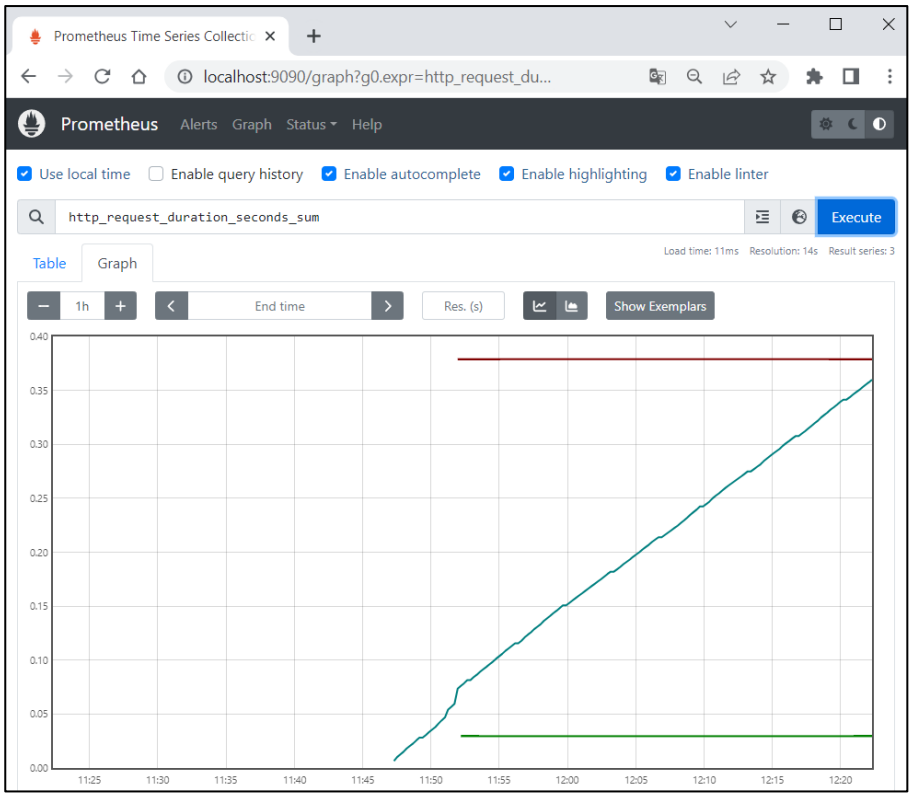
Exercise: IaC and Monitoring

Exercise assignment for the "Containers and Clouds" course @ SoftUni.

I. App Monitoring

1. Monitor the "Contact Book" Node.js App with Prometheus

We have the **`Node.js`** "**`Contact`** **Book**" **app** in the **resources**. We aim to **monitor it using Prometheus**, so we need its **metrics**. In this case, we will **instrument the app** to **expose the metrics** we want. And then we will **configure Prometheus** to **display these metrics**.



Step 1: Examine the App

We have the "**`Contact`** **Book**" **Node.js app**, which holds a **searchable list of contacts**. You have pages to **list all contacts** (**`/contacts`**), **view a single contact** (**`/contacts/:id`**), **search for a contact** (**`/contacts/search/:keyword`**) and **add a new contact** (**`/contacts/create`**).

Open the **project in Visual Studio Code** to **examine its files**:

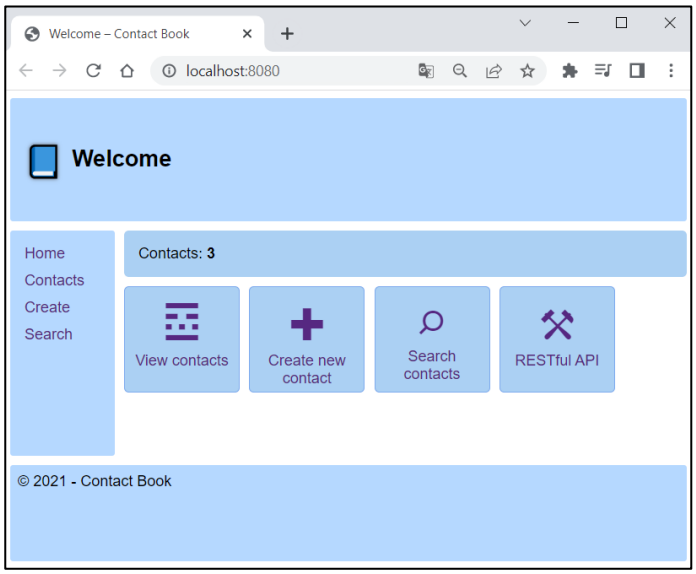
```
function setup(app, data) {
  app.get('/api', function(req, res) {
    let routes = app._router.stack
      .filter(r => r.route && r.route.path.startsWith('/api'))
    .map(rt => ({
      route: rt.route.path,
      method: rt.route.stack[0].method
    }));
    res.send(routes);
  });

  app.get('/api/contacts', function(req, res) {
    let result = data.getContacts();
    res.send(result);
  });
```

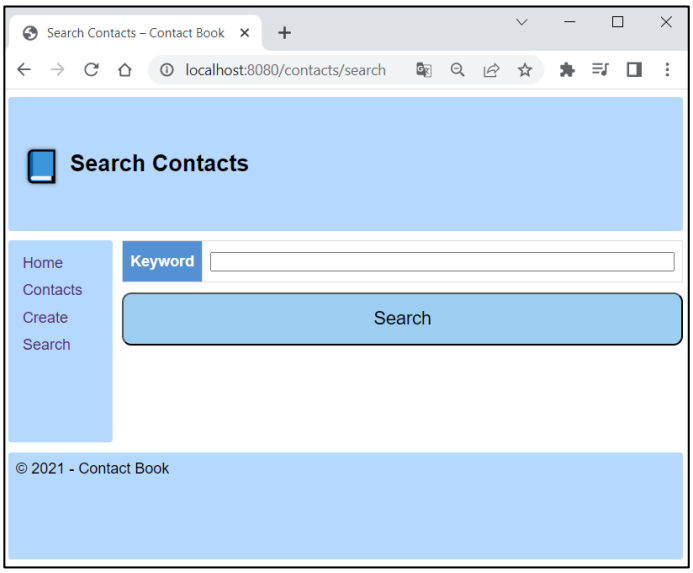
Open a **terminal** and **execute** the "**`npm install`**" and "**`npm start`**" **commands** to **run the app**:

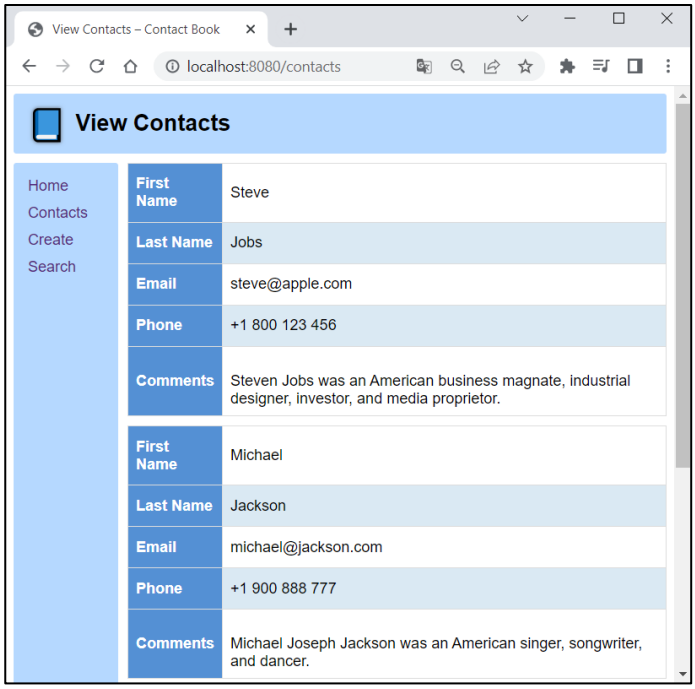
```
PS D:\SoftUni\ContactBook> npm install
npm WARN deprecated core-js@2.6.12: core-js@<3.23.3 is no longer maintained and no
t recommended for usage due to the number of issues. Because of the V8 engine whim
s, feature detection in old core-js versions could cause a slowdown up to 100x eve
n if nothing is polyfilled. Some versions have web compatibility issues. Please, u
pgrade your dependencies to the actual version of core-js.

added 123 packages, and audited 124 packages in 7s

13 packages are looking for funding
  run `npm fund` for details
1 high severity vulnerability

To address all issues (including breaking changes), run:
  npm audit fix --force
Run `npm audit` for details.
```

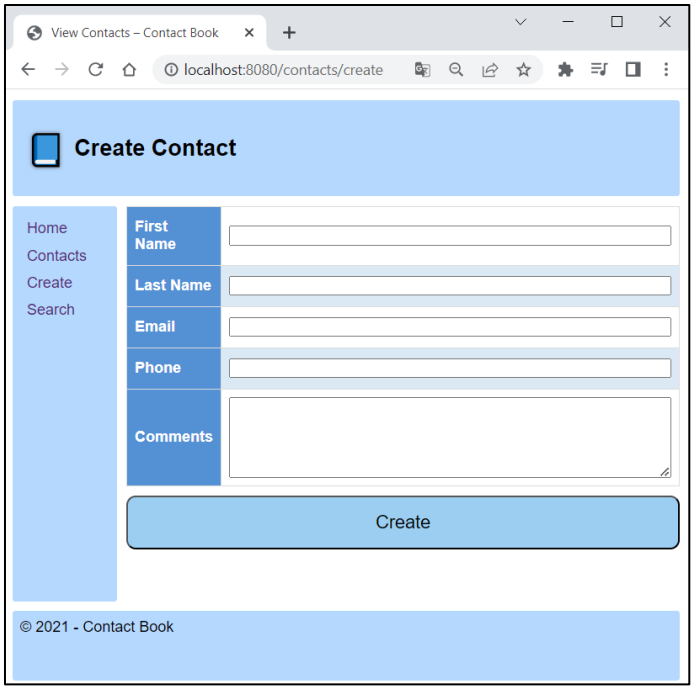
```
PS D:\SoftUni\ContactBook> npm start

> start
> node index.js

App started. Listening at http://localhost:8080
```

Look at the **app pages** on http://localhost:8080:









Let's now see how to **modify the app code** to **export app metrics** for **Prometheus**.

Step 2: Export Node.js App Metrics

To **make app metrics readable for Prometheus**, we should **install an additional client library for Node.js** and then **modify the code** to **define and export the metrics** we want.

Install Prom-Client

Stop the app with **`[Ctrl] + [C]`** in the **terminal**. Then, we should **install the `prom-client` package**, which is the **Prometheus client for Node.js** that supports histogram, summaries, gauges and counters. Do it with the following **command**:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL                                node

PS D:\SoftUni\ContactBook> npm install prom-client

added 3 packages, and audited 127 packages in 767ms

13 packages are looking for funding
  run `npm fund` for details

1 high severity vulnerability

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

You can see that the **`prom-client` package** is added to the **`package.json` file** with **project dependencies**:

```
File  Edit  Selection  View  Go  ...   • package.json - ContactBook - Visual Studio Code
EXPLORER
CONTACTBOOK
  controllers
    api-controller.js
    mvc-controller.js
  data
    app-data.js
  node_modules
  public
    styles.css
  views
  index.js
  package-lock.json   U
  package.json   1, M
OUTLINE
TIMELINE

package.json 1, M •
package.json > ...
1  {
2    "name": "ContactBook",
3    "main": "index.js",
4    "dependencies": {
5      "body-parser": "^1.19.0",
6      "express": "^4.17.1",
7      "prom-client": "^14.2.0",
8      "pug": "^2.0.4"
9    },
       Debug
10   "scripts": {
11     "start": "node index.js"
12   }
13 }

master*   0 1   Ln 14, Col 1   Spaces: 2   UTF-8   CRLF   JSON
```

Because of this, you **won't need to install the package** separately from the others next time.

Export Default Metrics

Now we will **modify our code** to **collect the default app metrics** together with **some custom ones** and **expose them** on the **`/metrics` endpoint**.

To do this, navigate to **`mvc-controller.js` file** where the main app routing is and **include the `prom-client` module**, as we will need it:

```
JS mvc-controller.js •
controllers > JS mvc-controller.js > setup
1  const client = require('prom-client');
2
```

Then, **create a registry** to register the metrics:

```
3  const register = new client.Registry();
4
```

Use the **`collectDefaultMetrics()` function** from the **imported module** to **collect and register default metrics** for **monitoring the Node.js application**, for example CPU usage, memory usage, event loop latency, and garbage collection duration:

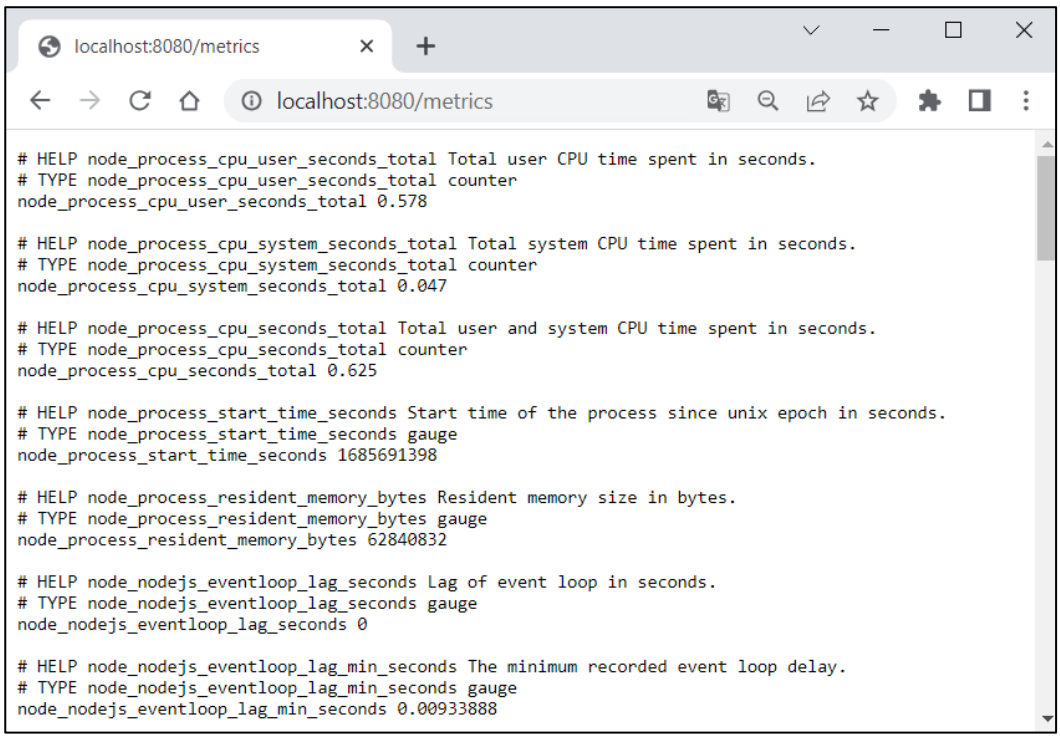
```
client.collectDefaultMetrics({
  app: 'node-application-monitoring-app',
  prefix: 'node_',
  timeout: 10000,
  gcDurationBuckets: [0.001, 0.01, 0.1, 1, 2, 5],
  register
});
```

This configuration sets **default metric names** to start with the "**_node**" prefix, the **timeout** to 10000ms, the **buckets for the default metric that measures garbage collection** (GC) **durations** (values represent the upper bounds of each bucket) and the **registry** that we created to be used.

These are the **default metrics** we will export. Now, in the **`setup()` function**, set up an **HTTP GET route** for the **`/metrics` endpoint**, which should **return the collected app metrics as response**:

```
function setup(app, data) {
  app.get('/metrics', async (req, res) => {
    res.setHeader('Content-Type', register.contentType);
    res.send(await register.metrics());
  });

```

Before we add some **custom metrics**, let's see how **default metrics are showed**. Save the changes and **start the app** again. Then, **navigate to** http://localhost:8080/metrics in the browser:



Now let's **add some more metrics**.

Export Custom Metrics

The **custom metrics** we shall export are about the **duration of HTTP requests to different endpoints in seconds**. They will be saved in a **histogram** with **buckets from 0.01 to 1 seconds** and will keep **request method**, **route** and **status code**.

Add the following code to **create the histogram metric** (before the **`setup()`** function):

```
const httpRequestTimer = new client.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Duration of HTTP requests in seconds',
  labelNames: ['method', 'route', 'code'],
  buckets: [0.01, 0.03, 0.05, 0.07, 0.1, 0.3, 0.5, 0.7, 1]
});

```

Then we should **register the metric**:

```
register.registerMetric(httpRequestTimer);

```

Now, for **each of the routes**, we should:

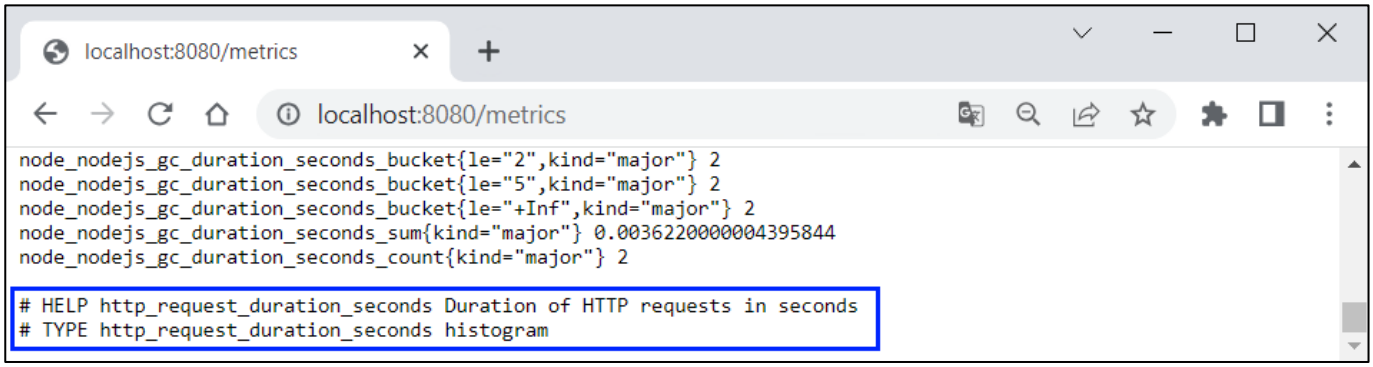
- Start an **HTTP request timer**, saving a **reference** to the returned method
- Save **reference to the path** so we can record it when ending the timer
- And finally **end the timer** and **add labels**

In this way, the **HTTP request data and duration** will be recorded. Do it for the **`/metrics` endpoint** like this:

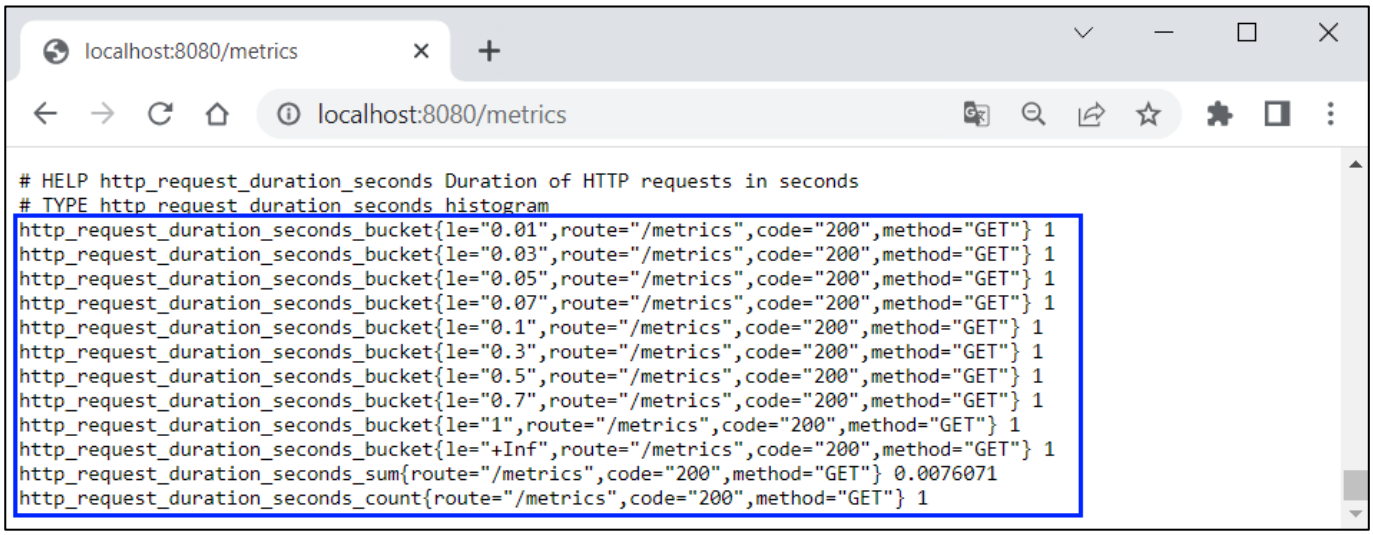
```
function setup(app, data) {
  app.get('/metrics', async (req, res) => {
    const end = httpRequestTimer.startTimer();
    const route = req.route.path;

    res.setHeader('Content-Type', register.contentType);
    res.send(await register.metrics());

    end({ route, code: res.statusCode, method: req.method });
  });

```

Do it for the **rest of the endpoint methods** in the same way by **adding the above three lines**. When ready, **run the app** again.

When you first **access `/metrics`**, you will see the **new metrics at the bottom**:



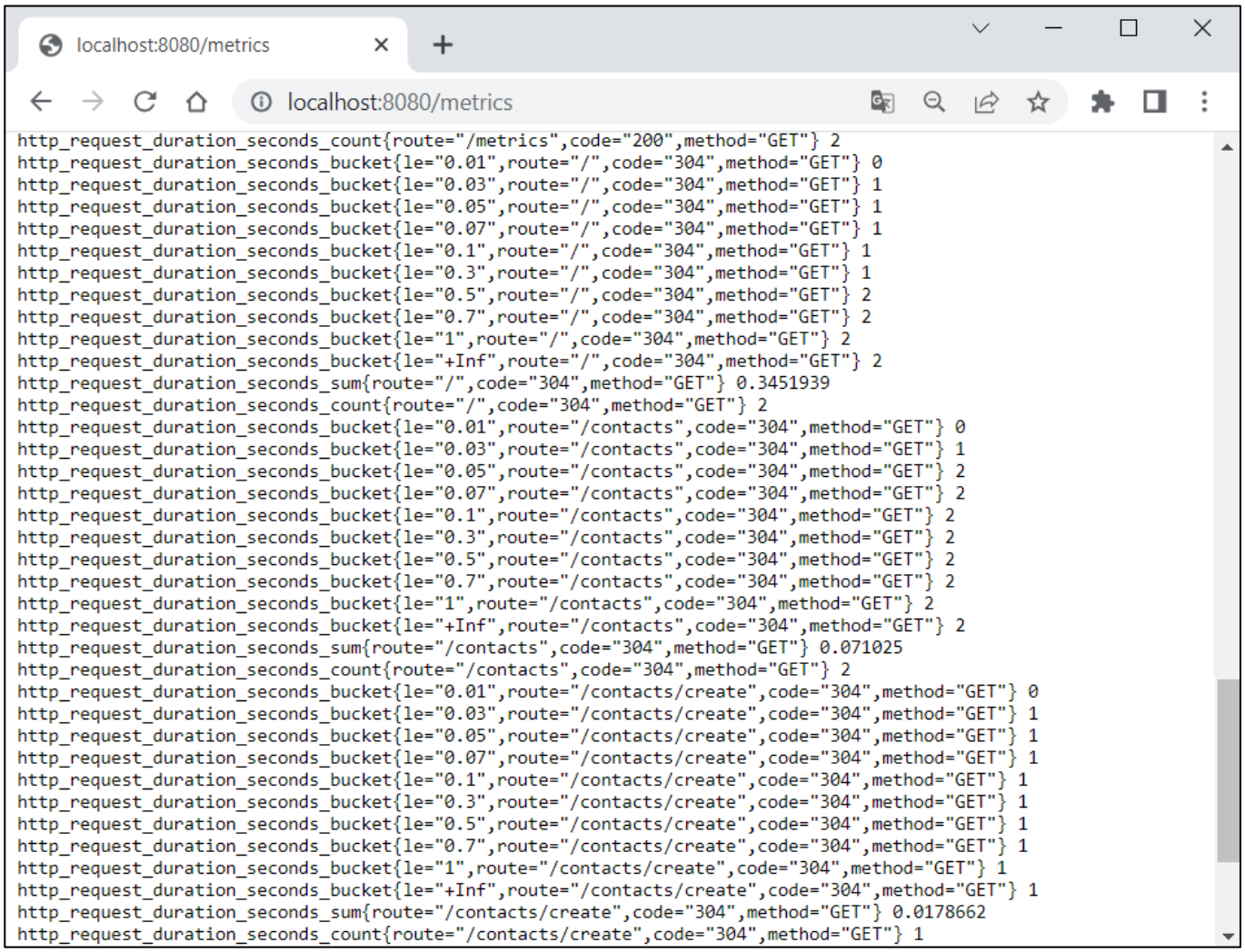
However, you still have **no metric values**. You should **refresh the page**, so that the metrics for the **previous HTTP request** to **`/metrics`** are displayed:



```
# HELP http_request_duration_seconds Duration of HTTP requests in seconds
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.01",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.03",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.05",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.07",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.1",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.3",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.5",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="0.7",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="1",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_bucket{le="+Inf",route="/metrics",code="200",method="GET"} 1
http_request_duration_seconds_sum{route="/metrics",code="200",method="GET"} 0.0076071
http_request_duration_seconds_count{route="/metrics",code="200",method="GET"} 1
```

As you can see, the **first HTTP request to `/metrics`** took about **0.0076 seconds**, which is **less than 0.01** and that's why it falls into **each of the buckets**.

If you **access the other app endpoints**, you will get even **more metric data**:

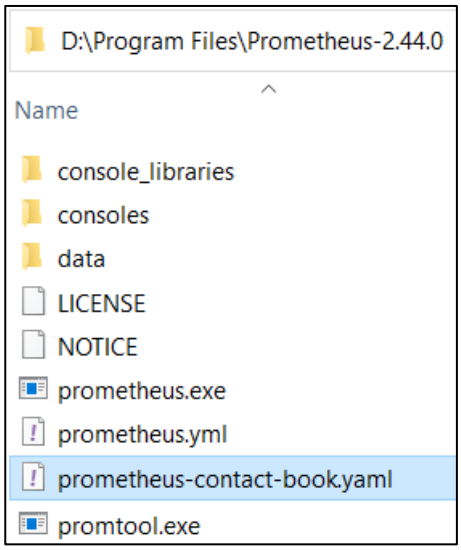


```
http_request_duration_seconds_count{route="/metrics",code="200",method="GET"} 2
http_request_duration_seconds_bucket{le="0.01",route="/",code="304",method="GET"} 0
http_request_duration_seconds_bucket{le="0.03",route="/",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.05",route="/",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.07",route="/",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.1",route="/",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.3",route="/",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.5",route="/",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.7",route="/",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="1",route="/",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="+Inf",route="/",code="304",method="GET"} 2
http_request_duration_seconds_sum{route="/",code="304",method="GET"} 0.3451939
http_request_duration_seconds_count{route="/",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.01",route="/contacts",code="304",method="GET"} 0
http_request_duration_seconds_bucket{le="0.03",route="/contacts",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.05",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.07",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.1",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.3",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.5",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.7",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="1",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="+Inf",route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_sum{route="/contacts",code="304",method="GET"} 0.071025
http_request_duration_seconds_count{route="/contacts",code="304",method="GET"} 2
http_request_duration_seconds_bucket{le="0.01",route="/contacts/create",code="304",method="GET"} 0
http_request_duration_seconds_bucket{le="0.03",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.05",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.07",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.1",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.3",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.5",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="0.7",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="1",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_bucket{le="+Inf",route="/contacts/create",code="304",method="GET"} 1
http_request_duration_seconds_sum{route="/contacts/create",code="304",method="GET"} 0.0178662
http_request_duration_seconds_count{route="/contacts/create",code="304",method="GET"} 1
```

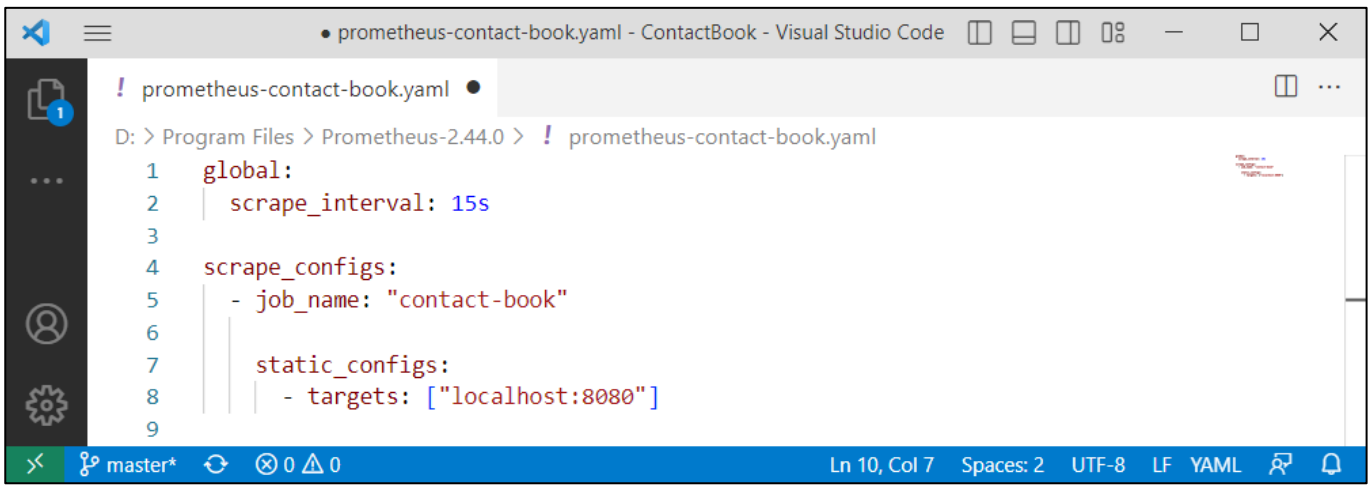
You can use the **browser inspector** in the browser to **compare the HTTP request times** shown here and there – they should be pretty close as values.

Step 3: Condifure and Run Prometheus

Go to the **Prometheus installation directory** where our binary files are and **create a YAML file** where we will **write the configuration** for **monitoring the** "**`Contact` Book**" **app**:



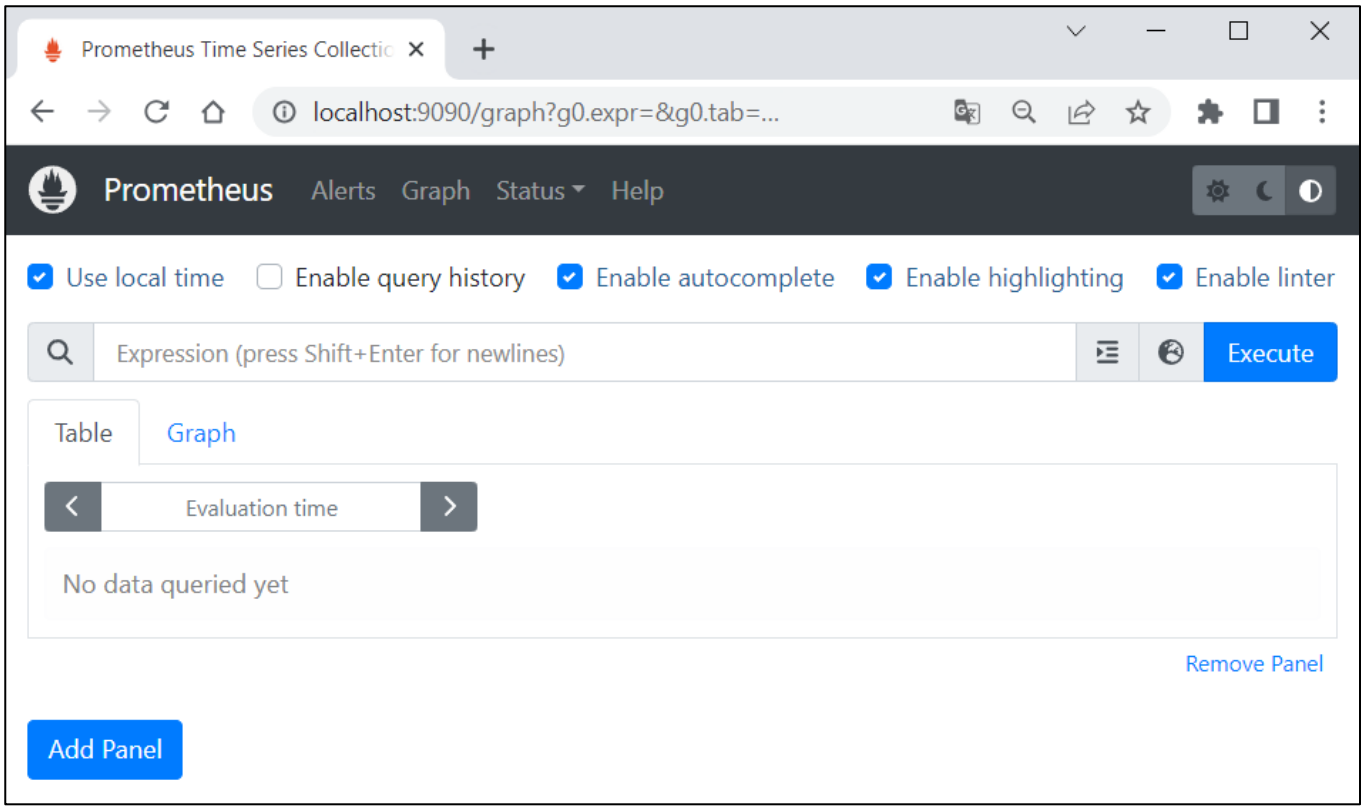
In the **Prometheus configuration file**, we should **define a single job** to **monitor our app on** http://localhost:8080 and **scrape target metrics** on **every 15 seconds**:



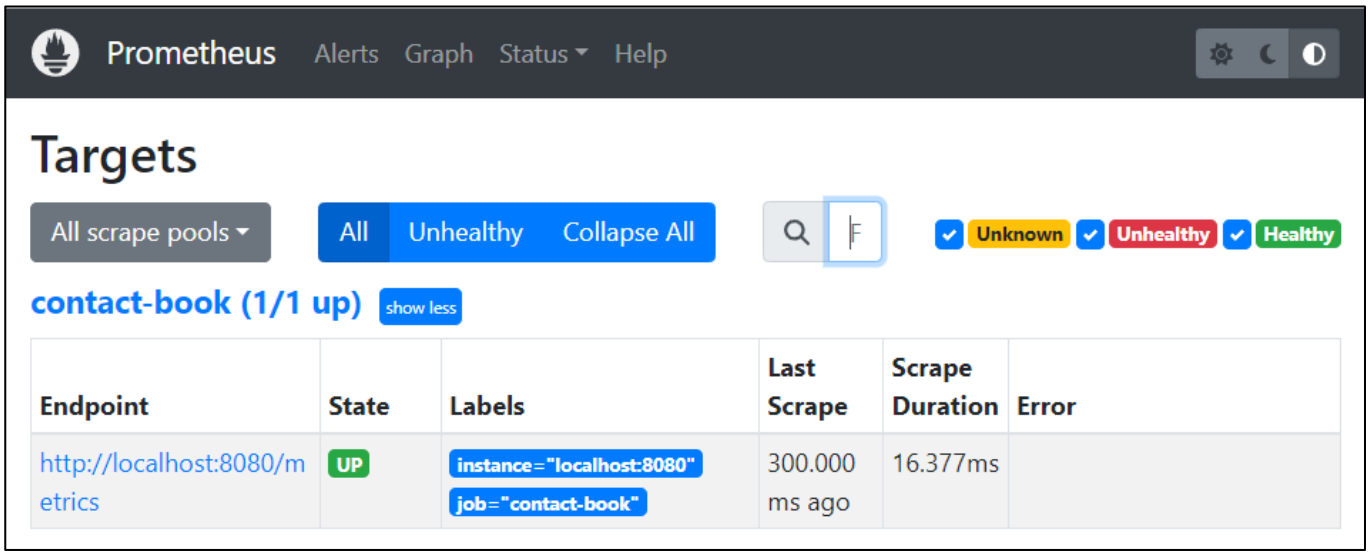
Save the file and **open a terminal**. Navigate to the **Prometheus installation directory** and **run Prometheus** with this **configuration file**:

```
Windows PowerShell
PS D:\Program Files\Prometheus-2.44.0> .\prometheus --config.file .\prometheus-contact-book.yaml

ts=2023-06-02T08:41:57.946Z caller=main.go:531 level=info msg="No time or size retention was set
 so using the default time retention" duration=15d
ts=2023-06-02T08:41:57.946Z caller=main.go:575 level=info msg="Starting Prometheus Server" mode=
server version="(version=2.44.0, branch=HEAD, revision=1ac5131f698ebc60f13fe2727f89b115a41f6558)
"
ts=2023-06-02T08:41:57.948Z caller=main.go:580 level=info build_context="(go=go1.20.4, platform=
windows/amd64, user=root@5be246f61ac8, date=20230514-06:23:08, tags=builtinassets,stringlabels)"

ts=2023-06-02T08:41:57.948Z caller=main.go:581 level=info host_details=(windows)
ts=2023-06-02T08:41:57.948Z caller=main.go:582 level=info fd_limits=N/A
ts=2023-06-02T08:41:57.948Z caller=main.go:583 level=info vm_limits=N/A
ts=2023-06-02T08:41:58.004Z caller=web.go:562 level=info component=web msg="Start listening for
connections" address=0.0.0.0:9090
ts=2023-06-02T08:41:58.005Z caller=main.go:1016 level=info msg="Starting TSDB ..."
ts=2023-06-02T08:41:58.008Z caller=tls_config.go:232 level=info component=web msg="Listening on"
 address=[::]:9090
ts=2023-06-02T08:41:58.008Z caller=tls_config.go:235 level=info component=web msg="TLS is disabl
ed." http2=false address=[::]:9090
ts=2023-06-02T08:41:58.012Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1684488907664 maxt=1684490400000 ulid=01H11YRK2ABZR9Y5TCFKKECFW5
ts=2023-06-02T08:41:58.016Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1684766739323 maxt=1684771200000 ulid=01H1BMFMMASW8418M3DPTMS68R
ts=2023-06-02T08:41:58.021Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1684412137655 maxt=1684425600000 ulid=01H1BMFN5MDYT76HJG8MZ6CMGK
ts=2023-06-02T08:41:58.027Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1685091504336 maxt=1685109600000 ulid=01H1NR6RD2QDVV13HJ88E8V911
ts=2023-06-02T08:41:58.032Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1685430949889 maxt=1685433600000 ulid=01H1P2G95FW9GXAF87ZZM0696F
ts=2023-06-02T08:41:58.037Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1685433604902 maxt=1685440800000 ulid=01H1P519Y7F8Z24W2W4YNC3HKC
ts=2023-06-02T08:41:58.042Z caller=repair.go:56 level=info component=tsdb msg="Found healthy blo
ck" mint=1685346574897 maxt=1685354400000 ulid=01H1P51B0YPBS4VJY2QCTHKW96
```

Prometheus server should be **available on** http://localhost:9090 by default:

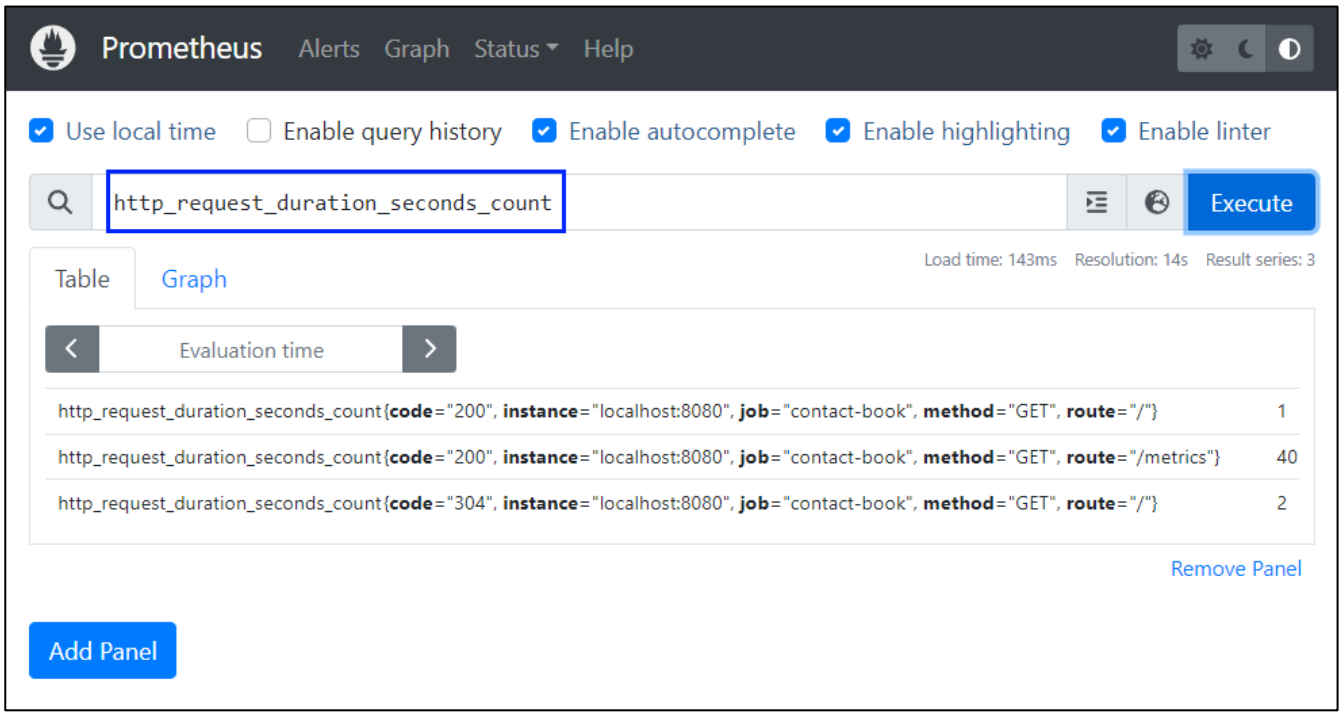


You can navigate to **[Status] → [Targets]** to see that **Prometheus connects** to the **configured target app** successfully:

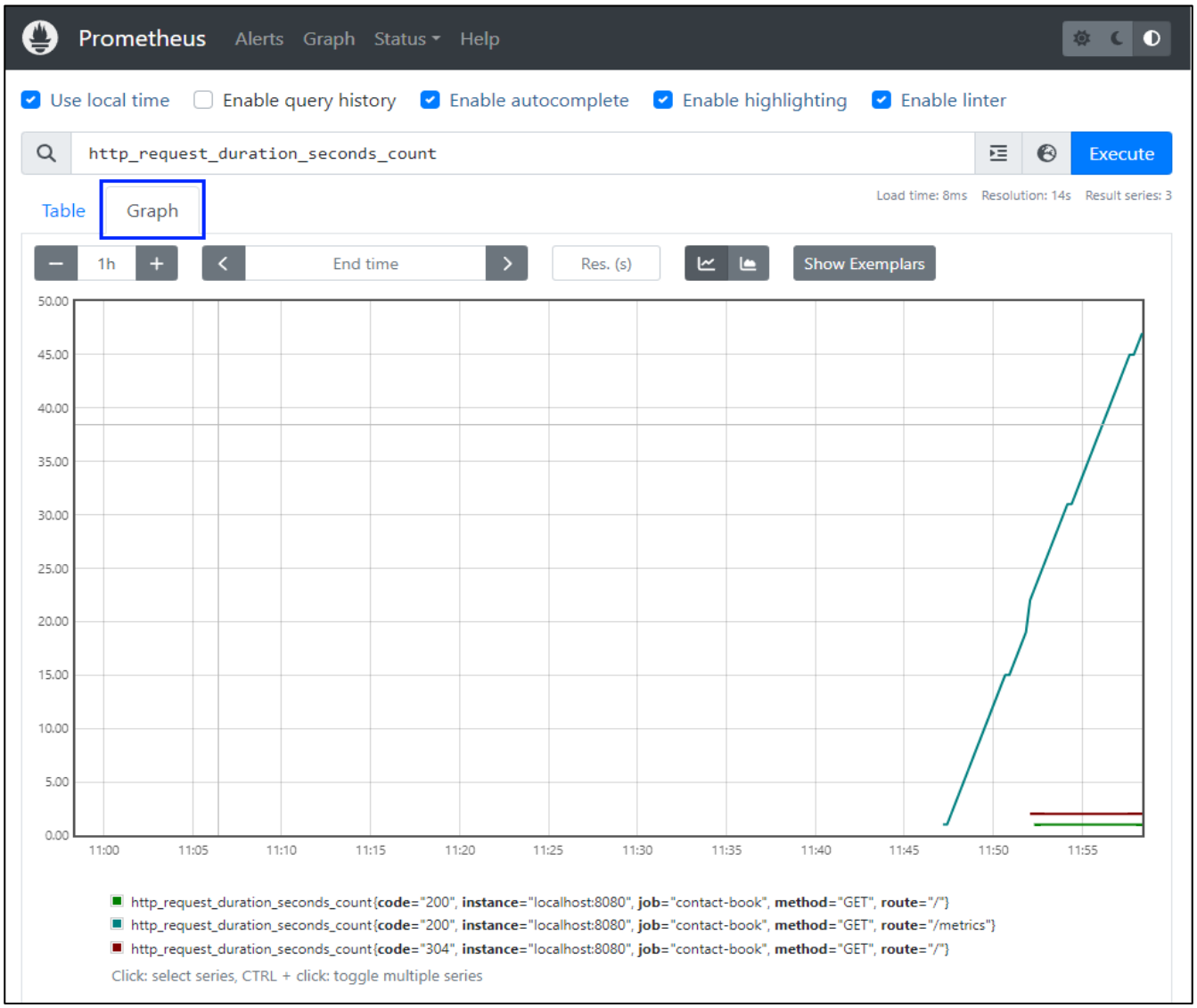


NOTE: the "**`Contact Book`**" **app should be running** to expose metrics.

Now you can go back to the **`[Graph]` page** and **display some of the metric values**, using an **expression**. For example, let's see the **count of all different HTTP requests**:



Switch to **`[Graph]`** to **look at a graph for the metric**:



You can **examine more metrics** you want. When **metrics change**, click on **`[Execute]`** to **load the changed graph**.

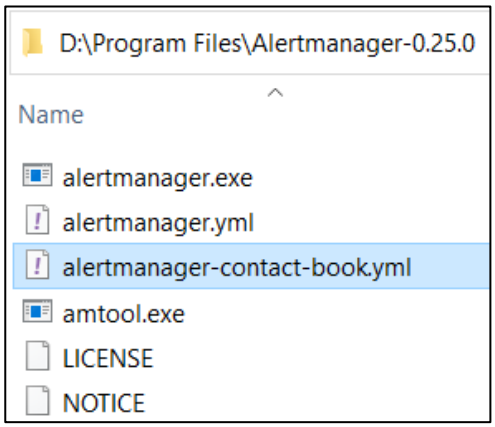
As we know how to **work with Prometheus**, let's see how to **add `Alertmanager`** to **manage alerts** and **send notifications**.

2. Manage "Contact Book" App Alerts with Alertmanager

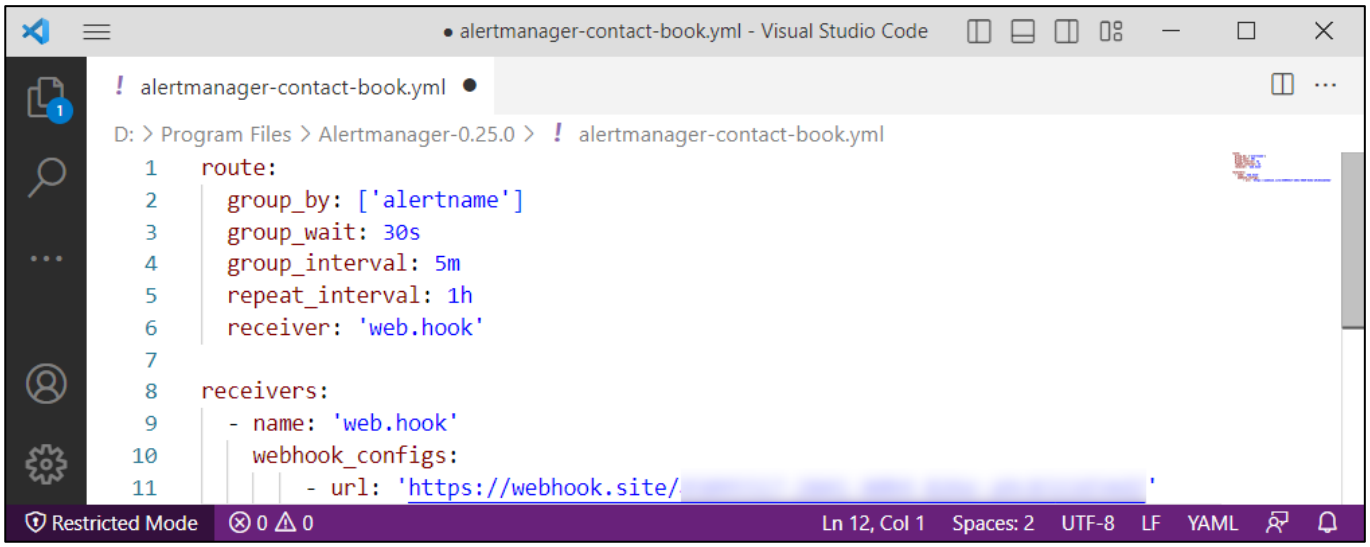
In this task, we will manage Prometheus alerts with Alertmanager and send them to Webhook.site to keep them. Our aim is to fire alerts when any page has been accessed more than 3 times during a 5-minute period.

Step 1: Configure and Run Alertmanager

Let's first see how to write a configuration file for Alertmanager to handle alerts. First, create a YAML file in the Alertmanager installation directory:



Open the file in an editor and write the sample configuration below:

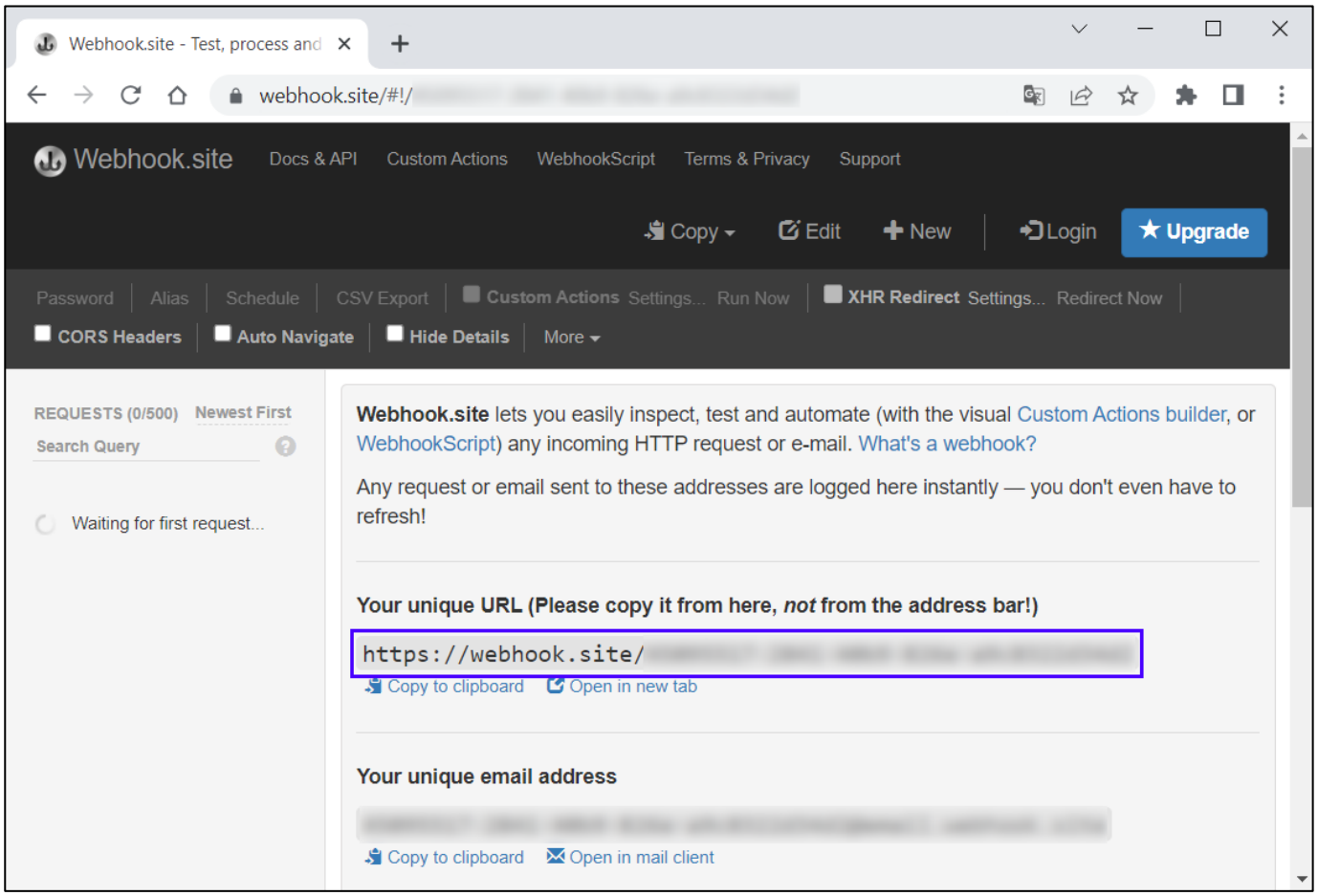


This configuration:

- Groups alerts by their name
- Sets alerts to be grouped together for a period of 30 seconds before being sent
- Sets notifications for unresolved alert groups to be sent every 5 minutes
- Sets notifications for unresolved alerts to be repeated every 1 hour
- Defines notification receiver to be "web.hook"
- Configures receiver

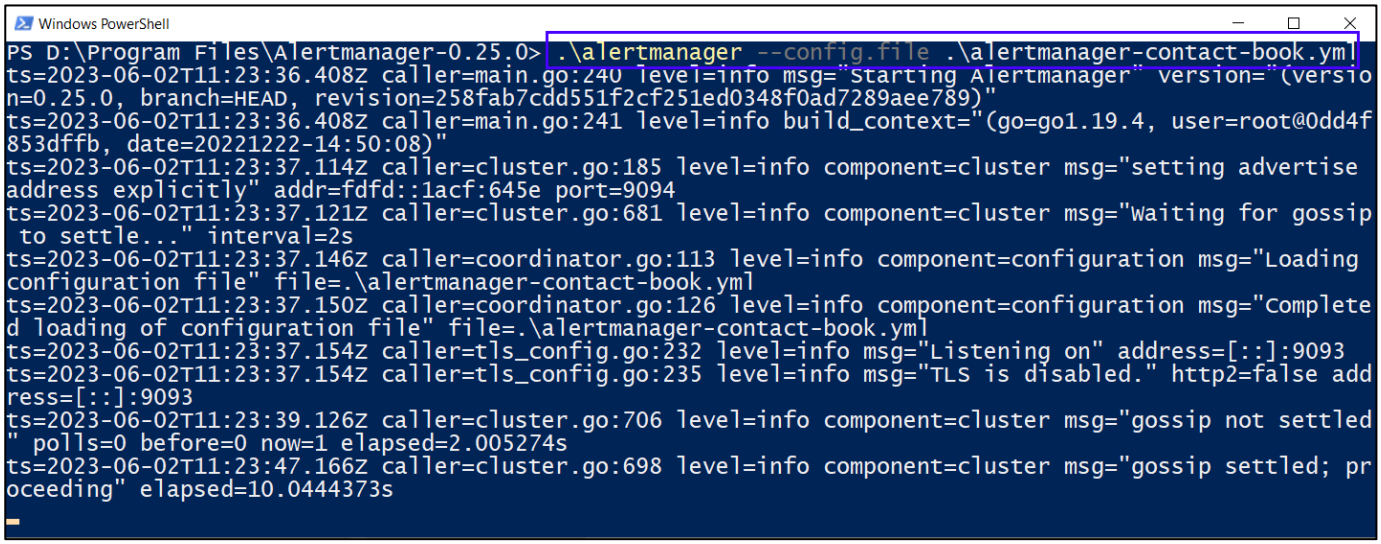
As you can see, the configuration contains a Webhook.site URL, which is unique. Webhook.site allows us to create temporary endpoints (webhooks) and capture the incoming requests sent to those endpoints, e.g., our Prometheus notifications.

To get your URL, navigate to Webhook.site and copy the provided URL, without closing the browser tab after this:

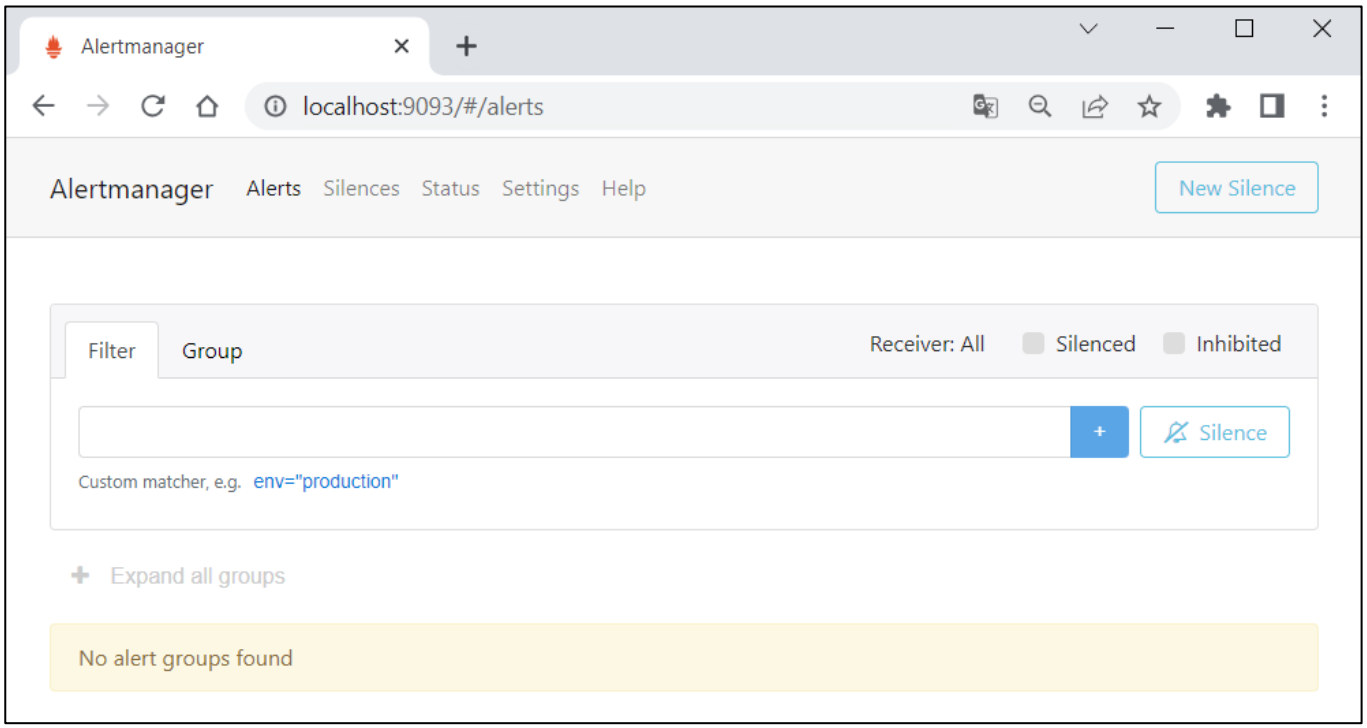


Add the URL to your configuration file and save it.

Next, run Alertmanager with the configuration file:



Go to http://localhost:9093 in the browser and you should see that Alertmanager is up and working:

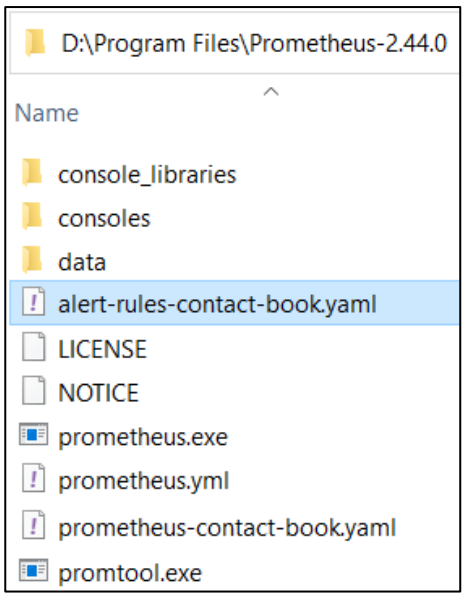


Now let's configure Prometheus to work with Alertmanager.

Step 2: Configure and Run Prometheus

We should do 2 things to make Prometheus send alerts to Alertmanager – first, create a YAML file with rules for firing an alert and, second, modify the Prometheus configuration file to use the rules and send alerts to Alertmanager.

Create a new YAML file in the Prometheus installation directory, which will define alert rules:



As we said earlier, we will fire an alert when an endpoint is accessed more than 3 times for 5 minutes. We will measure that using the **`http_request_duration_seconds_count`** metric – if its value has changed more than 3 times for the last 5 minutes. We shall have the following expression:

```
changes(http_request_duration_seconds_count[5m]) >= 3
```

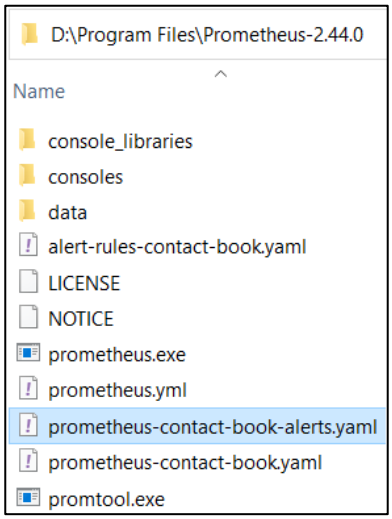
Note that in our case we count how many times the requests count has changed on data scrape (on every 15 seconds), not how many times the count has changed generally.

Having this expression, add the following rules configuration to the created YAML file:

```yaml
groups:
  - name: Page access 3 times for 5 minutes
    rules:
      - alert: PageAccessed3TimesFor5Minutes
        expr: changes(http_request_duration_seconds_count[5m]) >= 3
        for: 10s
        labels:
          severity: warning
        annotations:
          summary: "Page was accessed for more than 3 times within the last 5 minutes."
```

Here we have a single rules group and an alert that will be fired if the given expression is true for at least 10 seconds. The alert will have a label and summary.

Save the file and let's modify (or create a new separate file) the Prometheus configuration. It should look like this:



→

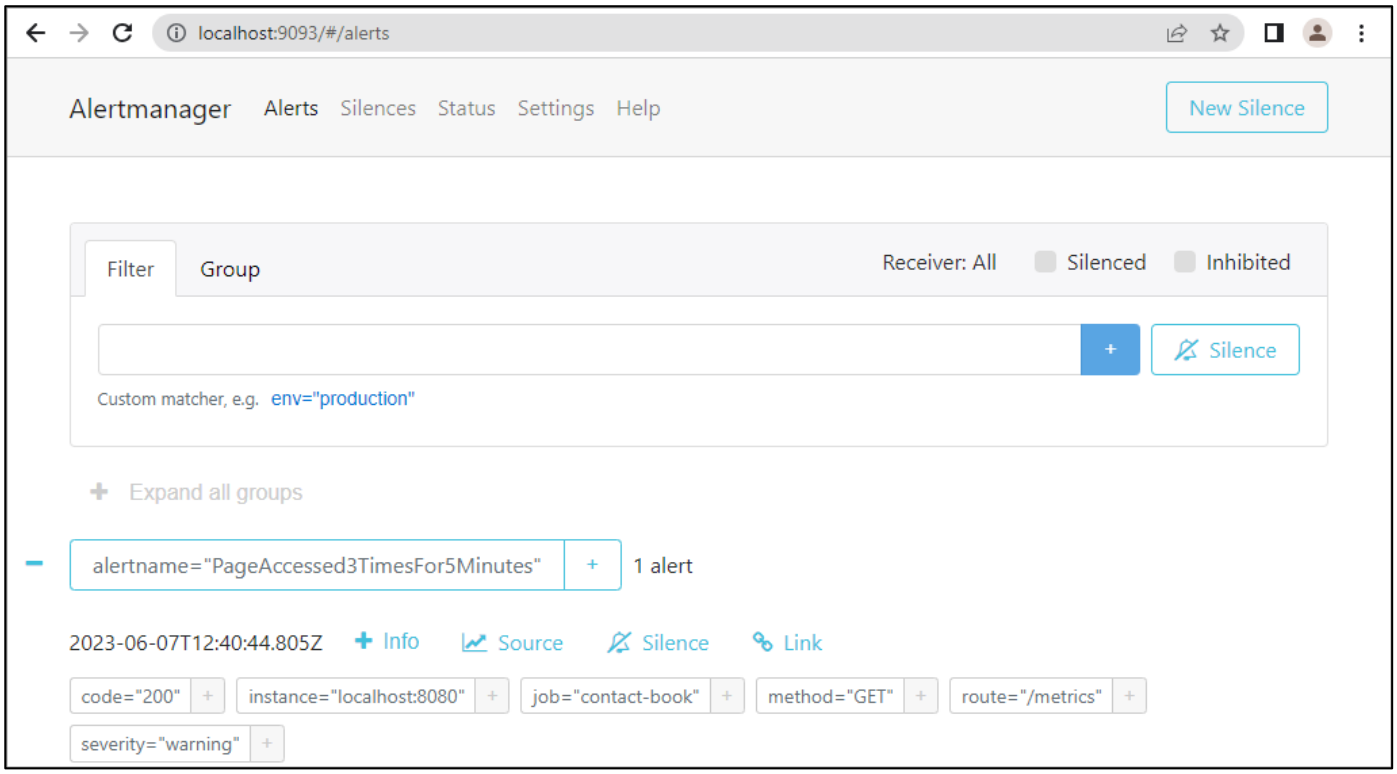
```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - alert-rules-contact-book.yaml

alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - localhost:9093

scrape_configs:
  - job_name: "contact-book"
    static_configs:
      - targets: ["localhost:8080"]
```

As you can see, we have added the name of the rules file and configurations for connection to Alertmanager, which is accessible on http://localhost:9093 by default. Also, we have set **`evaluation_interval`**, which is the interval based on which Prometheus evaluates the query for alerting.

Now you should run Prometheus with the new / modified configuration file (don't forget to change the name of the configuration file if you have named it in a different way):

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS D:\Program Files\Prometheus-2.44.0> .\prometheus --config.file .\prometheus-contact-book-alerts.yaml
```

Go to the **Contact Book app** on http://localhost:8080/ and reload the page more than 3 times. Now, access **Alertmanager** on http://localhost:9093 and you should be able to see the new alert:



Now, visit the opened tab with Webhook.site. You should be able to see the detailed info about the sent incoming requests: