

Lab 13

Part A: Logging

Modify the Bank application so that logging is done properly to a log file instead of `System.out.println()`.

Apply the different levels of logging for the bank and check that these levels work correctly

Part B: Actuators

Add the actuators dependency to the Bank application

Call and study the output of the following actuators:

`/health`

`/env`

`/beans`

`/configprops`

`/mappings`

`/scheduledtask`

Also shut the application down using the actuator.

Part C: Prometheus and Grifana

Write a simple spring boot web application with the following REST controller:

```
@RestController
public class GreetingController {

    @RequestMapping(value="/greeting")
    public String greeting() {
        return "Hello World";
    }
}
```

Run the application and check if it works.

Then add the following dependency in POM.xml.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Reload the maven POM dependencies

Then add the following configuration in application.properties:

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

Rerun the application again and check if you can call some endpoints.

```
localhost:8080/actuator
localhost:8080/actuator

{"_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"beans":{"href":"http://localhost:8080/actuator/beans","templated":false},"caches":{"href":"http://localhost:8080/actuator/caches","templated":false},"caches-cache":{"href":"http://localhost:8080/actuator/caches/{cache}","templated":true},"health":{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":{"href":"http://localhost:8080/actuator/health/{*path}","templated":true},"info":{"href":"http://localhost:8080/actuator/info","templated":false},"conditions":{"href":"http://localhost:8080/actuator/conditions","templated":false},"configprops":{"href":"http://localhost:8080/actuator/configprops","templated":false},"configprops-prefix":{"href":"http://localhost:8080/actuator/configprops/{prefix}","templated":true},"env-toMatch":{"href":"http://localhost:8080/actuator/env/{toMatch}","templated":true},"env":{"href":"http://localhost:8080/actuator/env","templated":false},"loggers":{"href":"http://localhost:8080/actuator/loggers","templated":false},"loggers-name":{"href":"http://localhost:8080/actuator/loggers/{name}","templated":true},"heapdump":{"href":"http://localhost:8080/actuator/heapdump","templated":false},"threaddump":{"href":"http://localhost:8080/actuator/threaddump","templated":false},"metrics-requiredMetricName":{"href":"http://localhost:8080/actuator/metrics/{requiredMetricName}","templated":true},"metrics":{"href":"http://localhost:8080/actuator/metrics","templated":false},"scheduledtasks":{"href":"http://localhost:8080/actuator/scheduledtasks","templated":false},"mappings":{"href":"http://localhost:8080/actuator/mappings","templated":false}}
```

```
localhost:8080/actuator/health
localhost:8080/actuator/health

{"status":"UP","components":{"diskSpace":{"status":"UP","details":{"total":510980517888,"free":14381453312,"threshold":10485760,"exists":true}}, "ping":{"status":"UP"}}}
```

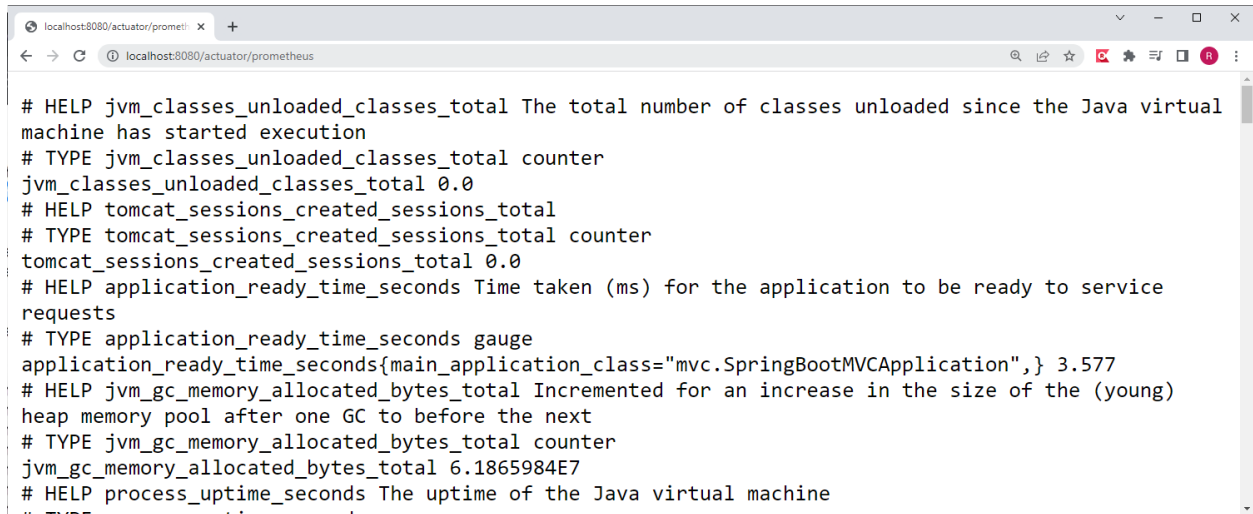
Now add the following micrometer-prometheus dependency in the POM file.

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Reload the maven POM dependencies

Rerun the application again and check if you can call the Prometheus actuator at

<http://localhost:8080/actuator/prometheus>

A screenshot of a web browser window displaying the Prometheus metrics endpoint at localhost:8080/actuator/prometheus. The browser's address bar shows the URL. The main content area displays a list of metrics in a plain text format, including HELP, TYPE, and the current value for each metric. The metrics listed are jvm_classes_unloaded_classes_total, tomcat_sessions_created_sessions_total, application_ready_time_seconds, jvm_gc_memory_allocated_bytes_total, and process_uptime_seconds.

```
# HELP jvm_classes_unloaded_classes_total The total number of classes unloaded since the Java virtual
machine has started execution
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total 0.0
# HELP tomcat_sessions_created_sessions_total
# TYPE tomcat_sessions_created_sessions_total counter
tomcat_sessions_created_sessions_total 0.0
# HELP application_ready_time_seconds Time taken (ms) for the application to be ready to service
requests
# TYPE application_ready_time_seconds gauge
application_ready_time_seconds{main_application_class="mvc.SpringBootMVCAApplication",} 3.577
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young)
heap memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total 6.1865984E7
# HELP process_uptime_seconds The uptime of the Java virtual machine
# TYPE process_uptime_seconds gauge
```

Open the file **C:\EnterpriseArchitetur\prometheus-2.35.0-rc0.windows-amd64\prometheus.yml**

Modify the file as follows:

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
  evaluation_interval: 15s # Evaluate rules every 15 seconds.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
            # - alertmanager:9093

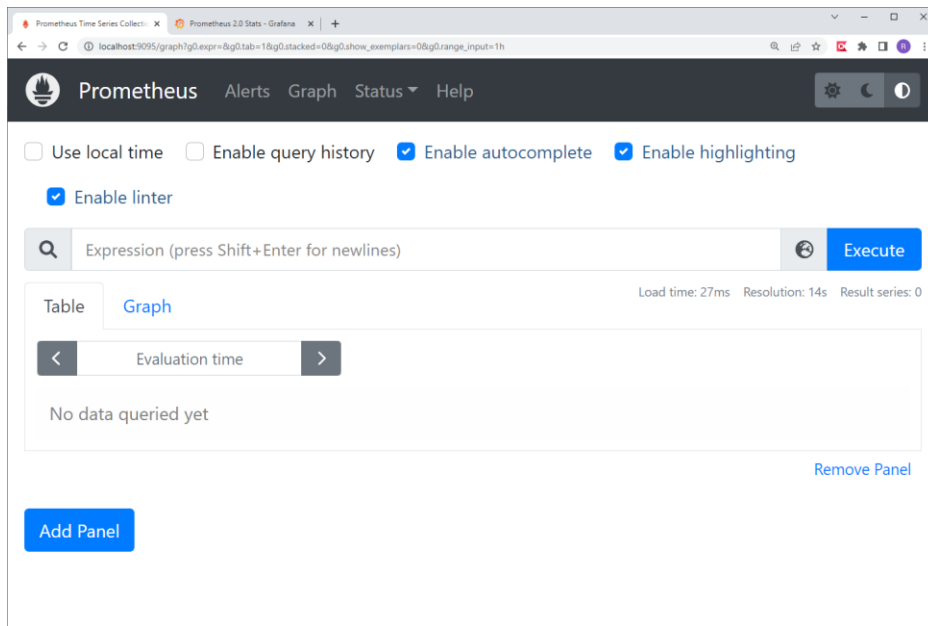
# Load rules once and periodically evaluate them according to the
# Alertmanager configuration
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to the
  # scraped metrics.
  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: [localhost:8080]
```

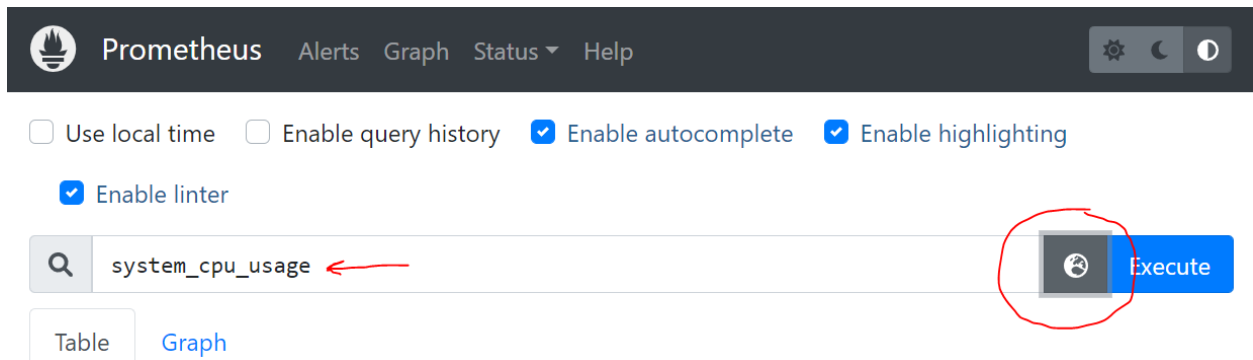
Then save the file.

Then double click the file **C:\prometheus-2.35.0-rc0.windows-amd64\startPrometheus.bat**.

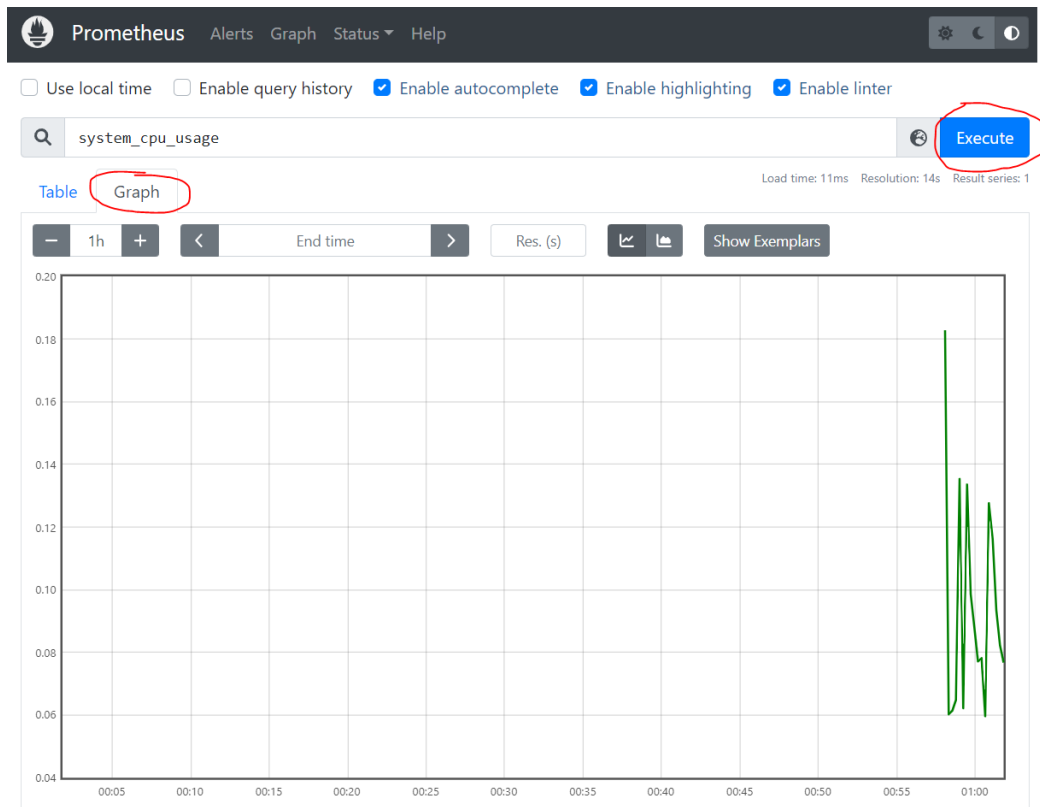
Then in the browser go to **<http://localhost:9095/>** and see if Prometheus is running.



Now click the Open Metrics Explorer button left from the Execute button and select **system_cpu-usage**.

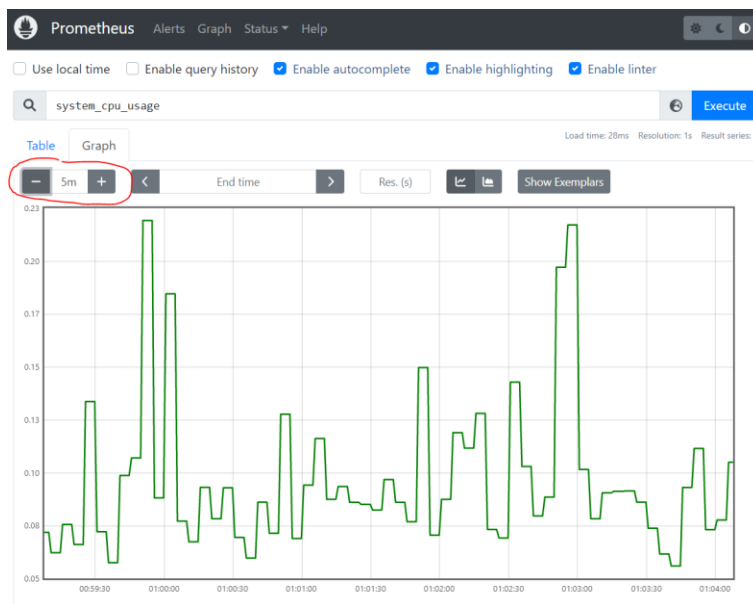


Then click the Execute button and then select the Graph tab:



You now see the cpu usage of your machine.

You can zoom in to 5 minutes:



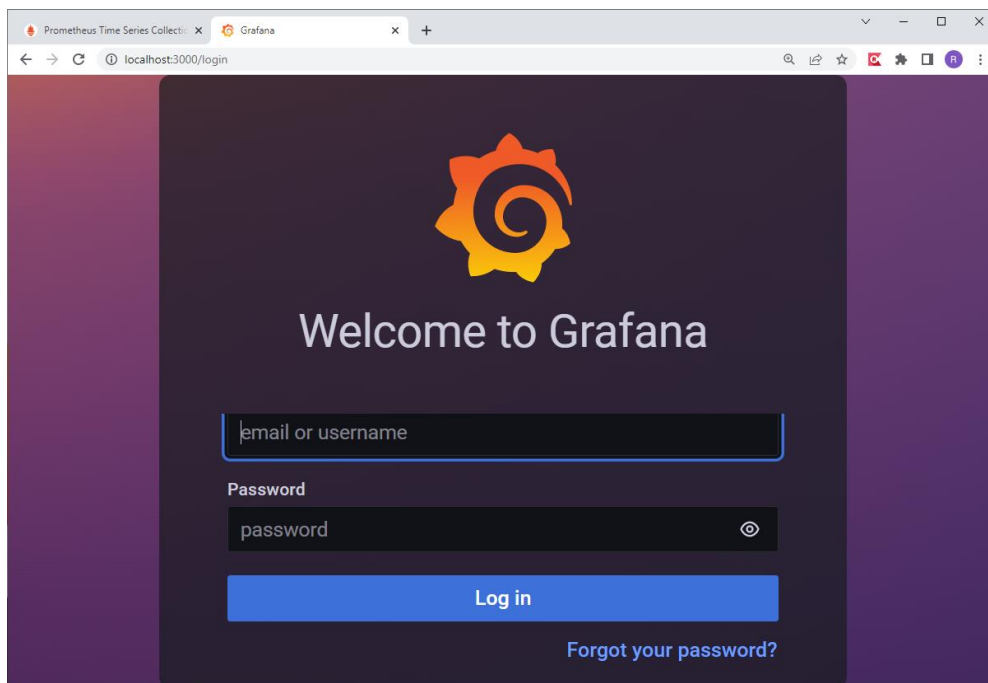
Now try the following metrics:

- disk_free_bytes
- process_cpu_usage
- jvm_classes_loaded_classes

Now double click the file **C:\EnterpriseArchitetur\grafana-8.4.6\bin\grafana-server.exe**

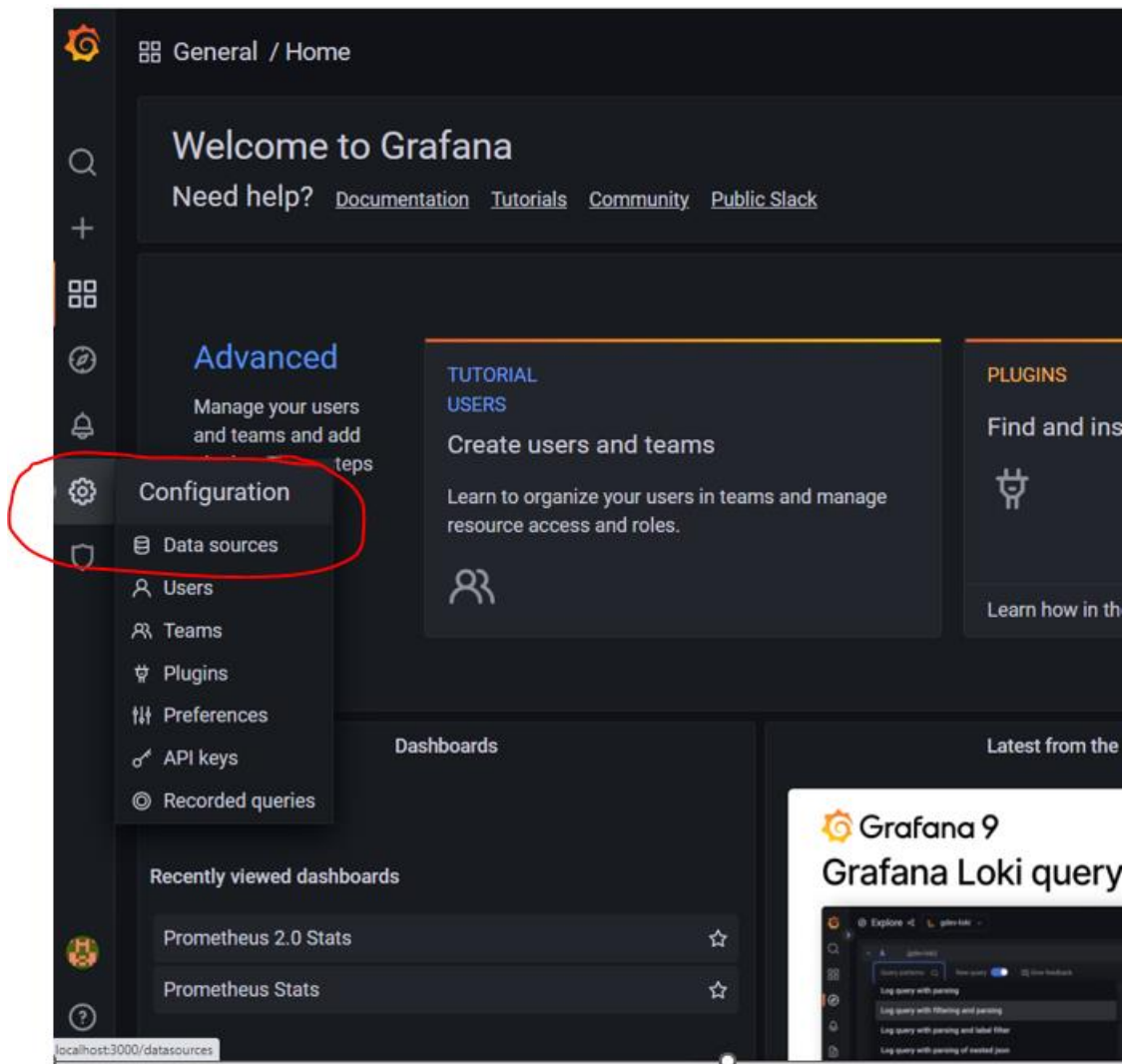
This should start grafana.

In the browser go to **http://localhost:3000/**

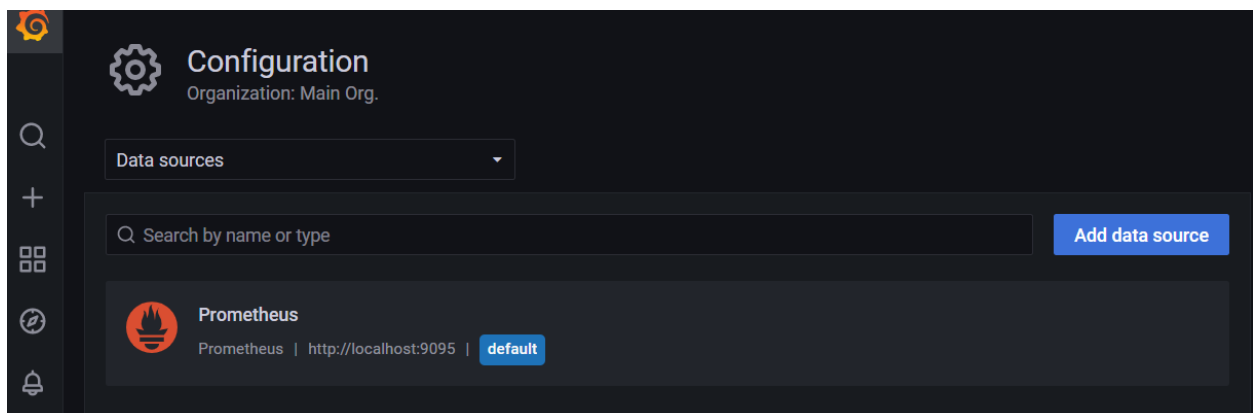


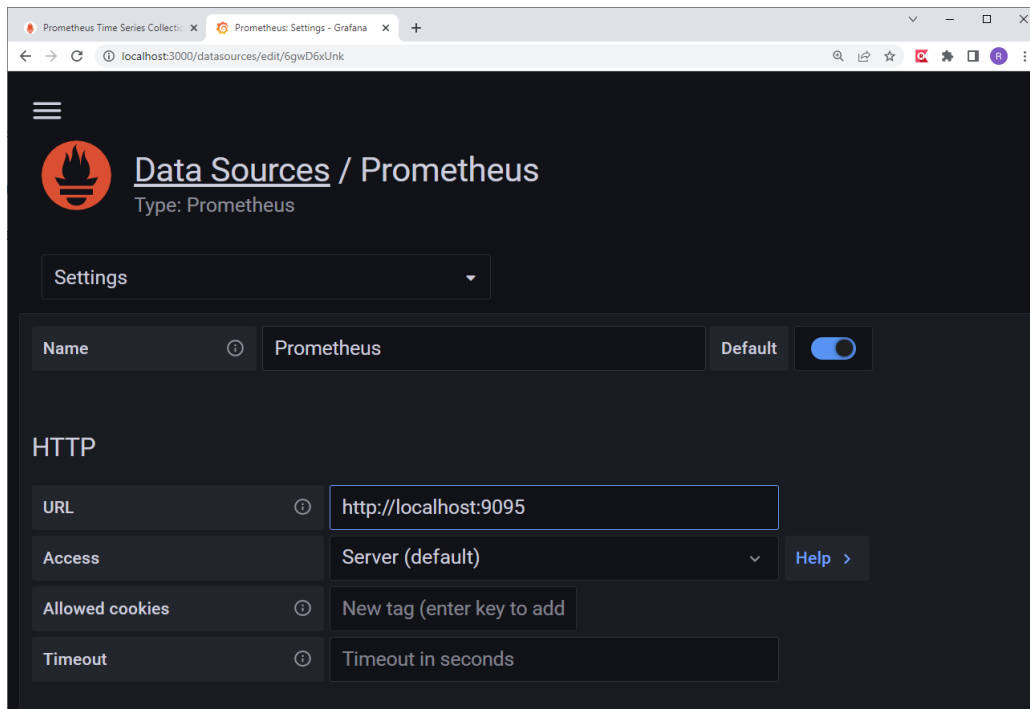
Login with username **admin** and password **admin**.

When Grafana asks for a new password, just type **admin** again.



Select **Configuration**-> **Data sources** and click **Prometheus**.





Prometheus Time Series Collect... x Prometheus Settings - Grafana x +

localhost:3000/datasources/edit/6gwD6xUnk

Data Sources / Prometheus

Type: Prometheus

Settings

Name Prometheus Default ☒

HTTP

URL

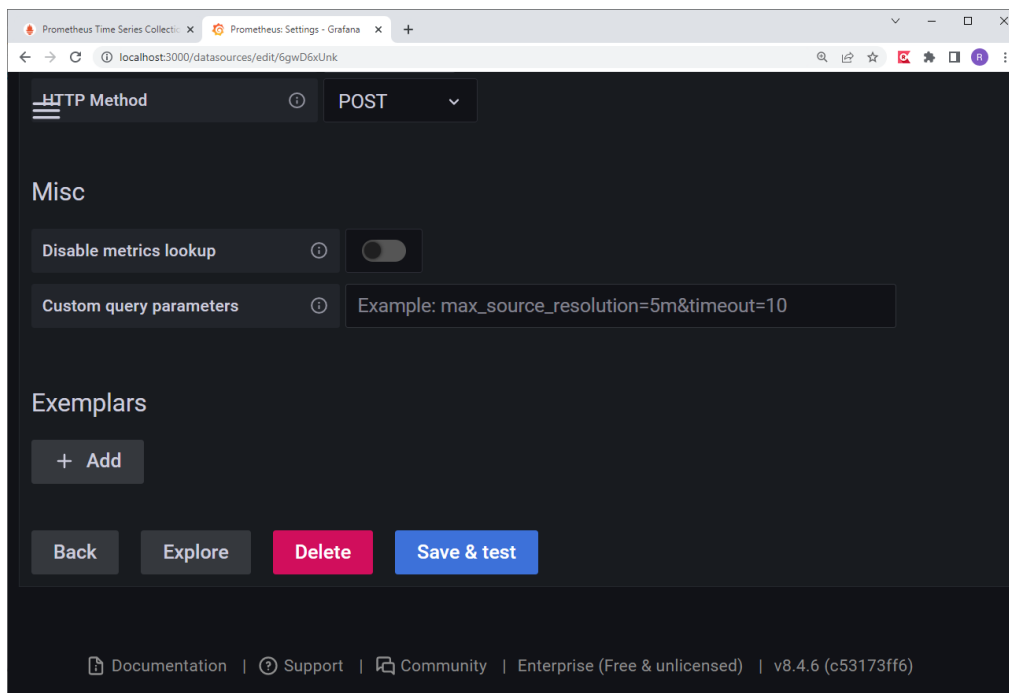
Access Server (default) Help >

Allowed cookies

Timeout

Change the URL to <http://localhost:9095/> if this is not done yet.

Then scroll down to the bottom of the page and select Save and Test



HTTP Method

Misc

Disable metrics lookup ☐

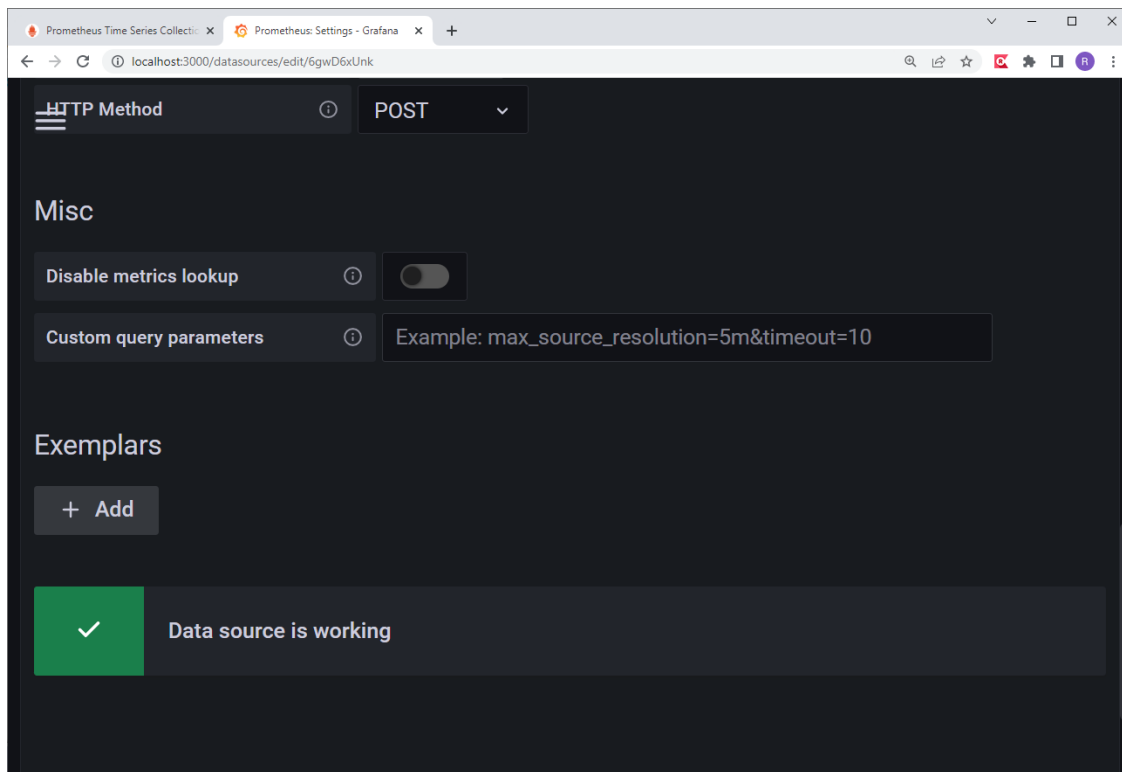
Custom query parameters

Exemplars

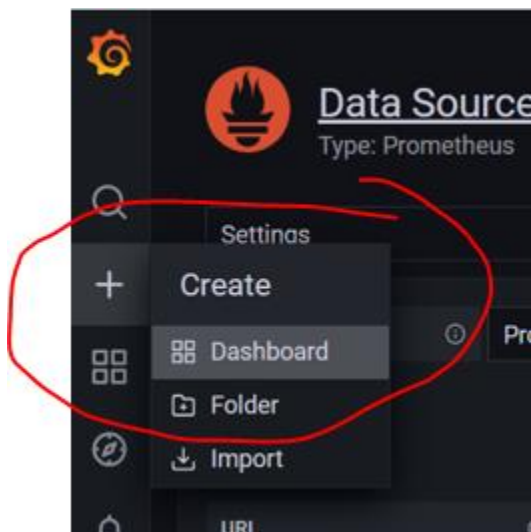
+ Add

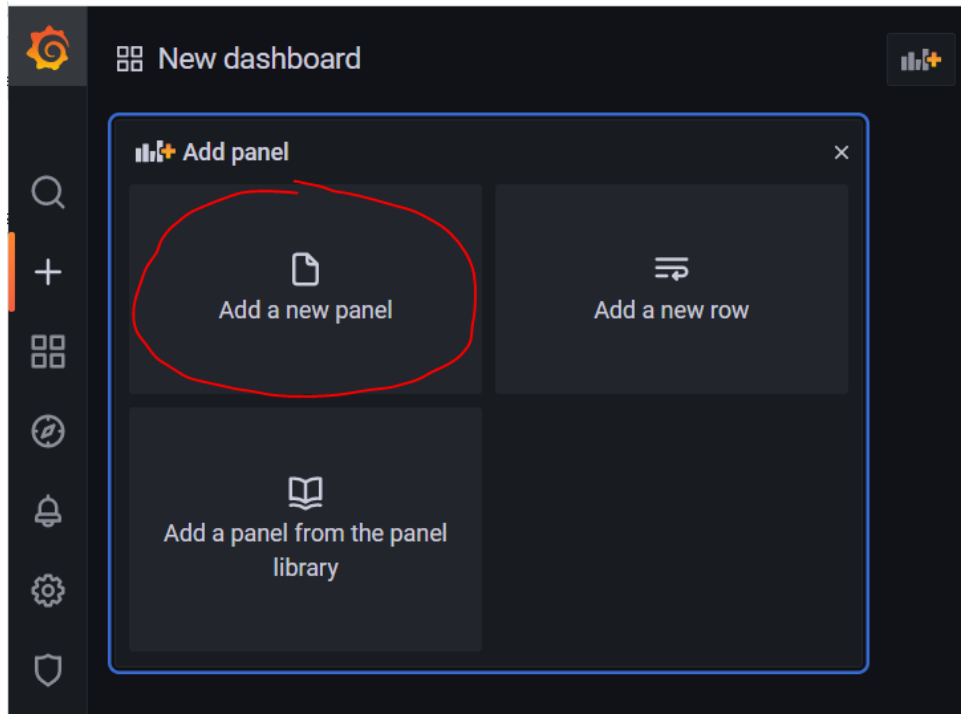
Back Explore Delete Save & test

Documentation | Support | Community | Enterprise (Free & unlicensed) | v8.4.6 (c53173ff6)

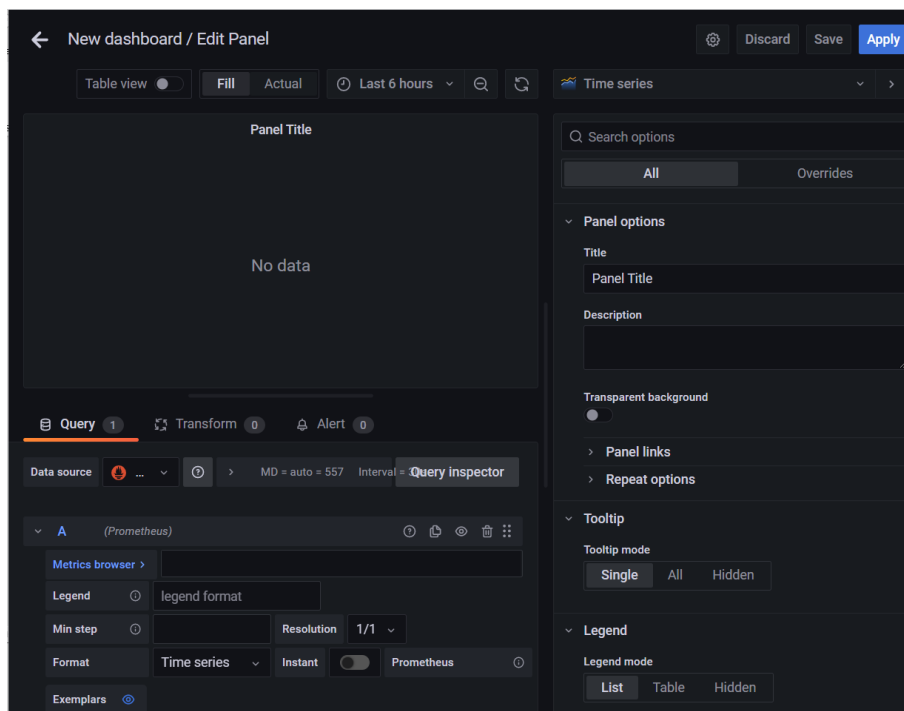


Then select + -> **Dashboard** in the Grafana menu

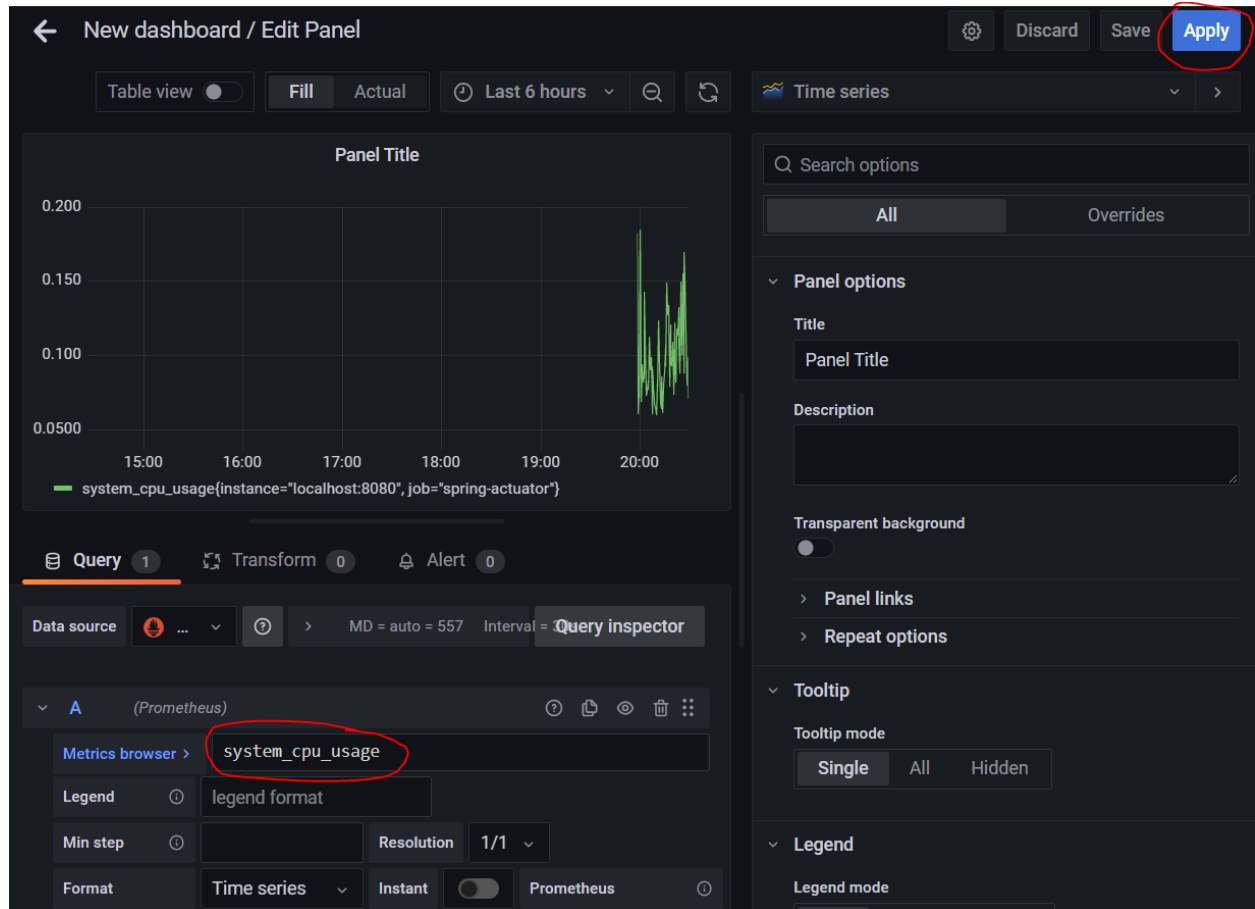




Then click Add a new panel



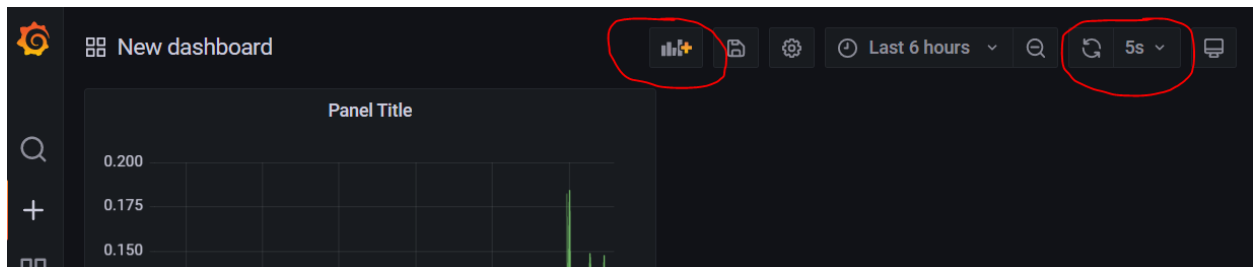
Now we can copy metric names that we used in Prometheus and type it in the metric box.



When you add `system_cpu_usage` in the Metrics field and select the Query inspector button, the graph will be shown.

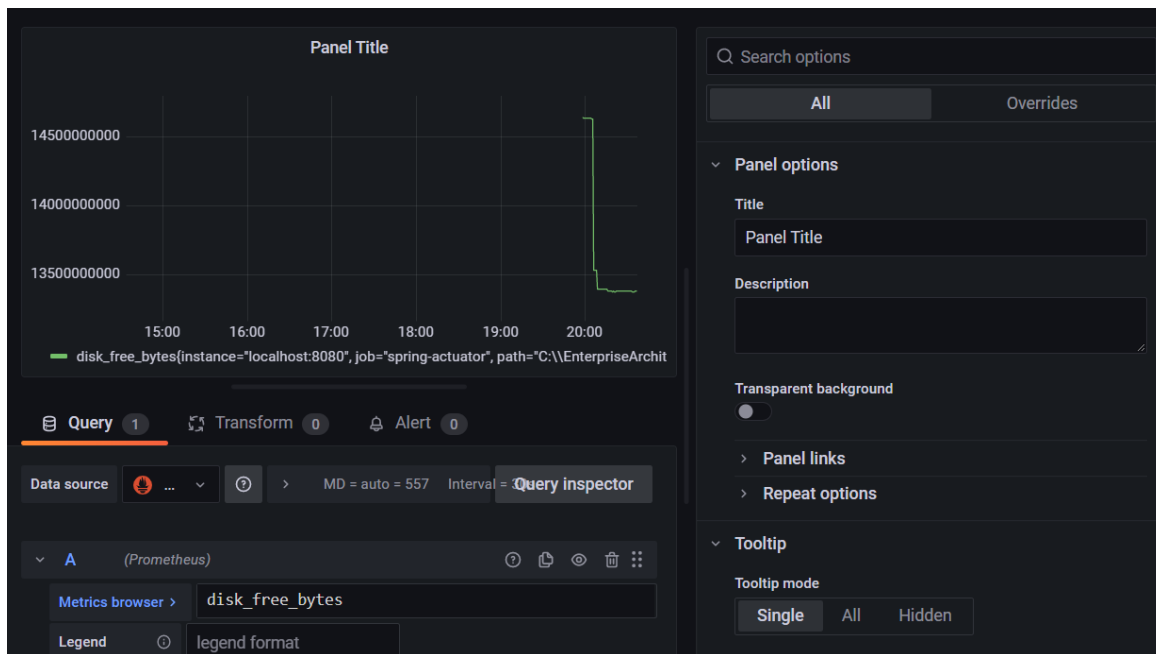
When you click the Apply button, this graph is added to our dashboard.



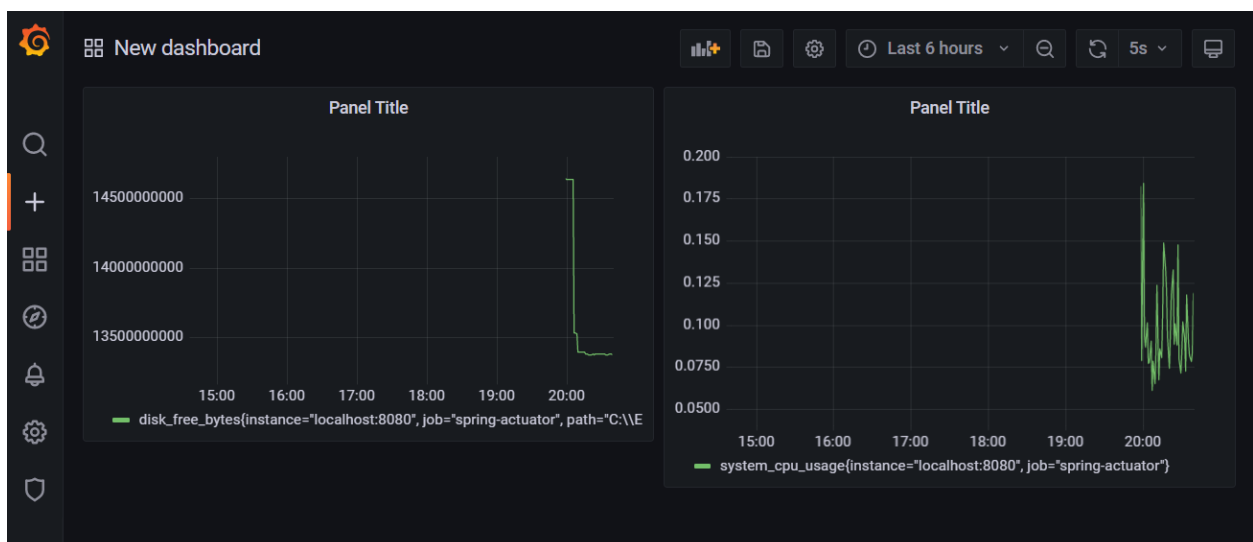


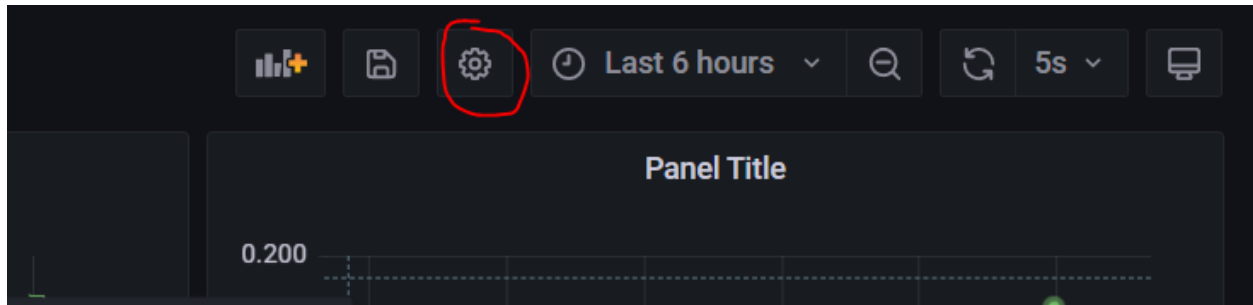
Let the dashboard update itself every 5 seconds

Then click the add new panel button

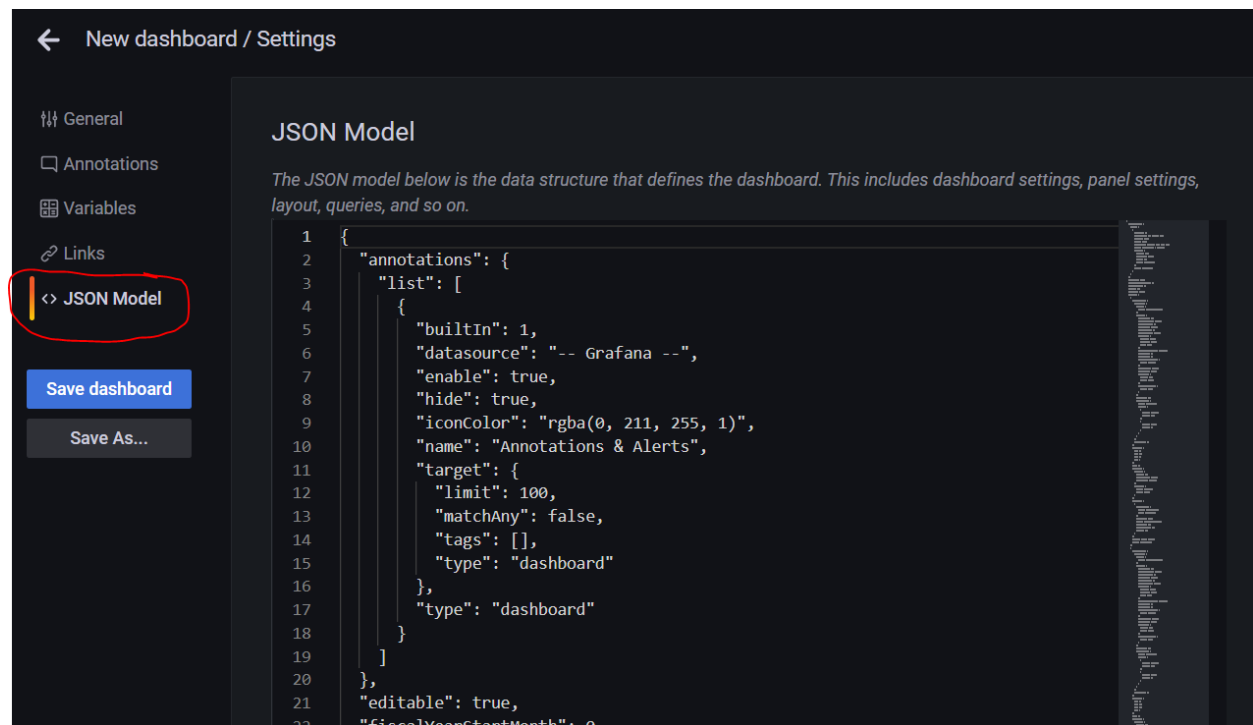


Create the **disk_free_bytes** graph and add it to the dashboard.





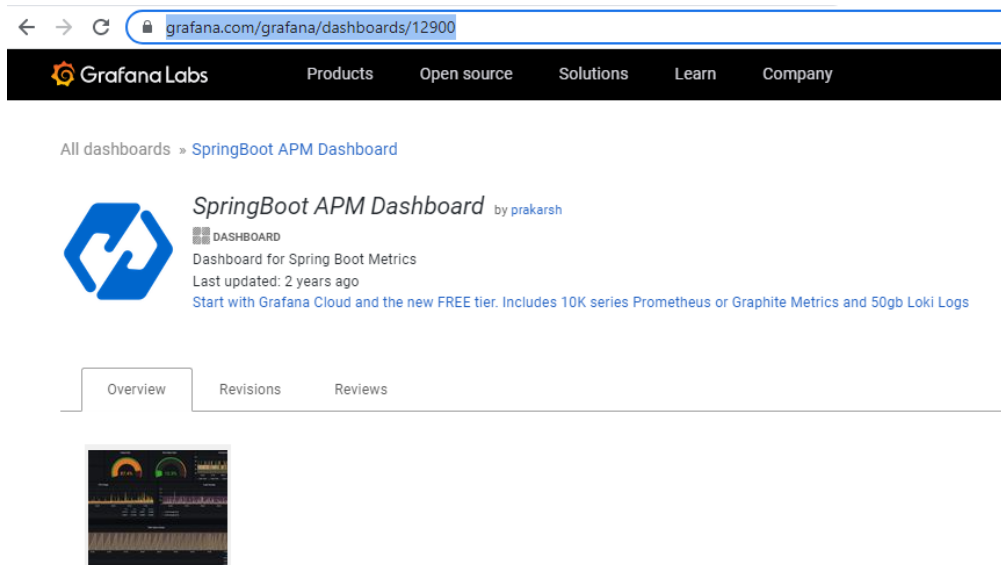
Select the dashboard setting button.



Then select the **JSON Model** option

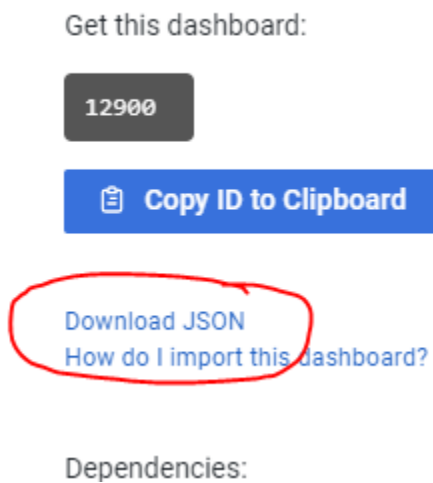
Now you can see the json code for our dashboard. We can save now our dashboard.

Now in the browser go to <https://grafana.com/grafana/dashboards/12900>

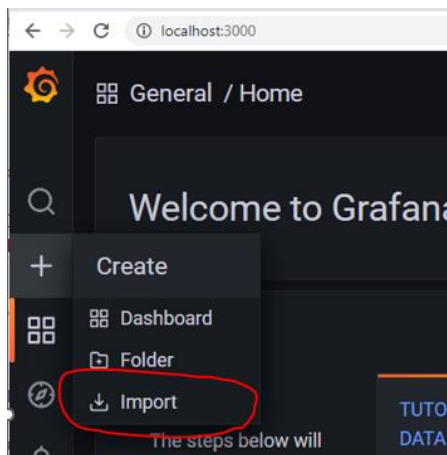


The screenshot shows the Grafana Labs website. The browser address bar displays grafana.com/grafana/dashboards/12900. The navigation bar includes links for Products, Open source, Solutions, Learn, and Company. The main content area features the title "SpringBoot APM Dashboard" by prakarsh, with a description: "Dashboard for Spring Boot Metrics", "Last updated: 2 years ago", and "Start with Grafana Cloud and the new FREE tier. Includes 10K series Prometheus or Graphite Metrics and 50gb Loki Logs". Below the title are tabs for Overview, Revisions, and Reviews. A thumbnail image of the dashboard is shown below the tabs.

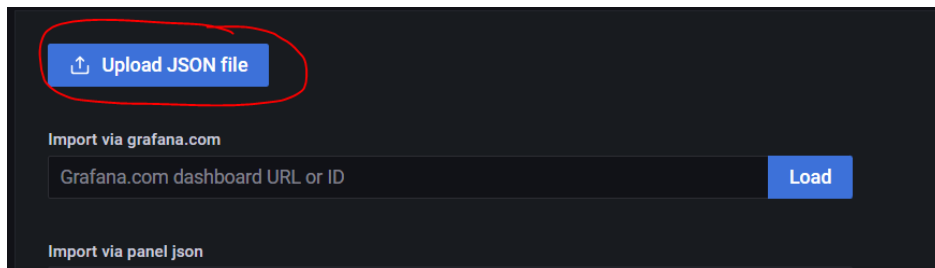
Click the **Download JSON** link



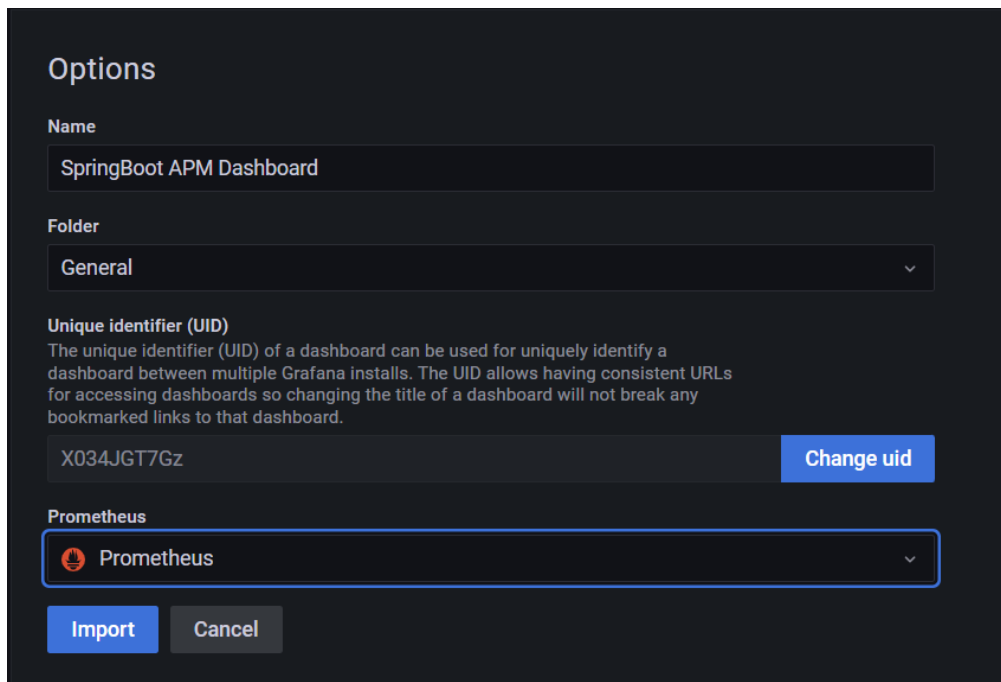
The screenshot shows the "Get this dashboard:" section of the Grafana Labs website. It includes a dark grey button with the ID "12900", a blue button labeled "Copy ID to Clipboard", and a blue link labeled "Download JSON" which is circled in red. Below the link is the text "How do I import this dashboard?". The "Dependencies:" section is visible at the bottom.



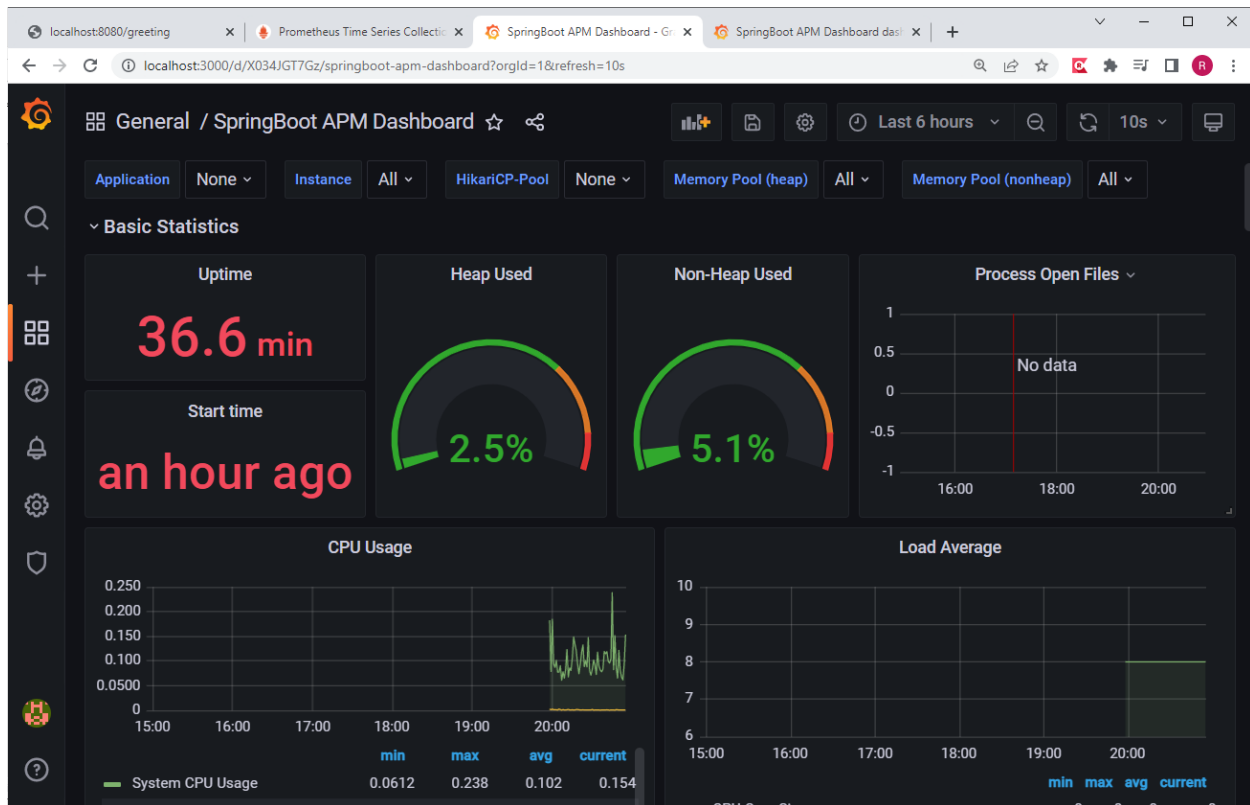
In Grafana, select + -> **Import**



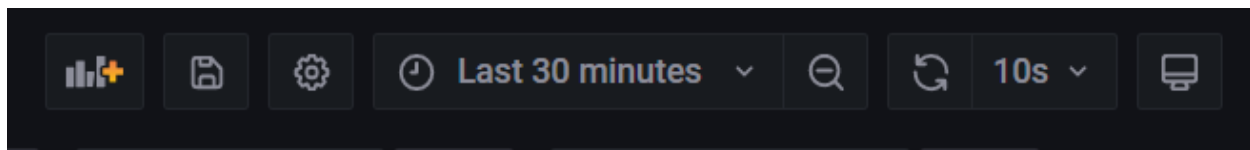
Click the **Upload JSON file** button.



Select the just downloaded JSON file and select Prometheus as data source. Then click **Import**.



We have now a nice dashboard for our application.



You can now play a little bit with the dashboard.

Now modify the application as follows:

```
@RestController
public class GreetingController {
    Logger logger = LoggerFactory.getLogger(GreetingController.class);

    @RequestMapping(value="/greeting")
    public String greeting() {
        logger.info("An INFO Message");
        logger.warn("A WARN Message");
        logger.error("An ERROR Message");
        return "Hello World";
    }
}
```

Restart the application and call the endpoint a few times.



Now you see the graphs of the logs change

Part D

Suppose you need to test the current Bank application. Elaborate what elements of the application you need to test. In other words, how would you test this application?

Part E

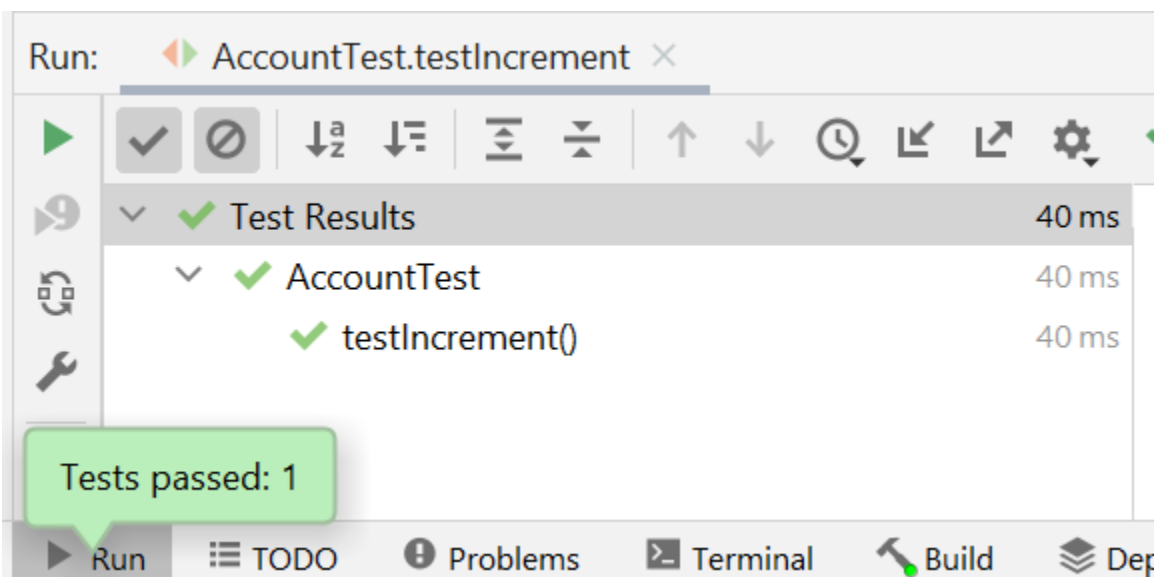
In the Bank application create the Class AccountTest in the package bank in src/test/java

```
package bank;

import bank.domain.Account;
import org.junit.jupiter.api.Test;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.Matchers.*;

public class AccountTest {
    @Test
    public void testIncrement() {
        Account account = new Account();
        account.deposit(100.0);
        assertThat( account.getBalance(), closeTo (100.0, 0.01));
    }
}
```

Run the test.



Write more unit tests for the Account class.

What to hand in?

- A zip file for part A
- Make screenshots of the actuator output for part B and put them in a word file.
- Make a screenshot of the Grafana dashboard of part C
- A document with the answer of part D.
- A zip file for part E