Digital Accessibility Centre - Accessibility Scanner Backend

This repository contains the backend services for the DAC's web accessibility scanner application. It's a Node.js-based monorepo that provides an API for initiating scans, a worker for processing them, and a database for storing the results.

Table of Contents

Architecture

The backend is designed as a set of containerized microservices that communicate asynchronously, making it scalable and resilient.

- **API Service (`apps/api`):** A Node.js/Express.js application that serves as the public-facing entry point. It receives scan requests from the frontend, adds them to a job queue, and provides endpoints to retrieve scan results.

- **Worker Service (`apps/worker`):** A Node.js application that listens for jobs on the queue. It performs the heavy lifting: launching a headless browser (Puppeteer) to visit the target URL, running `axe-core` to perform the accessibility audit, taking a screenshot, and saving the results to the database.

- **Redis / BullMQ:** A Redis-backed message queue (BullMQ) that decouples the API from the Worker. This prevents API requests from timing out while waiting for a long-running scan and allows for job retries and queuing.

- **PostgreSQL Database:** A relational database used to persist all scan results. The schema is managed via migrations.

- **Docker:** All services are containerized with Docker, and `docker-compose.yml` orchestrates the local development environment.

Request Flow

1. A user submits a URL to the frontend (SvelteKit app).
2. The frontend sends a `POST` request to the **API Service** (`/api/scan-website`).
3. The **API Service** generates a unique `jobId` and enqueues a new job in **BullMQ**. It immediately returns the `jobId` to the frontend.
4. The frontend begins polling the **API Service** using the `jobId` (`/api/scan-results/:jobId`).
5. The **Worker Service**, listening to the queue, picks up the job.
6. The **Worker** uses Puppeteer and `axe-core` to scan the site, takes a screenshot, and saves the results (or any errors) to the **PostgreSQL** database, referencing the `jobId`.
7. On a subsequent poll, the **API Service** finds the completed job results in the database and returns them to the frontend.
8. The frontend displays the results to the user.

Technology Stack

- **Backend:** Node.js, TypeScript, Express.js
- **Database:** PostgreSQL
- **Queue:** Redis, BullMQ
- **Browser Automation:** Puppeteer
- **Accessibility Engine:** `axe-core`
- **Containerization:** Docker, Docker Compose
- **Package Manager:** pnpm (in a monorepo setup)
- **Migrations:** `node-pg-migrate`

Project Structure

This is a `pnpm` monorepo. The main components are:

```
.
├── apps/
│   ├── api/            # Express API service
│   └── worker/         # BullMQ worker service
├── migrations/         # node-pg-migrate database migration files
├── docker-compose.yml # Local development environment setup
├── package.json        # Root package.json
└── pnpm-workspace.yaml # Defines the pnpm workspace
```

Setup and Installation

Prerequisites

- Docker
- Docker Compose
- Node.js (v18 or later)
- pnpm (`npm install -g pnpm`)

Running Locally

1. **Clone the repository:**

```
git clone <repository-url>
cd accessibility-scanner-backend
```

2. **Install dependencies:** This command installs dependencies for all workspace packages (`api` and `worker`).

```
pnpm install
```

3. **Create environment file:** Copy the example environment file and fill in any necessary values (the defaults in `docker-compose.yml` are usually sufficient for local development).

```
# This project does not currently use a .env file, relying on docker-compose.yml for environment variables.
# If a .env file is added, it should be created from a .env.example.
```

4. **Start the services:** This command will build the Docker images (if they don't exist) and start all services (api, worker, postgres, redis).

```
docker compose up --build
```

The API will be available at `http://localhost:3000`.

Environment Variables

Environment variables are defined in `docker-compose.yml` for local development. Key variables include:

- `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB`: Credentials for the PostgreSQL database.
- `POSTGRES_HOST`, `POSTGRES_PORT`: Connection details for PostgreSQL (e.g., `postgres`, `5432`).
- `REDIS_HOST`, `REDIS_PORT`: Connection details for Redis (e.g., `redis`, `6379`).
- `DATABASE_URL`: The full connection string for the database, required by `node-pg-migrate`. Format: `postgres://<user>:<password>@<host>:<port>/<db>`

Database Migrations

Database schema changes are managed with `node-pg-migrate`.

- **To run migrations (apply new changes):** The `migrate:up` script is defined in the root `package.json`. It executes the migrations inside the running `api` container.

```
pnpm -w run migrate:up
```

- **To create a new migration:**

  ```
  pnpm -w run migrate:create <migration-name>
  ```

API Endpoints

All endpoints are prefixed with `/api`.

`POST /api/scan-website`

Initiates a new accessibility scan for a given URL.

- **Request Body:**

  ```
  {
    "url": "https://example.com"
  }
  ```

- **Success Response (202 Accepted):**

  ```
  {
    "message": "Scan request accepted and enqueued. Job ID: <uuid>",
    "jobId": "<uuid>",
    "submittedUrl": "https://example.com"
  }
  ```

- **Error Response (400 Bad Request):** If the URL is invalid.

`GET /api/scan-results/:jobId`

Retrieves the results of a scan. This endpoint is designed to be polled by the frontend until the scan is complete.

- **URL Parameters:**
  - `jobId` (string, required): The ID returned from the `POST /api/scan-website` call.
- **Success Response (200 OK):** Returns the full scan result object from the database once the job is complete.

  ```
  {
      "id": 1,
      "job_id": "<uuid>",
      "original_job_id": "<uuid>",
      "submitted_url": "https://example.com",
  ```

```
    "actual_url": "https://example.com/",
    "scan_timestamp": "2023-10-27T10:00:00.000Z",
    "page_title": "Example Domain",
    "scan_success": true,
    "violations": [
        // Array of axe-core violation objects
    ],
    "error_message": null,
    "page_screenshot": "data:image/jpeg;base64,..."
}
```

- **Pending Response (404 Not Found):** If the job is still processing or does not exist, a 404 is returned, and the client should continue polling.

`GET /api/export-report/:jobId`

Generates and returns a full HTML report for a completed scan.

- **URL Parameters:**
  - `jobId` (string, required): The ID of a completed scan.
- **Success Response (200 OK):**
  - **Content-Type:** `text/html`
  - The response body is a full HTML document containing the formatted scan report, including summary metrics, a screenshot, and a detailed list of all violations.

Key Logic

Scan Processing

The worker (`apps/worker/src/index.ts`) contains the core scanning logic:

1. A job is received from the BullMQ queue.
2. Puppeteer launches a headless Chromium browser.
3. The browser navigates to the `submittedUrl`.
4. The `axe-core` script is injected into the page.
5. `axe.run()` is executed to get the accessibility violations.
6. `page.screenshot()` is called to capture a base64-encoded screenshot of the page.
7. The results, including the page title, final URL, violations array, and screenshot, are saved to the `scan_results` table in the PostgreSQL database.
8. If any step fails, the error is caught, and the `error_message` and `scan_success: false` are recorded in the database.

Accessibility Scoring

The accessibility score is not calculated on the backend but is derived on the **frontend** (`DigitalAccessibilityCentre/src/routes/scan-results/+page.svelte`) for display purposes. The logic is as follows:

- A starting score of 100 is assumed.

- Points are deducted for each violation based on its impact level and the number of nodes it affects:
  - `critical`: 10 points per failing node.
  - `serious`: 5 points per failing node.
  - `moderate`: 2 points per failing node.
  - `minor`: 1 point per failing node.
- The final score is capped between 0 and 100.