    const vapidPublicKey = await vapidPublicKeyResponse.text();

    const subscription = await registration.pushManager.subscribe({
      userVisibleOnly: true,
      applicationServerKey: urlBase64ToUint8Array(vapidPublicKey),
    });

    // Odczyt tokena JWT z localStorage
    const token = localStorage.getItem('authToken'); // Upewnij się, że klucz jest poprawny
    if (!token) {
      console.warn('No auth token found, cannot save subscription to backend.');
```

// Można rozważyć zapisanie subskrypcji lokalnie i wysłanie później

```
      return false;
    }

    await fetch('/api/push/subscribe', {
      method: 'POST',
      body: JSON.stringify(subscription),
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${token}`
      },
    });
    console.log('User is subscribed.');
```

return true;

```
  } catch (error) {
    console.error('Failed to subscribe the user: ', error);
    return false;
  }
}

function urlBase64ToUint8Array(base64String: string): ArrayBuffer {
  const padding = '='.repeat((4 - base64String.length % 4) % 4);
  const base64 = (base64String + padding)
    .replace(/-/g, '+')
    .replace(/_/g, '/');

  const rawData = window.atob(base64);
  const outputArray = new Uint8Array(rawData.length);

  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i);
  }
  return outputArray.buffer; // Zwraca ArrayBuffer
}
```

Obsługa powiadomień w Service Worker (`frontend/public/sw.js`):

Service Worker nasłuchuje na zdarzenie `push` i `notificationclick`.

- Po otrzymaniu zdarzenia `push`, Service Worker wyświetla powiadomienie systemowe z danymi otrzymanymi od serwera (np. tytuł, treść, ikonę). Wiadomość jest również przekazywana do aktywnych klientów aplikacji w celu wyświetlenia toastu.
- Zdarzenie `notificationclick` obsługuje interakcję użytkownika z powiadomieniem, np. otwarcie aplikacji

lub konkretnej strony.

```
// frontend/public/sw.js
self.addEventListener('push', event => {
  let data = { title: 'Nowa wiadomość', body: 'Otrzymałeś nowe powiadomienie.', data: { url: '/' } };
  if (event.data) {
    try {
      data = event.data.json();
    } catch (e) {
      console.error('Push event data is not valid JSON:', event.data.text());
      data.body = event.data.text(); // Użyj tekstu jako body jeśli JSON się nie sparsuje
    }
  }

  const title = data.title || 'Budget Manager';
  const options = {
    body: data.body || 'Nowa aktywność na Twoim koncie.',
    icon: data.icon || '/icons/icon-192px.jpg',
    badge: '/icons/icon-192px.jpg', // Użyj tej samej ikony dla badge
    data: data.data || { url: '/' }
  };

  event.waitUntil(self.registration.showNotification(title, options));

  // Wysyłanie wiadomości do aktywnych klientów (zakładek aplikacji)
  self.clients.matchAll({ type: 'window', includeUncontrolled: true }).then(clients => {
    if (clients && clients.length) {
      clients.forEach(client => {
        client.postMessage({
          type: 'PUSH_NOTIFICATION_RECEIVED',
          payload: data // Przekaż całe dane, w tym title, body, data.url
        });
      });
    }
  });

  self.addEventListener('notificationclick', event => {
    event.notification.close();

    const urlToOpen = event.notification.data && event.notification.data.url
      ? event.notification.data.url
      : '/';

    event.waitUntil(
      clients.matchAll({ type: 'window', includeUncontrolled: true }).then(windowClients => {
        for (let i = 0; i < windowClients.length; i++) {
          const client = windowClients[i];
          // Sprawdź, czy URL klienta (po usunięciu hasha) pasuje do urlToOpen
          const clientURL = new URL(client.url);
          const targetURL = new URL(urlToOpen, self.location.origin); // Upewnij się, że targetURL jest
            ↳ absolutny
          if (clientURL.origin + clientURL.pathname === targetURL.origin + targetURL.pathname && 'focus'
            ↳ in client) {
            return client.focus();
          }
        }
        if (clients.openWindow) {
          return clients.openWindow(urlToOpen);
        }
      })
    );
  });
});
```

3.4.3 Implementacja Backend Backend (backend/src/controllers/push.ts) odpowiada za:

1. Udostępnianie publicznego klucza VAPID (/api/push/vapid-public-key). Ten endpoint jest publicznie dostępny.

2. Rejestrowanie subskrypcji użytkowników (/api/push/subscribe). Subskrypcje są przechowywane w bazie danych, powiązane z użytkownikiem.
3. Wysyłanie powiadomień do zasubskrybowanych użytkowników. Na przykład, po dodaniu nowej transakcji (backend/src/controllers/protected.ts), serwer wysyła powiadomienie.

Fragment kodu z pushController.ts (uproszczony):

```
// backend/src/controllers/push.ts
import webpush from 'web-push';
import { Context } from 'koa';
import User from '../models'; // Załóżmy, że User model ma pole pushSubscriptions

// Konfiguracja VAPID keys przy starcie aplikacji (np. w app.ts lub dedykowanym pliku konfiguracyjnym)
// webpush.setVapidDetails(
//   'mailto:your-email@example.com',
//   process.env.VAPID_PUBLIC_KEY!,
//   process.env.VAPID_PRIVATE_KEY!
// );

export const getVapidPublicKey = async (ctx: Context) => {
  if (!process.env.VAPID_PUBLIC_KEY) {
    ctx.throw(500, 'VAPID public key not configured');
    return;
  }
  ctx.body = process.env.VAPID_PUBLIC_KEY;
  ctx.set('Content-Type', 'text/plain');
};

export const subscribe = async (ctx: Context) => {
  const subscription = ctx.request.body as webpush.PushSubscription;
  const userId = ctx.state.user.id; // ID zalogowanego użytkownika

  try {
    // Zapisz subskrypcję w dokumencie użytkownika
    // Załóżmy, że model User ma pole pushSubscriptions: webpush.PushSubscription[]
    await User.findByIdAndUpdate(userId, { $addToSet: { pushSubscriptions: subscription } });
    ctx.status = 201;
    ctx.body = { message: 'Subscribed successfully' };
  } catch (error) {
    console.error('Error saving subscription:', error);
    ctx.throw(500, 'Failed to save subscription');
  }
};

// Funkcja do wysyłania powiadomień (może być w serwisie)
export const sendPushNotification = async (userId: string, payload: object) => {
  try {
    const user = await User.findById(userId);
    if (!user || !user.pushSubscriptions || user.pushSubscriptions.length === 0) {
      console.log(`No push subscriptions found for user ${userId}`);
      return;
    }

    const notificationPayload = JSON.stringify(payload);

    for (const sub of user.pushSubscriptions) {
      try {
        await webpush.sendNotification(sub, notificationPayload);
      } catch (error: any) {
        console.error('Error sending notification to a subscription: ', error.body);
        // Jeśli subskrypcja jest nieaktualna (np. status 410 Gone), usuń ją
        if (error.statusCode === 410 || error.statusCode === 404) {
          await User.findByIdAndUpdate(userId, { $pull: { pushSubscriptions: sub } });
          console.log('Removed outdated subscription.');
```

```

    } catch (error) {
      console.error(`Failed to send push notifications for user ${userId}:`, error);
    }
  };
};

```

Integracja wysyłania powiadomień przy tworzeniu transakcji w `protectedController.ts`:

```

// backend/src/controllers/protected.ts
// ...
import { sendPushNotification } from './push'; // Zaimportuj funkcję

export const addTransaction = async (ctx: Context) => {
  // ... logika dodawania transakcji ...
  // const newTransaction = ...; // Wynik dodania transakcji
  // const userId = ctx.state.user.id;

  // Po pomyślnym dodaniu transakcji:
  if (newTransaction && userId) { // Upewnij się, że newTransaction i userId istnieją
    const notificationPayload = {
      title: 'Nowa transakcja',
      body: `Dodano: ${newTransaction.description} (${newTransaction.amount} ${newTransaction.currency}
↪ || 'PLN')`,
      data: { url: `/transactions` } // Ogólny link do transakcji lub ID:
↪ /transactions/${newTransaction._id}
    };
    // Wyślij asynchronicznie, aby nie blokować odpowiedzi
    sendPushNotification(userId, notificationPayload).catch(console.error);
  }
  // ...
};

```

Powiadomienia push znacząco zwiększają zaangażowanie użytkowników i użyteczność aplikacji Budget Manager PWA, informując ich na bieżąco o zmianach w ich budżecie.

4. Zarządzanie stanem aplikacji

4.1 Stores w Pinia

Aplikacja używa Pinia do zarządzania stanem, co ułatwia komunikację między komponentami i zarządzanie danymi aplikacji.

Przykładowy store dla transakcji:

```

export const useTransactionsStore = defineStore("transactions", () => {
  // Stan
  const transactions = ref<Transaction[]>([]);
  const loading = ref(false);
  const error = ref("");

  // API URL
  const apiBaseUrl = "http://localhost:3000/api";

  // Store autoryzacji
  const authStore = useAuthStore();

  // Gettery
  const incomes = computed(() =>
    transactions.value.filter((t) => t.type === "income"),
  );

  const expenses = computed(() =>
    transactions.value.filter((t) => t.type === "expense"),
  );

  const totalIncome = computed(() =>
    incomes.value.reduce((sum, t) => sum + t.amount, 0),
  );
});

```

```

const totalExpenses = computed(() =>
  expenses.value.reduce((sum, t) => sum + t.amount, 0),
);

const balance = computed(() => totalIncome.value - totalExpenses.value);

// Akcje
async function fetchTransactions() {
  if (!authStore.token) return;

  loading.value = true;
  error.value = "";

  try {
    const response = await fetch(`${apiBaseUrl}/transactions`, {
      headers: {
        Authorization: `Bearer ${authStore.token}`,
      },
    });

    if (!response.ok) {
      throw new Error("Failed to fetch transactions");
    }

    const data = await response.json();
    transactions.value = data.data;

    // Zapisywanie do localStorage dla dostępu offline
    saveToLocalStorage();
  } catch (err: any) {
    error.value = err.message;
    // Jeśli API call nie powiedzie się, próba załadowania z localStorage
    loadFromLocalStorage();
  } finally {
    loading.value = false;
  }
}

// ... inne metody

return {
  transactions,
  loading,
  error,
  incomes,
  expenses,
  totalIncome,
  totalExpenses,
  balance,
  fetchTransactions,
  // ... inne eksportowane metody
};
});

```

4.2 Synchronizacja offline/online

Aplikacja wykorzystuje strategię “offline-first”, gdzie dane są najpierw zapisywane lokalnie, a następnie synchronizowane z serwerem gdy połączenie jest dostępne:

```

// Dodawanie transakcji
async function addTransaction(transaction: Omit<Transaction, "_id">) {
  try {
    error.value = "";

    // Generujemy tymczasowe ID dla transakcji
    const tempId = `temp_${Date.now()}`;

```

```

const newTransaction = {
  ...transaction,
  _id: tempId,
  createdAt: new Date().toISOString(),
  updatedAt: new Date().toISOString(),
  synced: false
};

// Dodajemy do listy lokalnie
transactions.value.push(newTransaction);
saveToLocalStorage();

// Próbuje zapisać na serwerze
if (navigator.onLine && authStore.token) {
  const response = await fetch(`${apiBaseUrl}/transactions`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${authStore.token}`,
    },
    body: JSON.stringify(transaction),
  });

  if (response.ok) {
    // Aktualizujemy transakcję z danymi z serwera
    const data = await response.json();
    const index = transactions.value.findIndex(t => t._id === tempId);
    if (index !== -1) {
      transactions.value[index] = { ...data.data, synced: true };
      saveToLocalStorage();
    }
  } else {
    // Oznaczamy transakcję do późniejszej synchronizacji
    const syncStore = useSyncStore();
    syncStore.addPendingTransaction(newTransaction);

    throw new Error("Failed to save transaction to server");
  }
} else {
  // Jesteśmy offline, dodajemy do listy oczekujących na sync
  const syncStore = useSyncStore();
  syncStore.addPendingTransaction(newTransaction);
}
} catch (err: any) {
  error.value = err.message;
  throw err;
}
}

```

5. Interfejs użytkownika

5.1 Komponenty

Aplikacja wykorzystuje komponenty Vue do zbudowania interfejsu użytkownika. Przykładowy komponent Dashboard:

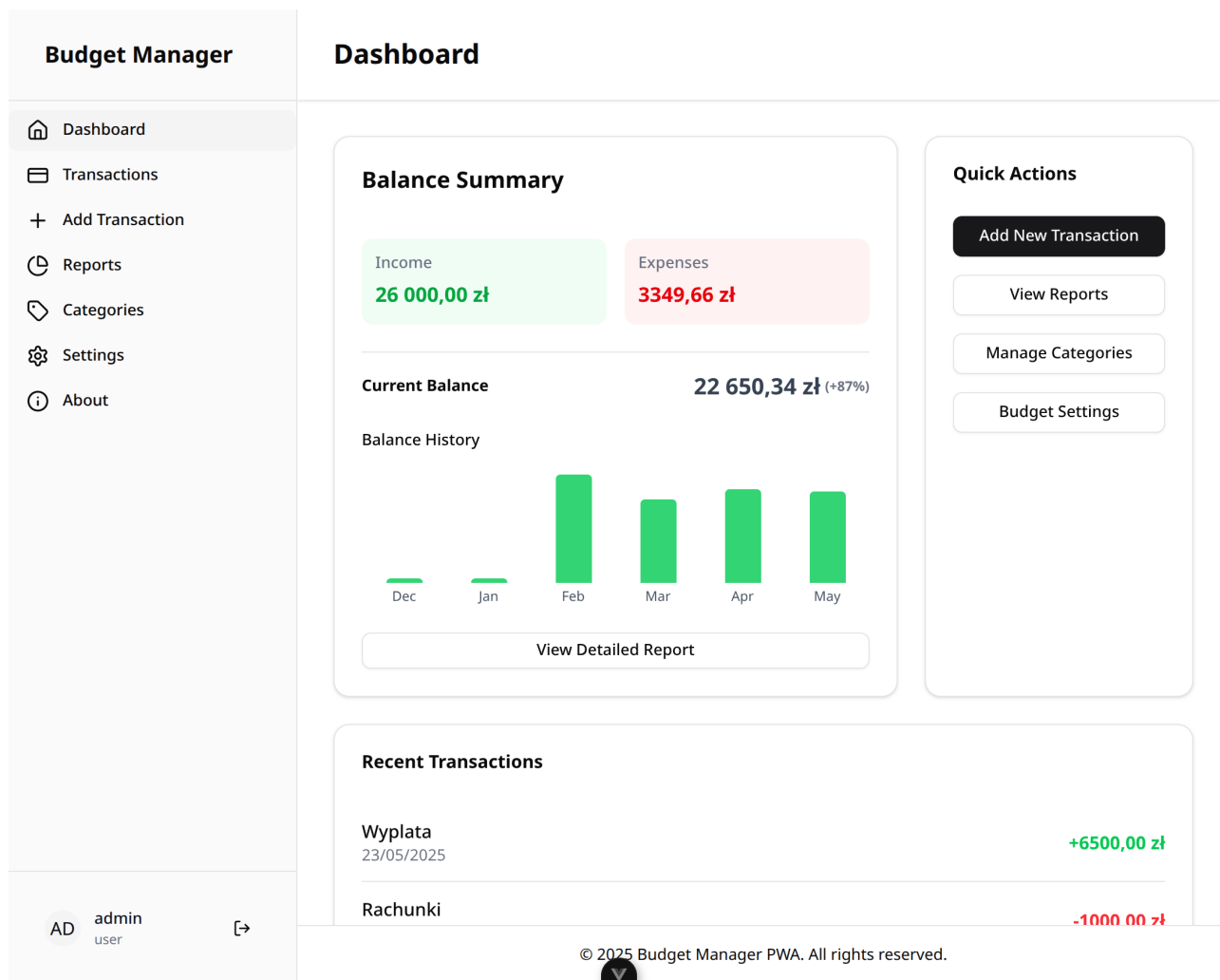


Figure 1: Komponent Dashboard

```
<template>
  <PageLayout header="Dashboard">
    <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
      <!-- Podsumowanie salda -->
      <div class="lg:col-span-2">
        <BalanceSummary />
      </div>

      <!-- Szybkie akcje -->
      <Card>
        <CardHeader>
          <CardTitle>Quick Actions</CardTitle>
        </CardHeader>
        <CardContent class="space-y-4">
          <router-link to="/add" class="w-full block">
            <Button class="w-full" variant="default">Add New Transaction</Button>
          </router-link>
          <router-link to="/reports" class="w-full block">
            <Button class="w-full" variant="outline">View Reports</Button>
          </router-link>
          <!-- Inne przyciski akcji -->
        </CardContent>
      </Card>

      <!-- Ostatnie transakcje -->
      <Card class="lg:col-span-3">
        <CardHeader>
```

```
        <CardTitle>Recent Transactions</CardTitle>
      </CardHeader>
      <CardContent>
        <!-- Zawartość sekcji transakcji -->
      </CardContent>
    </Card>
  </div>
</PageLayout>
</template>
```

5.2 Responsywność

Aplikacja wykorzystuje Tailwind CSS do tworzenia responsywnego interfejsu, który działa dobrze na urządzeniach mobilnych i desktopowych:

≡ Dashboard

Balance Summary

Income

26 000,00 zł

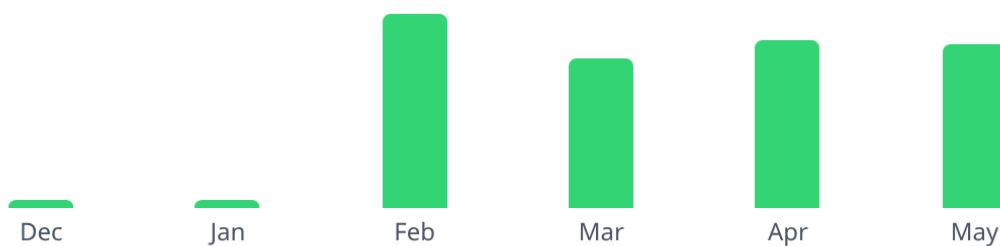
Expenses

3349,66 zł

Current Balance

22 650,34 zł (+87%)

Balance History



[View Detailed Report](#)

Quick Actions

[Add New Transaction](#)

[View Reports](#)

© 2025 Budget Manager PWA. All rights reserved.



Figure 2: Dashboard mobilny

- Używanie systemu grid z breakpointami (md:, lg:)
- Elastyczne komponenty dopasowujące się do różnych rozmiarów ekranów
- Mobilne menu nawigacyjne dla małych ekranów

5.3 Dostępność offline

Interfejs uwzględnia stan połączenia internetowego:

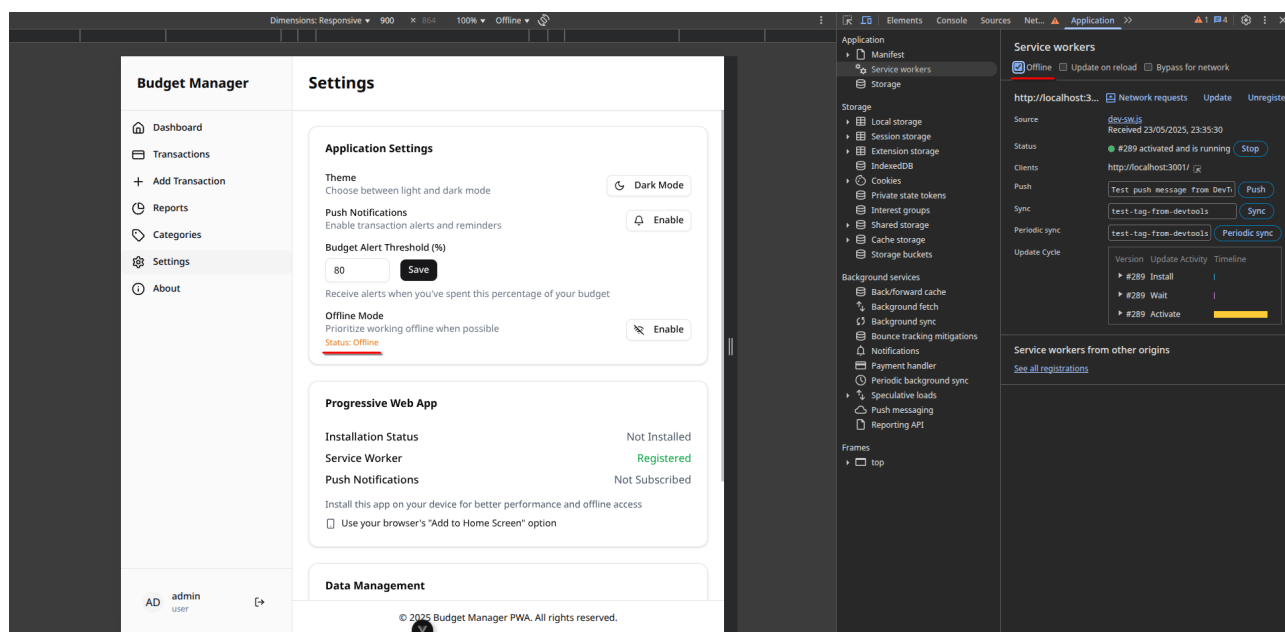


Figure 3: Tryb offline

- Wyświetlanie informacji o trybie offline
- Możliwość dodawania transakcji offline
- Automatyczna synchronizacja po ponownym połączeniu

6. Backend API

6.1 Struktura API

Backend implementuje RESTful API do komunikacji z frontendem:

- POST /api/auth/login - logowanie użytkownika
- POST /api/auth/register - rejestracja nowego użytkownika
- GET /api/transactions - pobieranie listy transakcji
- POST /api/transactions - dodawanie nowej transakcji
- PUT /api/transactions/:id - aktualizacja transakcji
- DELETE /api/transactions/:id - usunięcie transakcji
- GET /api/categories - pobieranie kategorii
- PUT /api/categories - dodawanie nowej kategorii

6.2 Bezpieczeństwo API

API zabezpieczone jest przez:

- Tokeny JWT do autoryzacji
- Middleware uwierzytelniające
- Zabezpieczenie przed CSRF
- CORS

```
// Middleware autoryzacji
export const authenticate = async (ctx: Context, next: Next) => {
  // Pomijamy middleware dla ścieżek publicznych
  if (ctx.path.startsWith('/api/auth/login') ||
    ctx.path.startsWith('/api/auth/register')) {
    return next();
  }

  const token = ctx.headers.authorization?.split(' ')[1];
```

```

    if (!token) {
      ctx.status = 401;
      ctx.body = {
        success: false,
        message: 'No token provided'
      };
      return;
    }

    try {
      const decoded = jwt.verify(token, jwtConfig.secret);
      ctx.state.user = decoded;
      return next();
    } catch (err) {
      ctx.throw(401, 'Invalid or expired token');
    }
  };
};

```

7. Baza danych

7.1 Model danych

Aplikacja wykorzystuje MongoDB do przechowywania danych, z następującymi kolekcjami:

1. users - informacje o użytkownikach
2. transactions - transakcje finansowe
3. categories - kategorie transakcji

7.2 Usługa bazodanowa

Abstrakcja dostępu do bazy danych zapewniona jest przez klasę DatabaseService:

```

export default class DatabaseService {
  private static instance: DatabaseService;
  private client: MongoClient | null = null;
  private db: Db | null = null;

  private constructor() {}

  // Wzorzec Singleton
  public static getInstance(): DatabaseService {
    if (!DatabaseService.instance) {
      DatabaseService.instance = new DatabaseService();
    }
    return DatabaseService.instance;
  }

  // Połączenie z bazą danych
  public async connect() {
    try {
      this.client = await MongoClient.connect(dbConfig.url);
      this.db = this.client.db(dbConfig.name);
      console.log('Connected to MongoDB');
    } catch (error) {
      console.error('Error connecting to MongoDB:', error);
      throw error;
    }
  }

  // Pobranie kolekcji
  public getCollection(collectionName: string): Collection {
    if (!this.db) {
      throw new Error('Database not connected');
    }
    return this.db.collection(collectionName);
  }
}

```

```

// Zamknięcie połączenia
public async disconnect() {
  if (this.client) {
    await this.client.close();
    this.client = null;
    this.db = null;
    console.log('Disconnected from MongoDB');
  }
}
}

```

8. Testy i użycie

8.1 Testowanie offline

Aplikacja została przetestowana w różnych scenariuszach:

- Dodawanie transakcji w trybie offline
- Automatyczna synchronizacja po przywróceniu połączenia
- Działanie Service Worker'a i cache'owanie zasobów

8.2 Instalacja jako PWA

Aplikacja może być zainstalowana na urządzeniach:

- Poprawnie wyświetla prompt instalacyjny
- Działa jako samodzielna aplikacja po zainstalowaniu
- Wykorzystuje ikony i kolory zdefiniowane w manifeście

9. Wnioski i perspektywy rozwoju

9.1 Osiągnięte cele

Aplikacja spełnia wszystkie założenia projektowe:

- Funkcjonuje jako pełnoprawna PWA
- Umożliwia zarządzanie budżetem osobistym
- Działa offline z synchronizacją
- Ma responsywny i przyjazny interfejs

9.2 Możliwe rozszerzenia

Potencjalne kierunki rozwoju aplikacji:

- Dodanie eksportu danych do CSV/PDF
- Implementacja celów oszczędnościowych
- Rozbudowa funkcji raportowania o wykresy
- Dodanie powiadomień o zbliżających się regularnych płatnościach
- Integracja z bankami poprzez API

10. Zrzuty ekranu

10.1 Ekran logowania

Budget Manager → Login ⓘ About GU Guest User Guest ↗	Login Account Login Username admin Password Login Need an account? Register © 2025 Budget Manager PWA. All rights reserved.
---	---

Figure 4: Ekran logowania

10.2 Ekran transakcji

Budget Manager

Dashboard

Transactions

+ Add Transaction

Reports

Categories

Settings

About

AD admin user

→

Transactions

All

Income

Expenses

Transaction History

+ Add New

Date ↑↓	Category ↑↓	Description	Amount ↑↓	Actions
23.05.2025	Work	Wypłata	+6500,00 zł	
10.05.2025	Pudło	Rachunki	-1000,00 zł	
8.04.2025	Pudło	Rachunki	-850,00 zł	
1.04.2025	Work	Wypłata	+6500,00 zł	
13.03.2025	Pudło	Rachunki	-1499,66 zł	
1.03.2025	Work	Wypłata	+6500,00 zł	
1.02.2025	Work	Wypłata	+6500,00 zł	

© 2025 Budget Manager PWA. All rights reserved.

▼

Figure 5: Ekran transakcji

10.3 Ekran kategorii

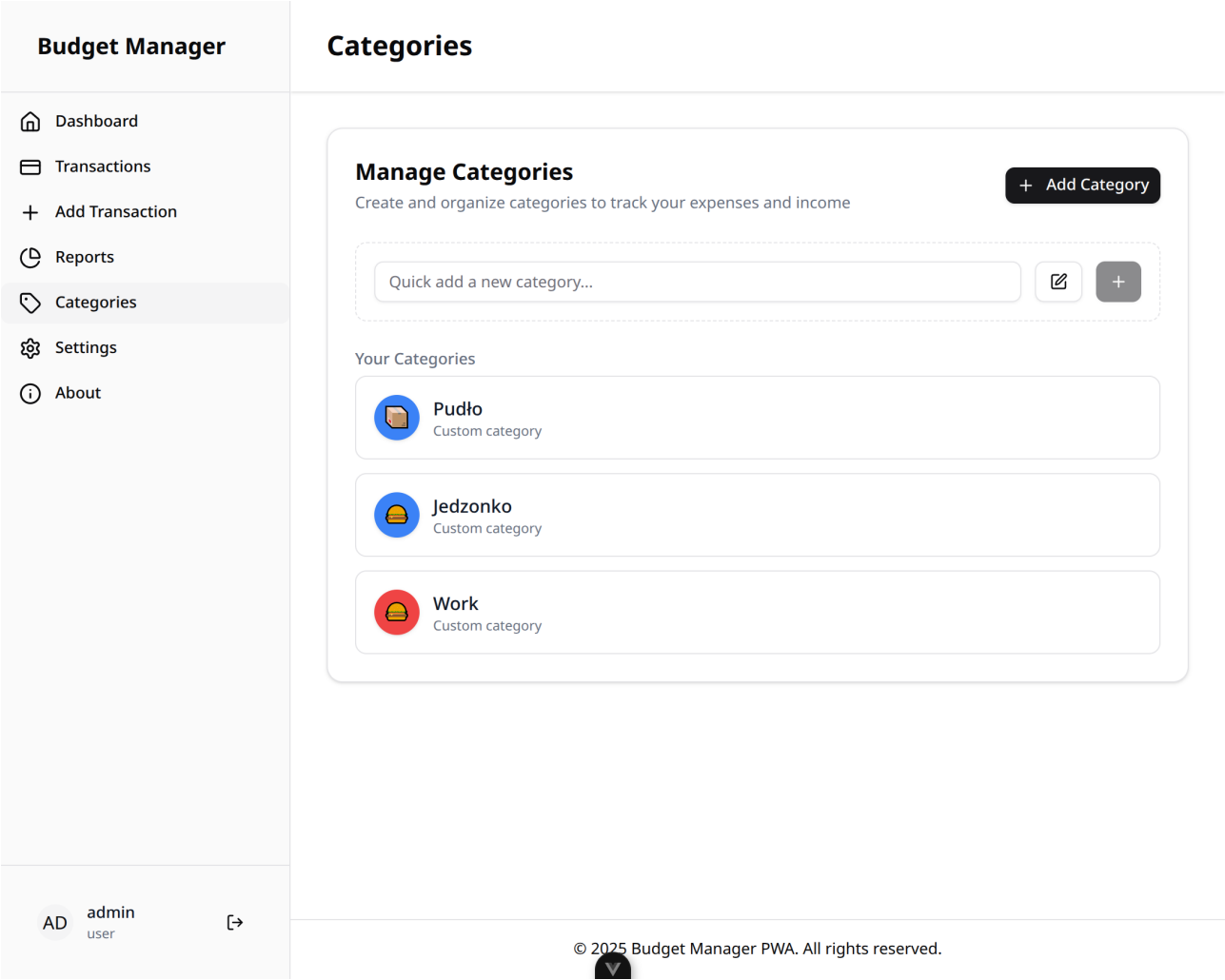


Figure 6: Ekran kategorii

10.4 Ekran raportów

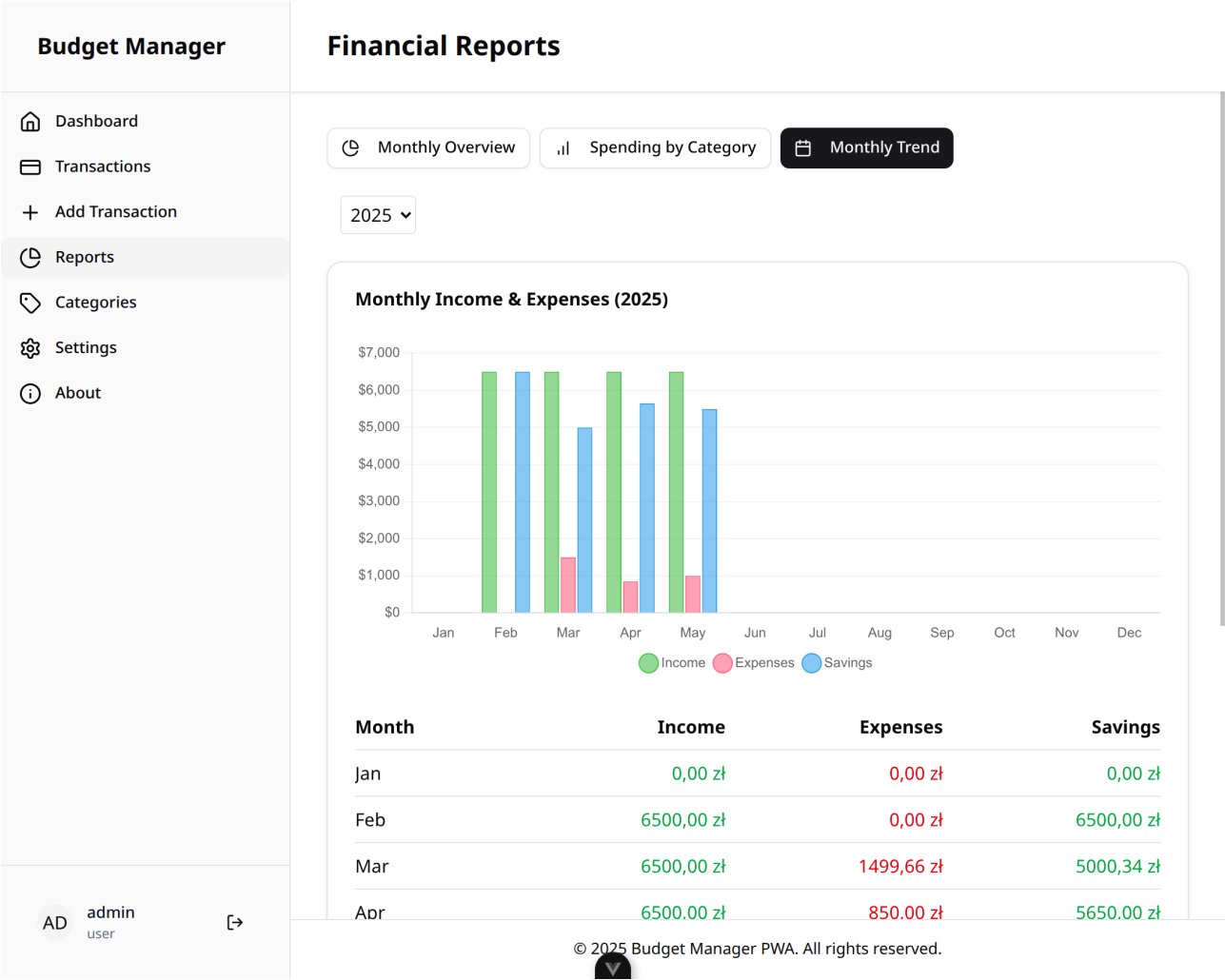


Figure 7: Ekran raportów

10.5 Ekran ustawień

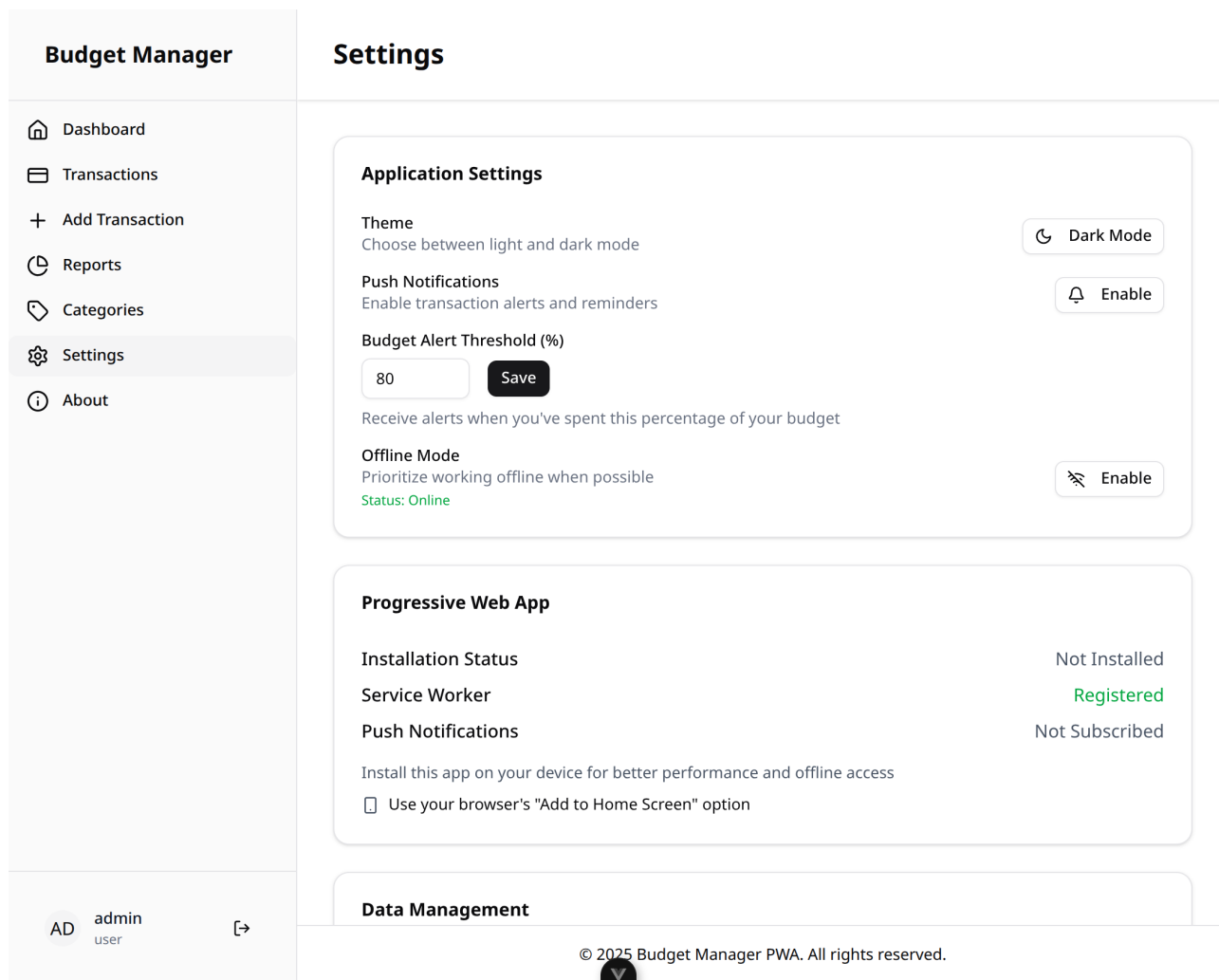


Figure 8: Ekran ustawień

11. Bibliografia

1. Vue.js Documentation: <https://vuejs.org/guide/introduction.html>
2. Progressive Web Apps: <https://web.dev/progressive-web-apps/>
3. MDN Web Docs - Service Worker API: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
4. Tailwind CSS Documentation: <https://tailwindcss.com/docs>
5. Koa.js Documentation: <https://koajs.com/>
6. MongoDB Documentation: <https://docs.mongodb.com/>