# URL Shortener Web Service

## 1. Overview

This section documents the backend module responsible for URL shortening, redirection, and URL listing within the team's containerized URL Shortener project. The service is implemented using **Node.js**, **Express**, **EJS**, and **SQLite**, and provides the core API and frontend rendering logic used by the system. It operates as part of the overall containerized environment that includes monitoring and visualization tools.

---

## 2. Application Structure

```
/project
├── server.js          // Express web server
├── db.js              // SQLite database connection
├── views/
│   ├── index.ejs      // URL submission page
│   └── routes.ejs     // Display all shortened URLs
├── public/
    └── styles/        // CSS files
```

---

## 3. Server Initialization

The application initializes by loading dependencies and preparing the Express server:

Configuration includes:

- **express** provides routing and HTTP handling
- **cors** allows cross-origin requests
- **path** resolves directory paths
- **body-parser** processes URL-encoded form data
- **db** handles SQLite operations

```
const express = require("express");
const cors = require("cors");
const path = require("path");
const bodyParser = require("body-parser");
const db = require("./db");
require("dotenv").config();

const app = express();
const port = process.env.PORT || 8000;
```

## 4. Middleware & View Engine Configuration

```
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
app.use(express.static(path.join(__dirname, "public"
)));
app.use(bodyParser.urlencoded({ extended: true }));
app.use(cors());
```

Configuration includes:

- **EJS** as the templating engine
- Static assets served from `/public`
- Form handling for POST requests
- CORS enabled for local testing

## 5. Implemented Routes

### 5.1 Home Page

```
app.get("/", (req, res) => {
  res.render("index.ejs");
});
```

Renders the main interface where users can submit URLs for shortening.

## 5.2 Display All Stored URLs

- Retrieves all entries in the `urls` table
- Passes them to the `routes.ejs` template for table rendering

```
app.get("/routes", (req, res) => {
  db.all("SELECT * FROM urls", [], (err, rows) => {
    if (err) return res.status(500).send(
"Database error");
    res.render("routes.ejs", { links: rows });
  });
});
```

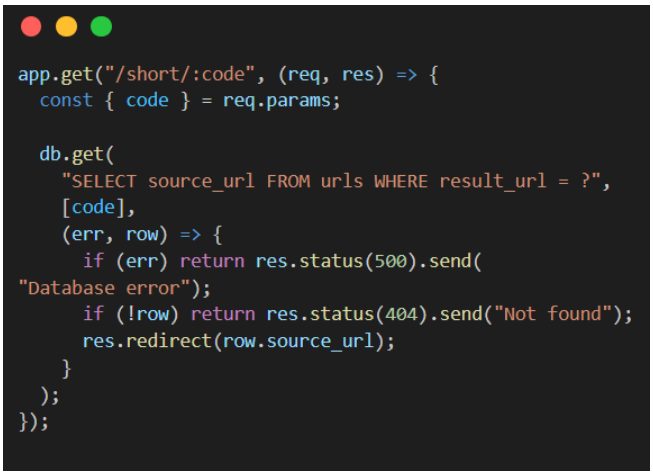
## 5.3 Handle URL Shortening

Functionality:

```
app.post("/short", (req, res) => {
  const { url } = req.body;
  if (!url) return res.status(400).send("Missing URL");

  const shortCode = Math.random().toString(36).substring(
2, 8);

  db.run(

"INSERT INTO urls (source_url, result_url) VALUES (?, ?)"
,
    [url, shortCode],
    function (err) {
      if (err) return res.status(500).send(
"Failed to save");
      res.redirect("/routes");
    }
  );
});
```

- Validates user input
- Generates a 6-character alphanumeric short code
- Stores the original URL and its short version in the SQLite database
- Refreshes the routes page after insertion

### 5.4 Redirecting Using the Shortened Code



```
app.get("/short/:code", (req, res) => {
  const { code } = req.params;

  db.get(
    "SELECT source_url FROM urls WHERE result_url = ?",
    [code],
    (err, row) => {
      if (err) return res.status(500).send(
"Database error");
      if (!row) return res.status(404).send("Not found");
      res.redirect(row.source_url);
    }
  );
});
```

This route:

- Accepts a short code
- Looks up the corresponding original URL
- Redirects the user if found
- Returns a 404 page if the code does not exist

---

# 6. Frontend Templates (EJS)

### 6.1 index.ejs

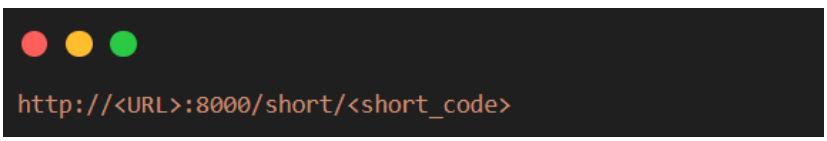
Provides:

- A simple form for entering a long URL
- A button to trigger URL shortening
- A link to view all saved URLs

### 6.2 routes.ejs

Responsible for:

- Displaying all stored URLs
- Showing both the original and shortened versions
- Generating clickable short links such as:



```
http://<URL>:8000/short/<short_code>
```

## 7. Database Layer

The SQLite database stores links in a minimal schema:

```
id INTEGER PRIMARY KEY AUTOINCREMENT,
source_url TEXT,
result_url TEXT
```

This enables lightweight, file-based persistence suitable for local and containerized environments.

# 8. Dockerfile Configuration

The Dockerfile defines the environment and build instructions required to run the URL Shortener backend inside a containerized production environment. It ensures that the Node.js service runs efficiently, securely, and with persistent storage for the SQLite database.

The complete Dockerfile used in this project is shown below:

## 8.1 Purpose of the Dockerfile

This Dockerfile prepares a lightweight, production-ready container image that runs the URL Shortener service with Node.js 20 on Alpine Linux. It installs dependencies, copies the

application code, configures environment variables, and defines the command executed when the container starts.

## 8.2 Breakdown of the Dockerfile Instructions

### Base Image

```
1   FROM node:20-alpine
```

Uses a lightweight Node.js 20 image built on Alpine Linux. This reduces image size and improves performance in production deployments.

### Working Directory

```
3   WORKDIR /app
```

Specifies that all operations inside the container will take place in the `/app` directory.

### Copy Dependencies

```
5   COPY package*.json ./
6
7   RUN npm install --production
```

Only dependency files are copied first. Installing production dependencies at this stage ensures:

- Faster rebuilds (leveraging Docker layer caching)
- Smaller final image size

- No unused development packages inside the container

### Copy Application Files

```
9    COPY . .
```

Copies the entire backend source code (Express server, database layer, views, public files) into the container.

### Create Persistent Data Directory

```
10   RUN mkdir -p /app/data && chown -R node:node /app
```

Creates a directory for the SQLite database file and assigns ownership to the unprivileged `node` user.
This supports:

- Secure container execution
- Compatibility with Docker volumes for data persistence

### Run the Application as a Non-Root User

```
USER node
```

Enhances security by preventing the service from running with root privileges.

### Environment Variables

```
ENV DB_PATH=/app/data/database.sqlite
ENV NODE_ENV=production
```

Defines:

- The location where SQLite will store the database file
- Production mode for optimized server execution

### Expose Application Port

```
EXPOSE 8000
```

Documents the service port used by the Express server so that Docker Compose can map it externally.

### Application Startup Command

```
CMD ["npm", "start"]
```

This command launches the backend service when the container is run.
It automatically executes the start script defined in `package.json`.

# 9. Docker Compose Configuration

This section documents the Docker Compose configuration responsible for orchestrating the three main services of the project: the URL Shortener application, Prometheus monitoring, and Grafana visualization. The `docker-compose.yml` file defines how these containers interact, how networking and data persistence are handled, and how the full environment is launched as a unified system.

## 9.1 Overview

The Compose setup launches a multi-container environment consisting of:

- **app** — The Node.js URL Shortener backend
- **prometheus** — The monitoring system that scrapes service metrics
- **grafana** — The dashboard and alerting platform

Each service runs inside its own container while sharing networks and volumes as needed for storage and communication.

## 9.2 Service Definitions

### 1) Application Service (app)

```yaml
app:
  build: .
  container_name: depi_app
  restart: unless-stopped
  ports:
    - "8000:8000"
  environment:
    - NODE_ENV=production
    - DB_PATH=/app/data/database.sqlite
  volumes:
    - app_data:/app/data
```

**Function:**
Runs the main URL Shortener Node.js server.

**Key configurations:**

- **build:** Builds the image using the local Dockerfile.
- **restart: unless-stopped:** Ensures the service automatically restarts unless manually stopped.
- **ports:** Maps port 8000 inside the container to port 8000 on the host machine.
- **environment:** Passes environment variables required for production mode and database path.
- **volumes:** Mounts a persistent volume (`app_data`) to store the SQLite database file.

### 2) Prometheus Monitoring Service

```yaml
prometheus:
  image: prom/prometheus:v2.47.2
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
  command:
    - "--config.file=/etc/prometheus/prometheus.yml"
  depends_on:
    - app
```

**Function:**
Scrapes metrics exposed by the backend service and provides a monitoring interface accessible at port 9090.

**Key configurations:**

- **image:** Uses the official Prometheus image.
- **volumes:** Mounts the custom Prometheus configuration file from the project directory.
- **command:** Ensures Prometheus uses the correct config path.
- **depends_on:** Ensures that Prometheus only starts after the main `app` service is running.

### 3) Grafana Visualization Service

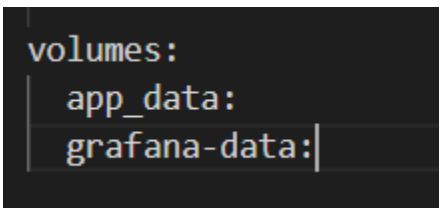
```yaml
grafana:
  image: grafana/grafana:latest
  container_name: grafana
  ports:
    - "3000:3000"
  depends_on:
    - prometheus
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
    - GF_USERS_ALLOW_SIGN_UP=false
  volumes:
    - grafana-data:/var/lib/grafana
    - ./grafana/provisioning:/etc/grafana/provisioning
```

**Function:**
Provides dashboards, metrics visualization, and alerting through a web interface.

**Key configurations:**

- **ports:** Exposes Grafana on port 3000.
- **environment:** Sets predefined admin credentials and disables user sign-up.
- **volumes:**
    - Stores Grafana dashboards and alert settings persistently
    - Loads provisioning files automatically (datasources, alert rules, dashboards)
- **depends_on:** Ensures Grafana starts only after Prometheus is available.

### 9.3 Defined Volumes




These volumes ensure that important data remains intact even if containers are recreated:

- **app_data:** Stores SQLite database used by the URL Shortener
- **grafana-data:** Stores Grafana dashboards, users, and settings

---

# Observability and Monitoring Module Documentation

## . Overview

To ensure high availability and performance reliability for the URL Shortener Web Service, an observability layer has been implemented using **Prometheus**. This module allows for real-time tracking of application usage, error rates, and request latency, transforming the application from a "black box" into a measurable, observable system.

## . Objectives

The primary goals of this implementation are:

- **Traffic Analysis:** Tracking the number of shortened URLs and successful redirects.
- **Error Detection:** Monitoring failed lookups (404 errors) to identify broken links or misuse.
- **Performance Monitoring:** Measuring request latency (response time) for critical operations.
- **Container Integration:** Ensuring seamless integration within the Docker ecosystem.

---

# . Architecture & Tech Stack

- **Application:** Node.js (Express) with SQLite database.
- **Instrumentation Library:** `prom-client` (Default metrics + Custom metrics).
- **Monitoring System:** Prometheus (Time-series database).
- **Orchestration:** Docker Compose.

---

# . Implementation Details

## . Application Instrumentation (index.js)

- The web service was modified to expose a `/metrics` endpoint. We utilized the `prom-client` library to register custom metrics.

**Key Metrics Defined:**

1. `webservice_urls_shortened_total` **(Counter):** Increments when a new short URL is created.
2. `webservice_redirects_total` **(Counter):** Increments when a user is successfully redirected.
3. `webservice_failed_lookups_total` **(Counter):** Increments when a short code is not found (404).
4. `webservice_request_latency_seconds` **(Histogram):** Tracks the duration of HTTP requests across different routes.

**Code Snippet (Metric Logic):**

```
JavaScript

// Prometheus Setup

const client = require("prom-client");

const register = new client.Registry();

client.collectDefaultMetrics({ register });
```

```
// Exposing the /metrics endpoint
app.get("/metrics", async (req, res) => {
  res.set("Content-Type", register.contentType);
  res.end(await register.metrics());
});
```

### . Prometheus Configuration (`prometheus.yml`)

A scraper configuration was created to pull data from the application every 15 seconds.

```
YAML
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: "webservice"
    static_configs:
      - targets: ["app:8000"]
```

---

# . Infrastructure & Deployment

## . Dockerization

- The `Dockerfile` was optimized to support the building of native dependencies (`sqlite3`) by including Python and build tools in the Alpine image.

## . Service Orchestration

- The `docker-compose.yml` file was updated to run the **Application** and **Prometheus** as coupled services in the same network.
  **App Service:** Runs on port `8000`.

**Prometheus Service:** Runs on port 9090 and depends on the App service.

---

# . Verification & Visualization

## . Service Health Check

- Upon deployment, the Prometheus target status confirms that the scraper is successfully connecting to the application container.

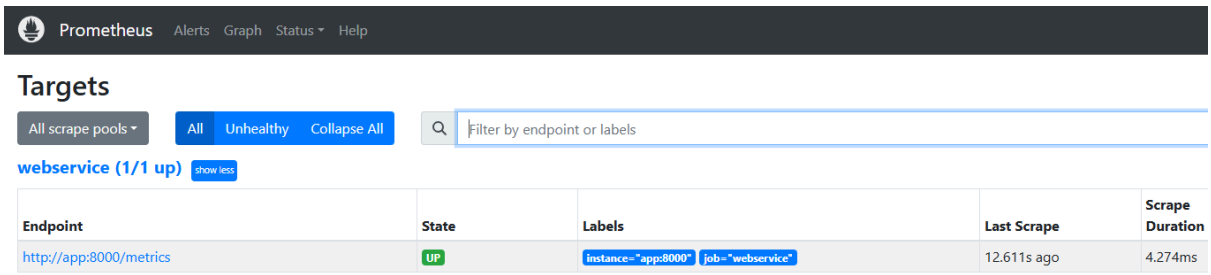



Figure 1: Prometheus Targets page showing the Webservice is UP.

## . Metric Visualization

- We verified the data collection by performing operations on the web service (creating links) and observing the counters incrementing in real-time within the Prometheus graph interface.

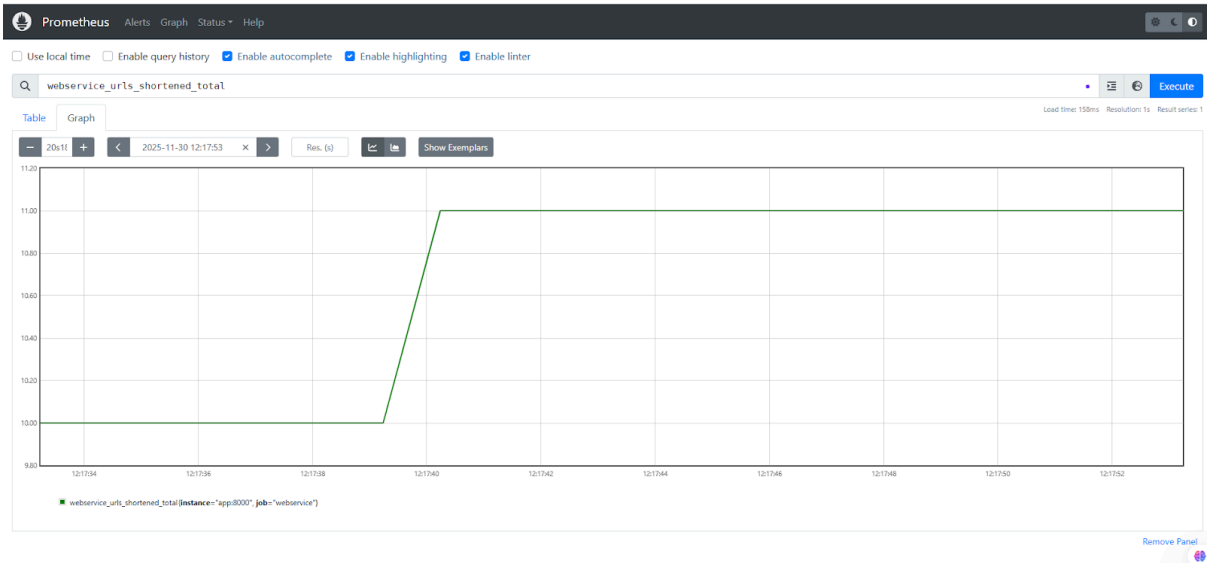



Figure 2: 'webservice_urls_shortened_total' metric showing growth in shortened links.

# Grafana  provisioning

- ## Overview

  Grafana supports an automated configuration system called Provisioning, which allows you to load datasources, dashboards, and alerting rules automatically when Grafana starts. This is especially useful when using Docker, Kubernetes, or CI/CD pipelines.

- ## Docker Compose Update


To enable Grafana and Prometheus to run together, you must update your docker-compose.yml file to include services for Grafana and Prometheus. Ensure that: - The Prometheus container exposes port 9090. - The Grafana container exposes port 3000. - Both services are connected to the same network. Grafana's provisioning folder is correctly mounted.

- ## Datasource Provisioning

Datasource provisioning automatically pre-configures datasources (like Prometheus, Loki, InfluxDB, MySQL, Elasticsearch, etc.) using YAML files stored inside the provisioning/datasources directory.

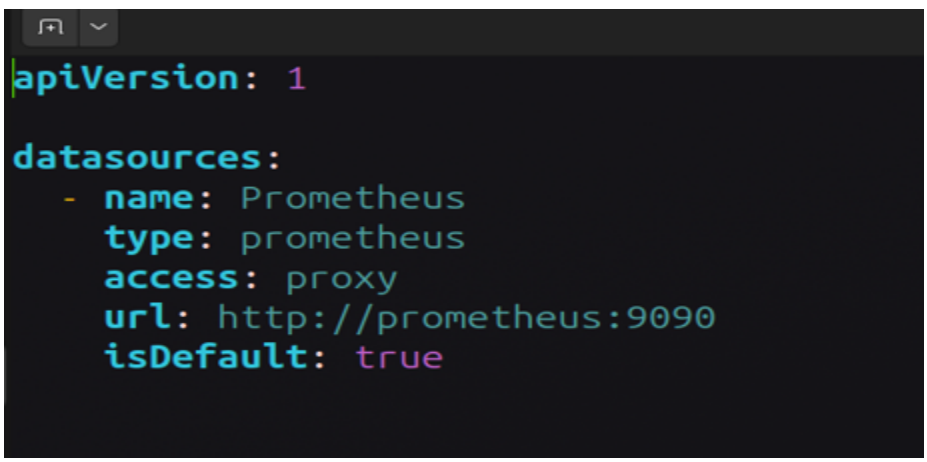
- **Folder Structure**

```
grafana/
  provisioning/
    dashboards/
      dashboard.yml
```

```
      url_shortner.json
    datasources/
      datasource.yml
```

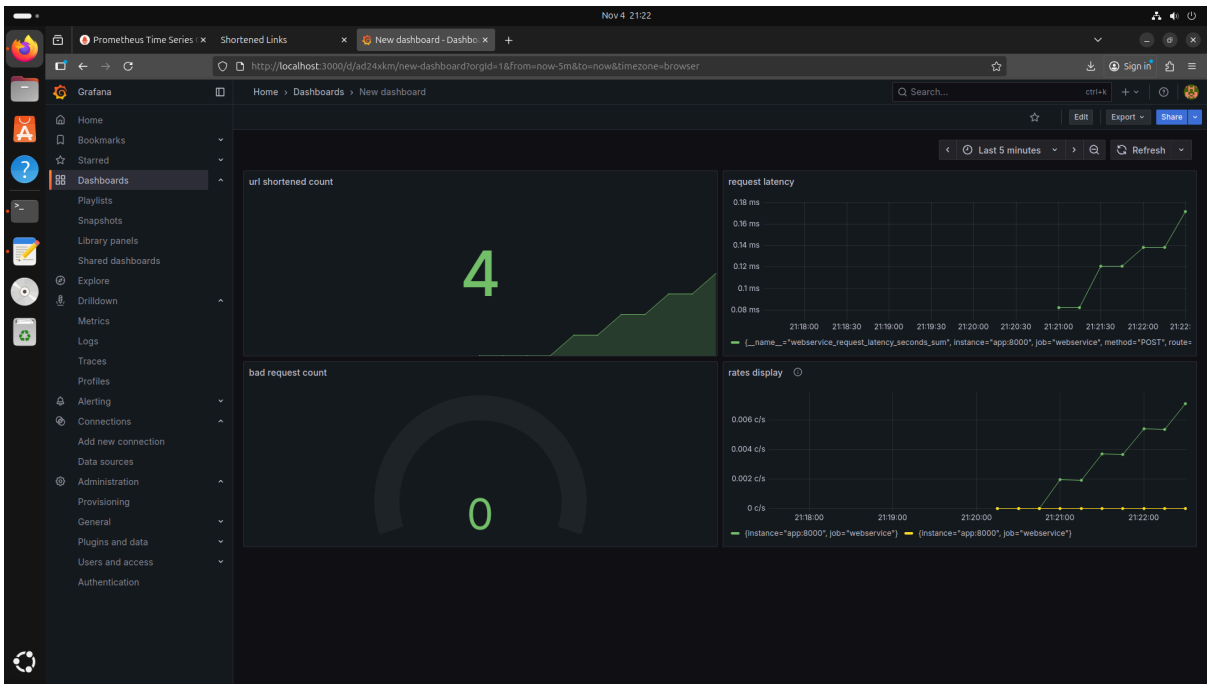
**Purpose of Each File:**

- dashboard.yml → Points Grafana to JSON dashboard files.
- url_shortner.json → Contains the full exported dashboard definition
- datasource.yml → Automatically configures the Prometheus datasource.



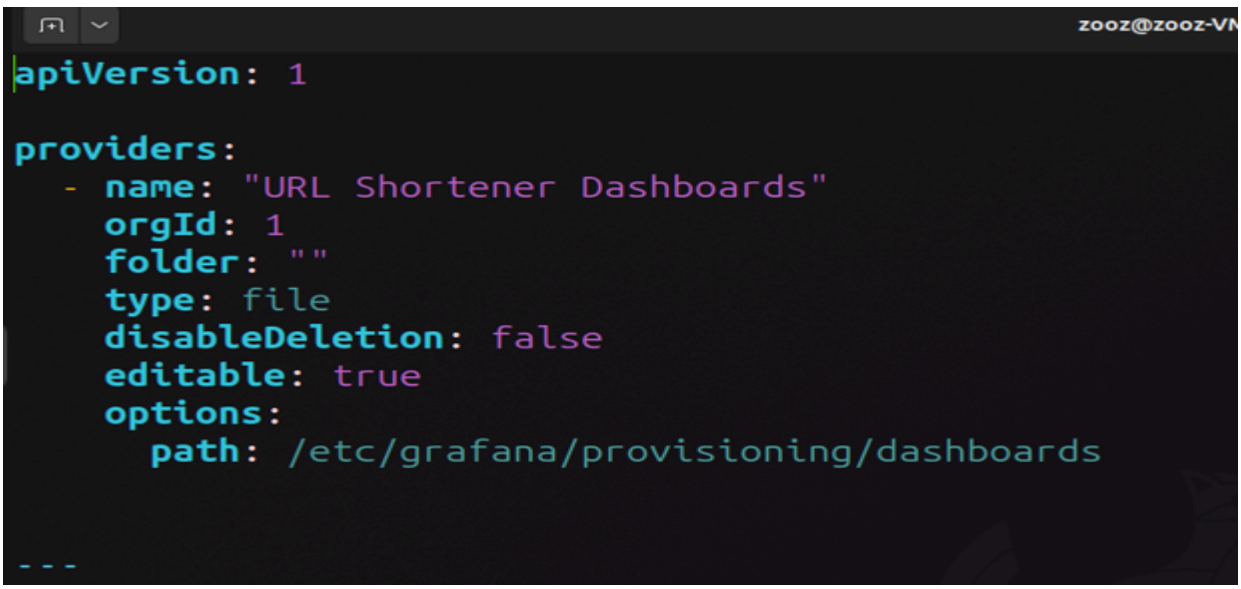
```yaml
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
    isDefault: true
```

- # Dashboard Auto-Provisioning




Use this section to automatically load dashboards into Grafana at startup without manually importing them through the UI.

1. Create the Folder Structure
2. Export Your Dashboard JSON From Grafana -> your_dashboard.json inside the dashboards folder.
3. Create the Dashboard Provisioning File-> Inside:grafana/provisioning/dashboards/dashboard.yml

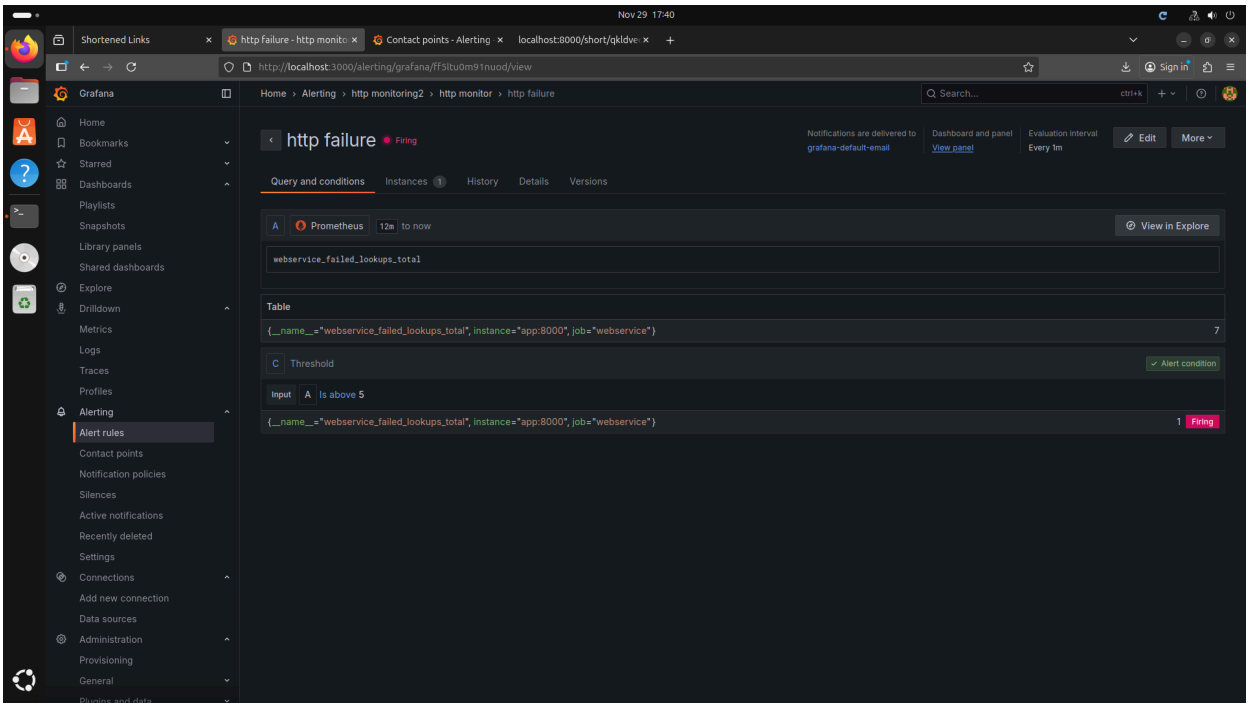


# Adding Alert rule

To configure alerts in Grafana using Prometheus metrics

### Steps

1. Open your dashboard.
2. Select a panel to attach an alert.
3. Click Alert → Create Alert Rule.
4. Choose a metric—for example: http_requests_total{status="404"} .
5. Define a condition, such as: If 404 rate > 5 for 5 minutes.
6. Set the notification channel (Email, Slack, etc.)




then export the alert rule as a yml file and put it in following path

provisioning/alerting/alert-rule.yml

**finally add the following bind mount**


- It maps your local folder ./grafana/provisioning to the container folder /etc/grafana/provisioning.
- Grafana reads provisioning files (datasources, dashboards, alerts) directly from your local machine.
- Any changes you make in the local folder are applied instantly inside the container.

# Conclusion

This project successfully demonstrated the end-to-end process of building, containerizing, and monitoring a modern web service. We developed a fully functional URL shortener, instrumented its code to expose custom business and performance metrics, and established a complete observability stack using industry-standard tools.

The core achievement was the seamless integration of the entire system—application, Prometheus for metrics collection, and Grafana for visualization—within a local Docker environment using Docker Compose. The final Grafana dashboard provides real-time, actionable insights into the service's health, usage patterns (URL shortening and redirect rates), and performance (request latency, error rates).