ScanTotal

This is a bare bones VirusTotal clone for my Distributed Systems class.

The core idea consists of three main components:

- A Web Frontend where:
  - A user can upload a file and see the scan results
  - Query the scan results based on metadata (e.g. hash)
- A Backend API that:
  - Receives the file
  - Scans it with (multiple) antivirus engines, such as:
    - ClamAV (for general file scanning)
    - capa (for executable files)
    - oletools (for MS Office files)
    - pdfid (for PDF files)
    - etc.
  - Returns the results to the frontend.
- A Database to:
  - Store the scan results and metadata about the files (like hash, file name, upload date, etc.)

Quick Start

1. Clone the repository:

```
git clone https://github.com/tillstuder/DSy-FS26.git
cd DSy-FS26
```

2. Start the system with Docker Compose:

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

3. Access the frontend at `http://localhost:80` and start uploading files to see the scan results.

> **Note**
>
> To test if the system is working, you can upload the EICAR test file which is used to test antivirus software.

> **Note**
>
> As some of the dependencies (like ClamAV) only work on x86_64 architecture, the system is currently only tested on that architecture. It may not work on ARM-based systems (like Apple Silicon) without modifications.

Debugging and Development

To reset the system (e.g., to clear the database and uploaded files):

```
docker compose down -v
```

Architecture and Design

This section uses the C4 model to describe ScanTotal at three levels: system context, containers, and components.

C1: System Context

System: ScanTotal is a VirusTotal-like web application for authenticated file uploads, malware scanning, and scan result lookup.

Primary actor:

- User: logs in, uploads files, and retrieves scan results through the browser.

External systems:

- ClamAV: antivirus engine used by workers to scan uploaded files.

System context:

Element	Relationship to ScanTotal
User	Uses the web UI to authenticate, upload files, and view scan results
ScanTotal	Accepts files, queues scans, stores metadata/results, and exposes results via HTTP API
ClamAV	Receives files from the worker process and returns scan findings

C2: Container Diagram

ScanTotal is split into a small set of deployable containers with clear runtime responsibilities:

Container	Technology	Responsibility
Traefik	Traefik	Entry point on port 80; routes `/` to the frontend and `/api`, `/health` to the backend service
Frontend	Vanilla JavaScript + Nginx	Serves the SPA assets to the browser
Backend API	Go HTTP service (2 instances)	Authenticates users, stores uploaded files, creates scan jobs, and serves scan results
Worker	Go background process	Claims pending jobs from PostgreSQL and runs scanners asynchronously
PostgreSQL	PostgreSQL	Persists users, files, scan jobs, findings, and summaries
Uploads Volume	Docker named volume	Shared storage for uploaded files used by backend, worker, and ClamAV
ClamAV	ClamAV daemon	Performs malware scanning for worker-executed jobs

Container relationships:

1. User interacts with the Frontend in the browser.
2. The browser reaches both the frontend and the API through Traefik on port 80.
3. Traefik routes `/` to the frontend container and distributes `/api` and `/health` traffic across `backend-1` and `backend-2`.
4. Backend API writes uploaded files to the shared `uploads` volume and persists metadata, jobs, and results to PostgreSQL.
5. Worker polls PostgreSQL for `PENDING` jobs using `SELECT FOR UPDATE SKIP LOCKED`.
6. Worker reads files from the shared `uploads` volume and requests scans from ClamAV over `clamav:3310`.
7. Worker persists findings back to PostgreSQL, and the browser continues polling the API through Traefik until the job reaches a terminal state.

C3: Component Diagram (Backend)

Within the Go backend, the main components are:

Component	Files	Responsibility
Router	`backend/internal/handlers/router.go`	Registers routes and applies JWT protection
Auth Handler	`backend/internal/handlers/auth.go`	Handles login and token issuance
Upload Handler	`backend/internal/handlers/upload.go`	Accepts multipart uploads, computes SHA-256, stores files, and creates scan jobs
Scan Handler	`backend/internal/handlers/scan.go`	Returns scan status, summaries, and findings
Auth Module	`backend/internal/auth/auth.go`	Generates and validates JWTs
DB Layer	`backend/internal/db/connect.go`	Creates the PostgreSQL connection pool

Component	Files	Responsibility
Worker Job Logic	`backend/internal/worker/job.go`	Claims jobs, updates job state, and stores findings
Scanner Adapter	`backend/internal/scanner/clamav.go`	Encapsulates ClamAV scan execution

Runtime Flow

1. A user logs in and receives a JWT from `POST /api/v1/auth/login`.
2. The frontend uploads a file to `POST /api/v1/files`.
3. The backend computes the SHA-256 hash, stores metadata, and inserts a `PENDING` job.
4. A worker claims the job from PostgreSQL, scans the file with ClamAV, and stores findings.
5. The frontend polls `GET /api/v1/scans/{id}` until the backend reports `COMPLETED` or `FAILED`.

Project Requirements

1. Load balancing with multiple instances with failover of a service instance
2. Containerized (e.g., with docker)
3. Simple frontend
4. JWT-based authentication
5. Persistent storage
6. Use latest stable releases of chosen libraries and frameworks
7. A fresh clone of the repository + a single command (e.g., docker compose up) must bring up the entire working system, including database seeding/migration if needed. No manual setup steps allowed.
8. Perform a load test against a JWT-protected endpoint of your system (e.g., using ab, wrk, hey, or k6) and document the results. During the presentation, show how your system behaves under load, what the bottleneck could be, and how many requests/second it can handle.

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Contributors 1

tillstuder Till Studer

Languages

Go 57.9% **JavaScript** 23.4% **CSS** 8.2% **HTML** 5.0% **Shell** 4.3% **Dockerfile** 1.2%