Article
Mikhail Khomenko · May 15, 2017 12m read

Making Prometheus Monitoring for InterSystems IRIS and Caché

Prometheus is one of the monitoring systems adapted for collecting time series data.

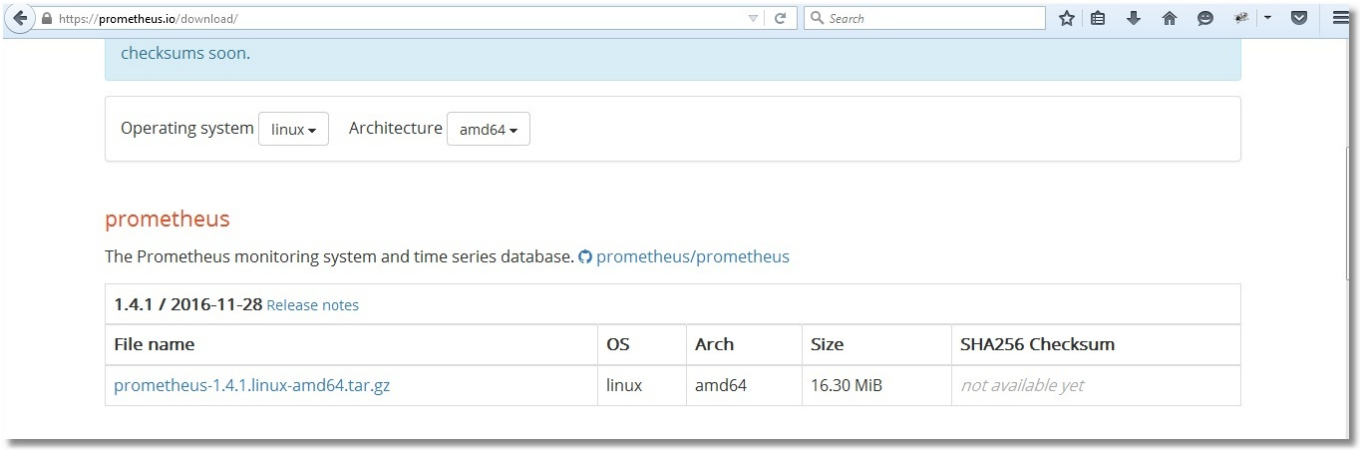
Its installation and initial configuration are relatively easy. The system has a built-in graphic subsystem called PromDash for visualizing data, but developers recommend using a free third-party product called Grafana. Prometheus can monitor a lot of things (hardware, containers, various DBMS's), but in this article, I would like to take a look at the monitoring of a Caché instance (to be exact, it will be an Ensemble instance, but the metrics will be from Caché). If you are interested – read along.

In our extremely simple case, Prometheus and Caché will live on a single machine (Fedora Workstation 24 x8664). Caché version:

```
%SYS>write $zv
Cache for UNIX (Red Hat Enterprise Linux for x86-64) 2016.1 (Build 656U) Fri Mar 11 2016 17:58:47 EST
```

Installation and configuration

Let's download a suitable Prometheus distribution package from the official site and save it to the /opt/prometheus folder.



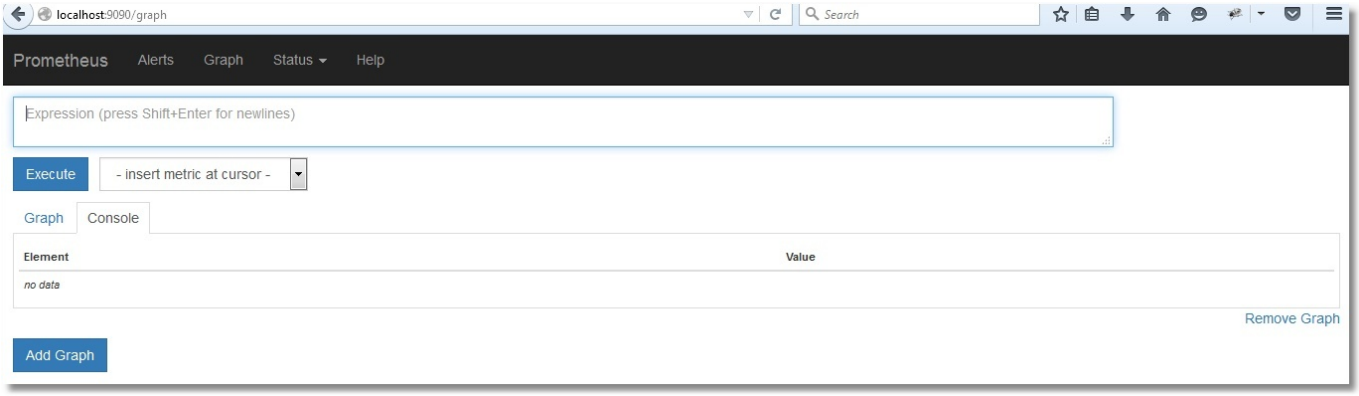
Unpack the archive, modify the template config file according to our needs and launch Prometheus. By default, Prometheus will be displaying its logs right in the console, which is why we will be saving its activity records to a log file.

Launching Prometheus

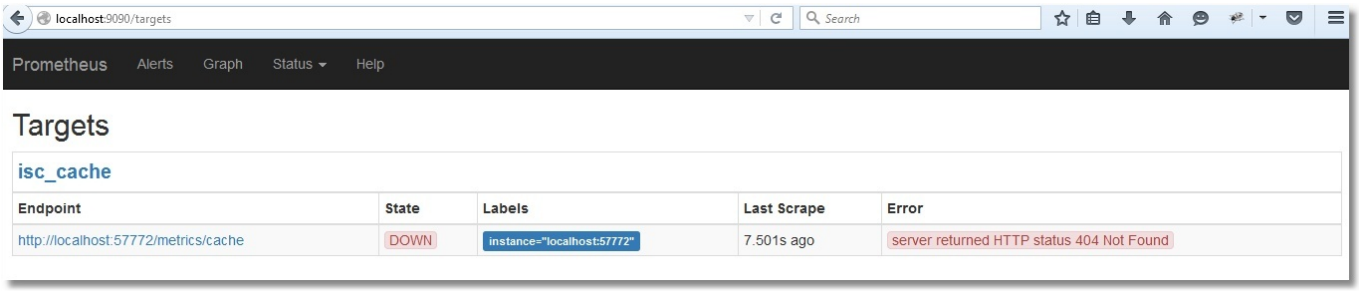
```
# pwd
/opt/prometheus
# ls
prometheus-1.4.1.linux-amd64.tar.gz
# tar -xzf prometheus-1.4.1.linux-amd64.tar.gz
# ls
prometheus-1.4.1.linux-amd64 prometheus-1.4.1.linux-amd64.tar.gz
# cd prometheus-1.4.1.linux-amd64/
# ls
consolelibraries consoles LICENSE NOTICE prometheus prometheus.yml promtool
# cat prometheus.yml
global:
```

```
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

scrape_configs:
  - job_name: 'isc_cache'
    metrics_path: '/metrics/cache'
    static_configs:
    - targets: ['localhost:57772']
# ./prometheus > /var/log/prometheus.log 2>&1 &
[1] 7117
# head /var/log/prometheus.log
time=«2017-01-01T09:01:11+02:00» level=info msg=«Starting prometheus (version=1.4.1, branch=master,
revision=2a89e8733f240d3cd57a6520b52c36ac4744ce12)» source=«main.go:77»
time=«2017-01-01T09:01:11+02:00» level=info msg=«Build context (go=go1.7.3, user=root@e685d23d8809, date=20161128-09:59:22)»
source=«main.go:78»
time=«2017-01-01T09:01:11+02:00» level=info msg=«Loading configuration file prometheus.yml» source=«main.go:250»
time=«2017-01-01T09:01:11+02:00» level=info msg=«Loading series map and head chunks...» source=«storage.go:354»
time=«2017-01-01T09:01:11+02:00» level=info msg=«23 series loaded.» source=«storage.go:359»
time=«2017-01-01T09:01:11+02:00» level=info msg="Listening on :9090" source=«web.go:248»
```

The prometheus.yml configuration is written in the YAML language, which doesn't like tabulation symbols, which is why you should use spaces only. We have already mentioned that metrics will be downloaded from http://localhost:57772 and we'll be sending requests to /metrics/cache (the name of the application is arbitrary), i.e. the destination address for collecting metrics will be http://localhost:57772/metrics/cache. A "job=isc_cache" tag will be added to each metric. A tag, very roughly, is the equivalent of WHERE in SQL. In our case, it won't be used, but will do just fine for more than one server. For example, names of servers (and/or instances) can be saved to tags and you can then use tags to parameterize requests for drawing graphs. Let's make sure that Prometheus is working (we can see the port it's listening to in the output above – 9090):

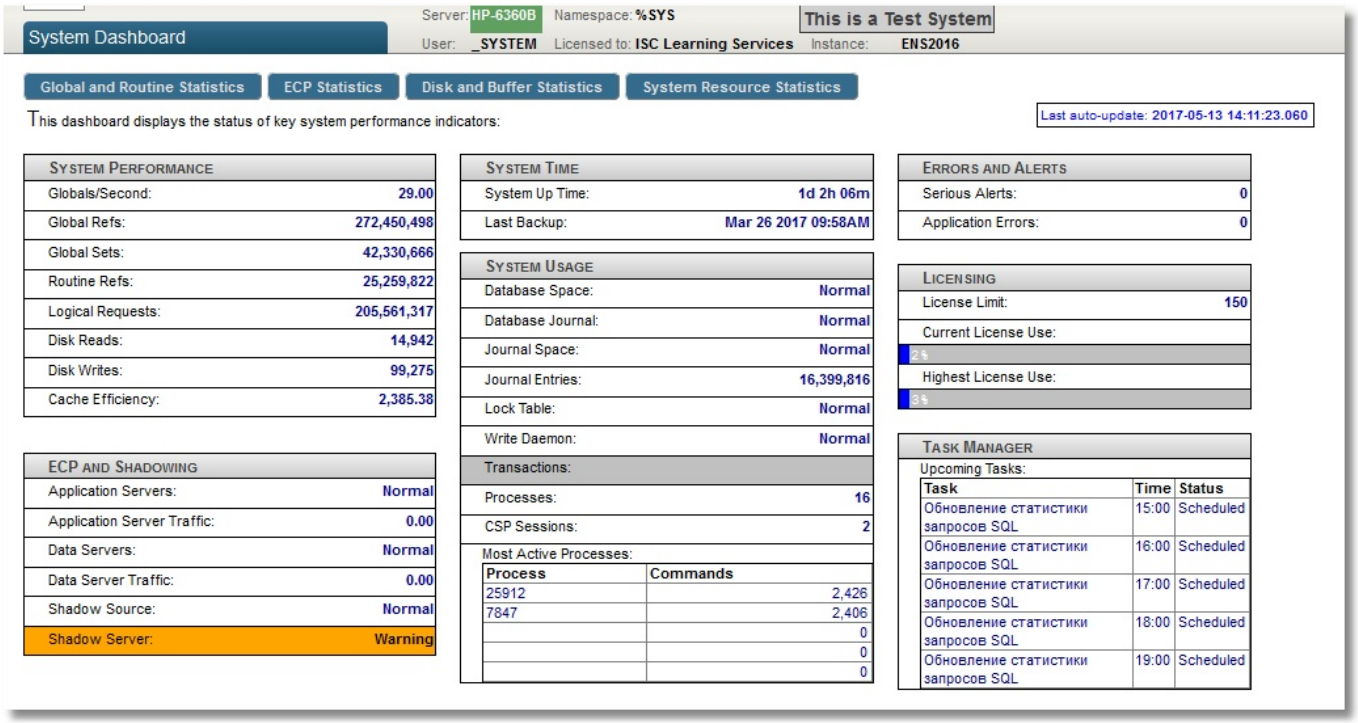


A web interface opens, which means that Prometheus is working. However, it doesn't see Caché metrics yet (let's check it by clicking Status → Targets):



Preparing metrics

Our task is to make metrics available to Prometheus in a suitable format at http://localhost:57772/metrics/cache. We'll be using the REST capabilities of Caché because of their simplicity. It should be noted that Prometheus only " understands " numeric metrics, so we will not export string metrics. To get the latter, we will use the API of the SYS.Stats.Dashboard class. These metrics are used by Caché itself for displaying the System toolbar:



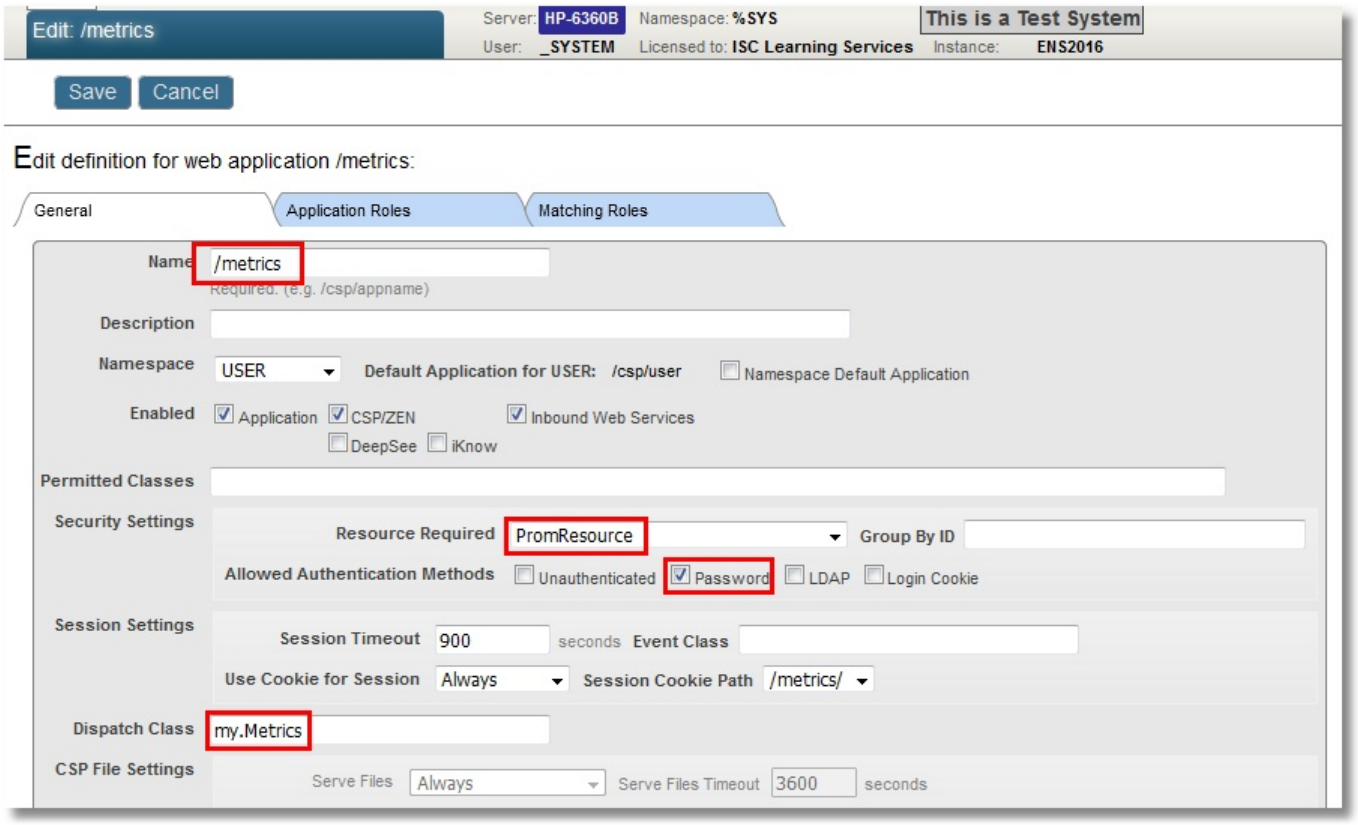
Example of the same in the Terminal:

```
%SYS>set dashboard = ##class(SYS.Stats.Dashboard).Sample()

%SYS>zwrite dashboard
dashboard=<OBJECT REFERENCE>[2@SYS.Stats.Dashboard]
+---------- general information ----------
| oref value: 2
| class name: SYS.Stats.Dashboard
| reference count: 2
+---------- attribute values ------------------
| ApplicationErrors = 0
| CSPSessions = 2
| CacheEfficiency = 2385.33
| DatabaseSpace = "Normal"
| DiskReads = 14942
| DiskWrites = 99278
| ECPAppServer = "OK"
| ECPAppSrvRate = 0
| ECPDataServer = "OK"
| ECPDataSrvRate = 0
| GloRefs = 272452605
| GloRefsPerSec = "70.00"
| GloSets = 42330792
| JournalEntries = 16399816
| JournalSpace = "Normal"
| JournalStatus = "Normal"
| LastBackup = "Mar 26 2017 09:58AM"
| LicenseCurrent = 3
| LicenseCurrentPct = 2
...
```

The USER space will be our sandbox. To begin with, let's create a REST application /metrics. To add come very basic security, let's protect our log-in with a password and associate the web application with some resource – let's call it PromResource. We need to disable public access to the resource, so let's do the following:

```
%SYS>write ##class(Security.Resources).Create("PromResource", "Resource for Metrics web page", "")
1
```

Our web app settings:



We will also need a user with access to this resource. The user should also be able to read from our database (USER in our case) and save data to it. Apart from this, this user will need read rights for the CACHESYS system database, since we will be switching to the %SYS space later in the code. We'll follow the standard scheme, i.e. create a PromRole role with these rights and then create a PromUser user assigned to this role. For password, let ' s use "Secret":

```
%SYS>write ##class(Security.Roles).Create("PromRole","Role for PromResource","PromResource:U,%DBUSER:RW,%DBCACHESYS:R")
1
%SYS>write ##class(Security.Users).Create("PromUser","PromRole","Secret")
1
```

It is this user PromUser that we will used for authentication in the Prometheus config. Once done, we'll re-read the config by sending a SIGNUP signal to the server process.

A safer config

```
# cat /opt/prometheus/prometheus-1.4.1.linux-amd64/prometheus.yml
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

scrape_configs:
  - job_name: 'isc_cache'
    metrics_path: '/metrics/cache'
    static_configs:
    - targets: ['localhost:57772']
    basic_auth:
      username: 'PromUser'
      password: 'Secret'
#
# kill -SIGHUP $(pgrep prometheus) # or kill -1 $(pgrep prometheus)
```

Prometheus can now successfully pass authentication for using the web application with metrics.

Metrics will be provided by the my.Metrics request processing class. Here is the implementation:

```
Class my.Metrics Extends %CSP.REST
```

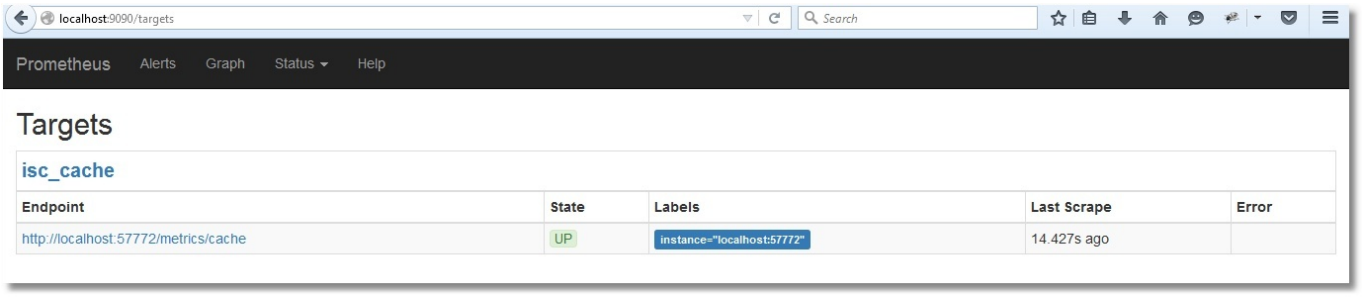
```
{

Parameter ISCPREFIX = "isc_cache";

Parameter DASHPREFIX = {..#ISCPREFIX_"_dashboard"};

XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
<Route Url="/cache" Method="GET" Call="getMetrics"/>
</Routes>
}

/// Output should obey the Prometheus exposition formats. Docs:
/// https://prometheus.io/docs/instrumenting/exposition_formats/
///
/// The protocol is line-oriented. A line-feed character (\n) separates lines.
/// The last line must end with a line-feed character. Empty lines are ignored.
ClassMethod getMetrics() As %Status
{
 set nl = $c(10)
 do ..getDashboardSample(.dashboard)
 do ..getClassProperties(dashboard.%ClassName(1), .propList, .descrList)

 for i=1:1:$ll(propList) {
  set descr = $lg(descrList,i)
  set propertyName = $lg(propList,i)
  set propertyValue = $property(dashboard, propertyName)

  // Prometheus supports time series database
  // so if we get empty (for example, backup metrics) or non-digital metrics
  // we just omit them.
  if ((propertyValue '= "") && ('$match(propertyValue, ".*[-A-Za-z ]+.*"))) {
   set metricsName = ..#DASHPREFIX_..camelCase2Underscore(propertyName)
   set metricsValue = propertyValue

   // Write description (help) for each metrics.
   // Format is that the Prometheus requires.
   // Multiline descriptions we have to join in one string.
   write "# HELP "_metricsName_" "_$replace(descr,nl," ")_nl
   write metricsName_" "_metricsValue_nl
  }
 }

 write nl
 quit $$$OK
}

ClassMethod getDashboardSample(Output dashboard)
{
 new $namespace
 set $namespace = "%SYS"
 set dashboard = ##class(SYS.Stats.Dashboard).Sample()
}

ClassMethod getClassProperties(className As %String, Output propList As %List, Output
 descrList As %List)
{
 new $namespace
```

```
 set $namespace = "%SYS"

 set propList = "", descrList = ""
 set properties = ##class(%Dictionary.ClassDefinition).%OpenId(className).Properties

 for i=1:1:properties.Count() {
  set property = properties.GetAt(i)
  set propList = propList_$lb(property.Name)
  set descrList = descrList_$lb(property.Description)
 }
}

/// Converts metrics name in camel case to underscore name with lower case
/// Sample: input = WriteDaemon, output = _write_daemon
ClassMethod camelCase2Underscore(metrics As %String) As %String
{
 set result = metrics
 set regexp = "([A-Z])"
 set matcher = ##class(%Regex.Matcher).%New(regexp, metrics)
 while (matcher.Locate()) {
  set result = matcher.ReplaceAll("_"_"$1")
 }

 // To lower case
 set result = $zcvt(result, "l")

 // _e_c_p (_c_s_p) to _ecp (_csp)
 set result = $replace(result, "_e_c_p", "_ecp")
 set result = $replace(result, "_c_s_p", "_csp")

 quit result
}
}
```

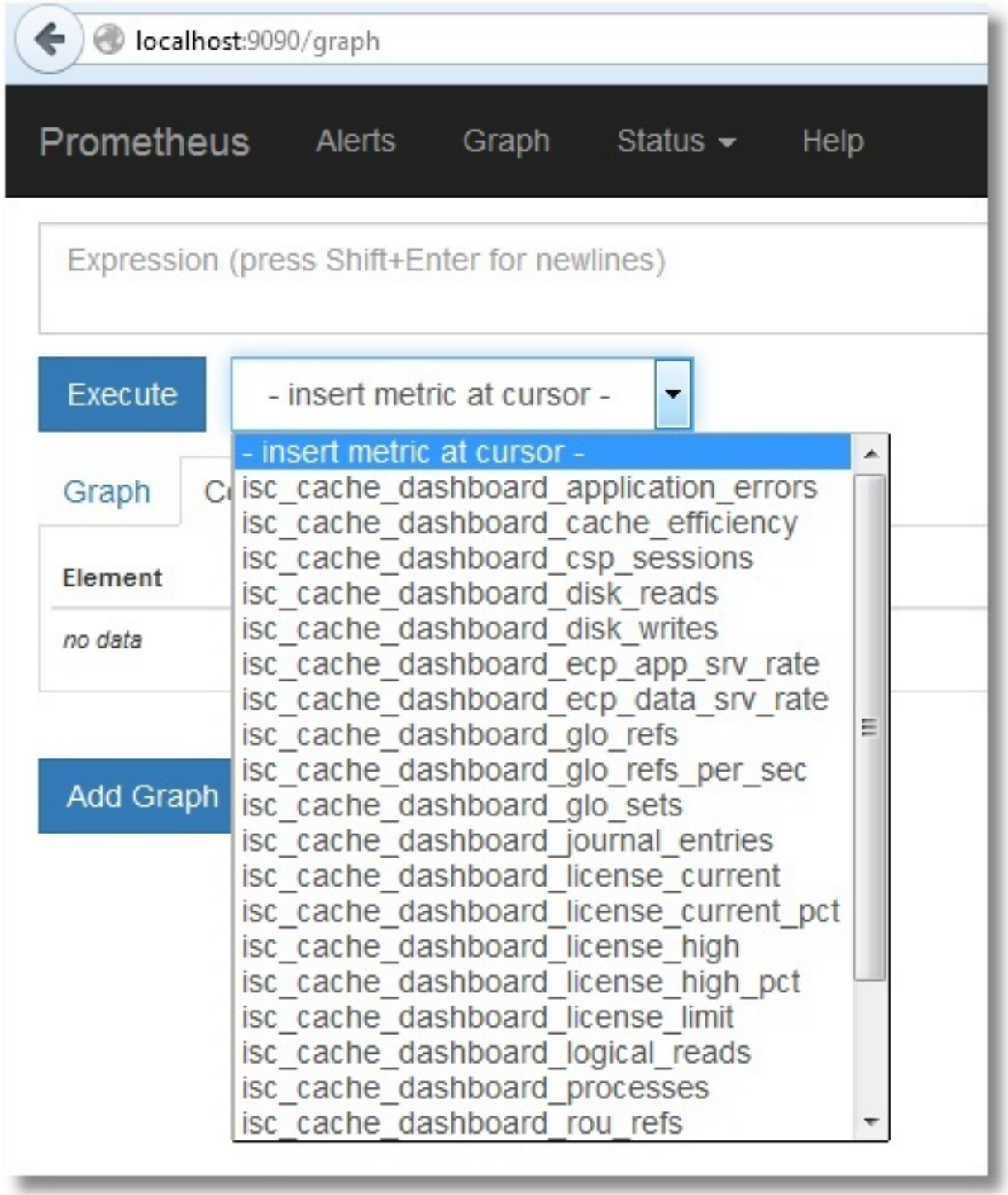
Let's use the console to check that our efforts have not been vain (added the --silent key, so that curl doesn't impede us with its progress bar):

```
# curl --user PromUser:Secret --silent -XGET 'http://localhost:57772/metrics/cache' | head -20
# HELP iscachedashboardapplicationerrors Number of application errors that have been logged.
iscachedashboardapplicationerrors 0
# HELP iscachedashboardcspsessions Most recent number of CSP sessions.
iscachedashboardcspsessions 2
# HELP iscachedashboardcacheefficiency Most recently measured cache efficiency (Global references / (physical reads + writes))
iscachedashboardcacheefficiency 2378.11
# HELP iscachedashboarddiskreads Number of physical block read operations since system startup.
iscachedashboarddiskreads 15101
# HELP iscachedashboarddiskwrites Number of physical block write operations since system startup
iscachedashboarddiskwrites 106233
# HELP iscachedashboardecpappsrvrate Most recently measured ECP application server traffic in bytes/second.
iscachedashboardecpappsrvrate 0
# HELP iscachedashboardecpdatasrvrate Most recently measured ECP data server traffic in bytes/second.
iscachedashboardecpdatasrvrate 0
# HELP iscachedashboardglorefs Number of Global references since system startup.
iscachedashboardglorefs 288545263
# HELP iscachedashboardglorefspersec Most recently measured number of Global references per second.
iscachedashboardglorefspersec 273.00
# HELP iscachedashboardglosets Number of Global Sets and Kills since system startup.
iscachedashboardglosets 44584646
```

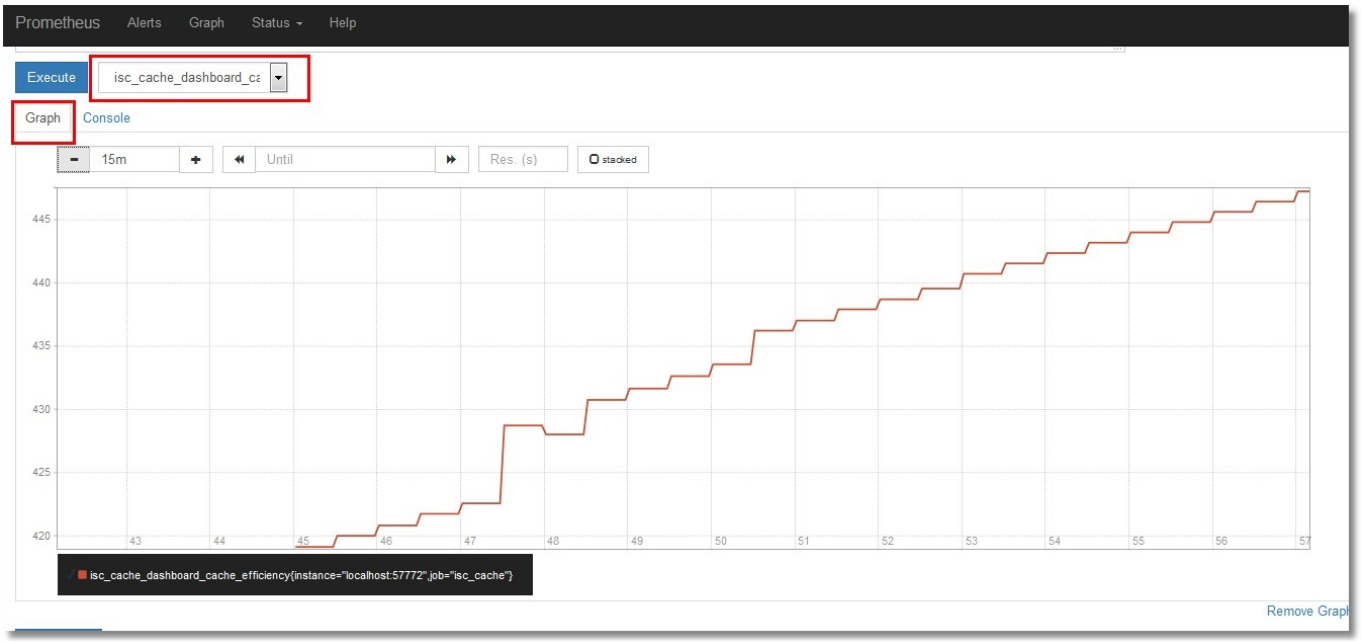
We can now check the same in the Prometheus interface:



And here is the list of our metrics:

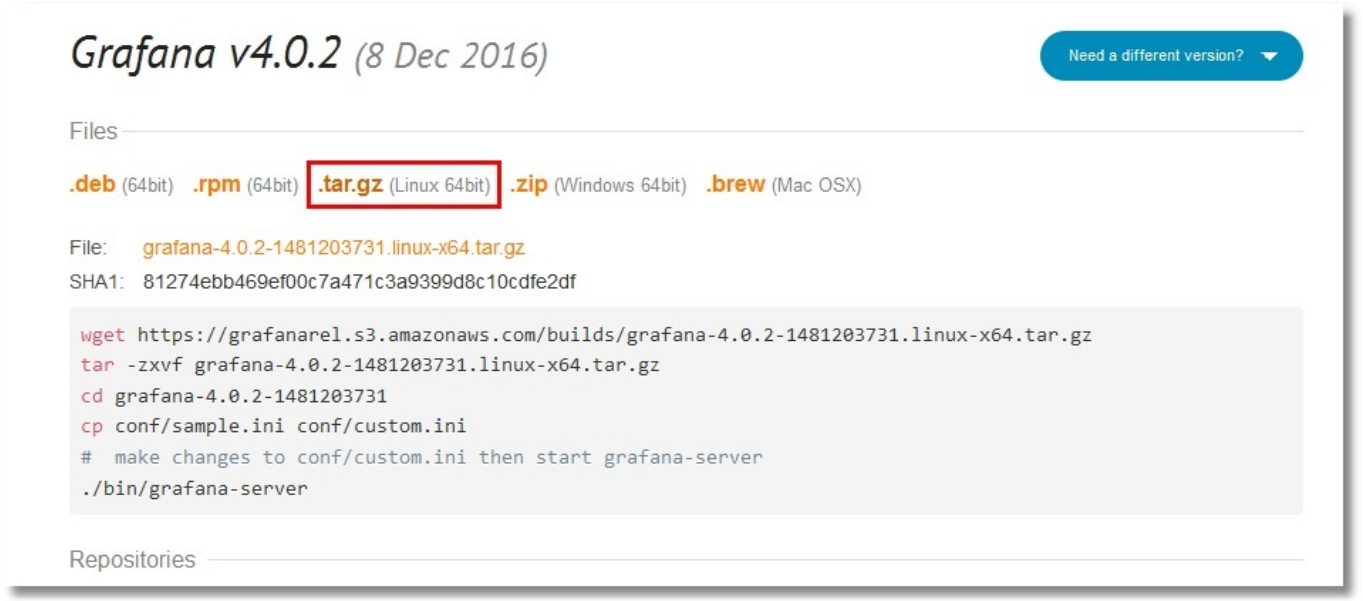


We won ' t focus on viewing them in Prometheus. You can select the necessary metric and click the " Execute " button. Select the " Graph " tab to see the graph (shows the cache efficiency):



Visualization of metrics

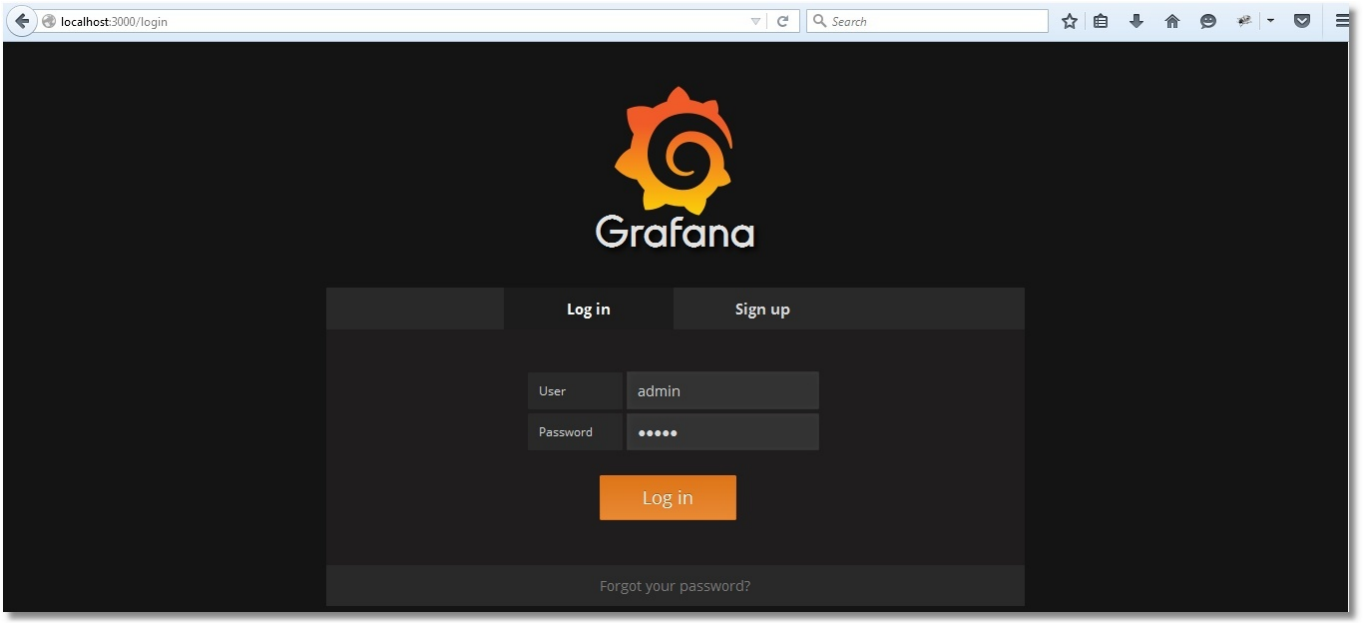
For visualization purposes, let ' s install Grafana. For this article, I chose installation from a tarball. However, there are other installation options, from packages to a container. Let's perform the following steps (after creating the /opt/grafana folder and switching to it):



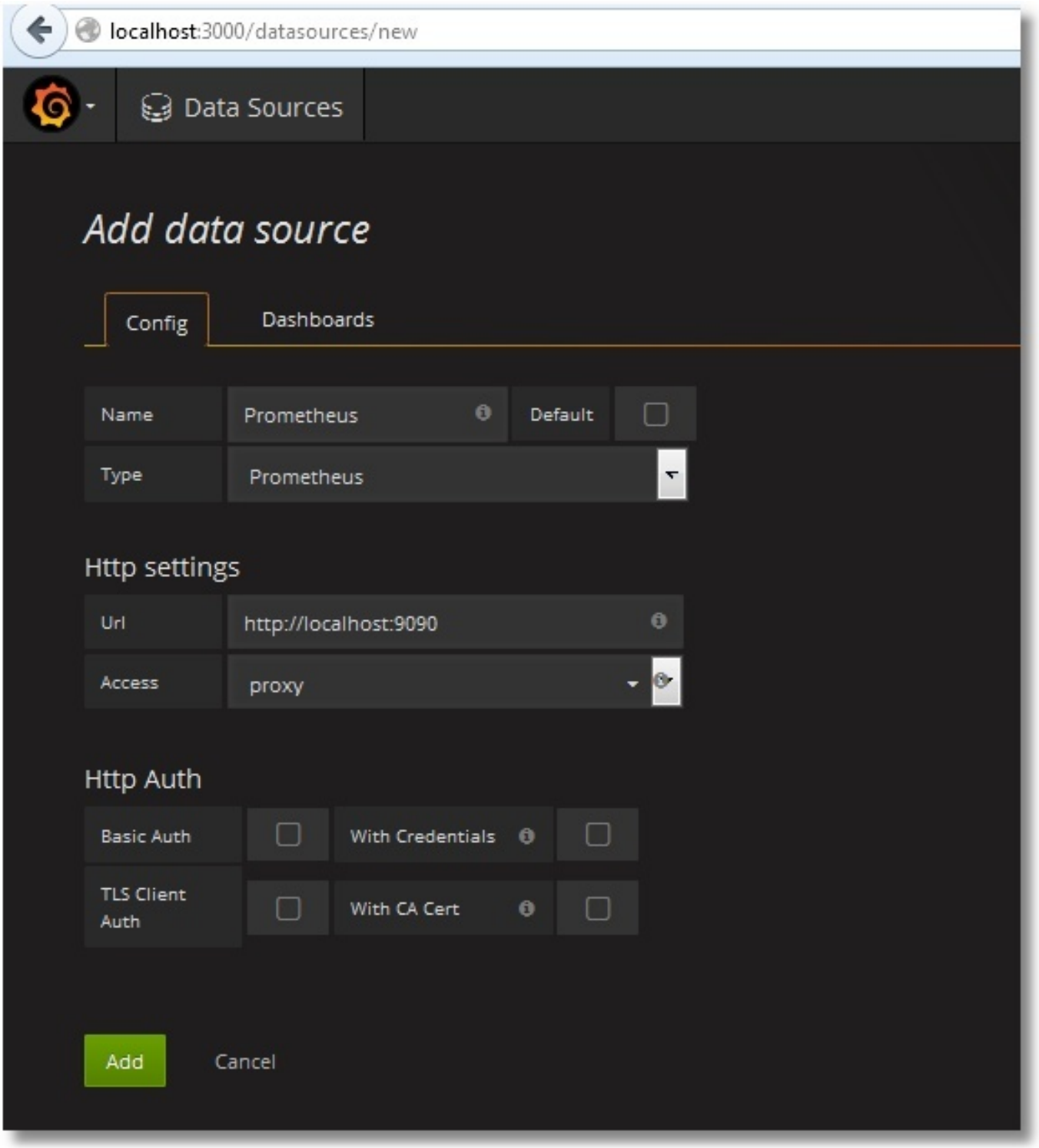
Let's leave the config unchanged for now. On our last step, we launch Grafana in the background mode. We'll be saving Grafan's log to a file, just like we did with Prometheus:

```
# ./bin/grafana-server > /var/log/grafana.log 2>&1 &
```

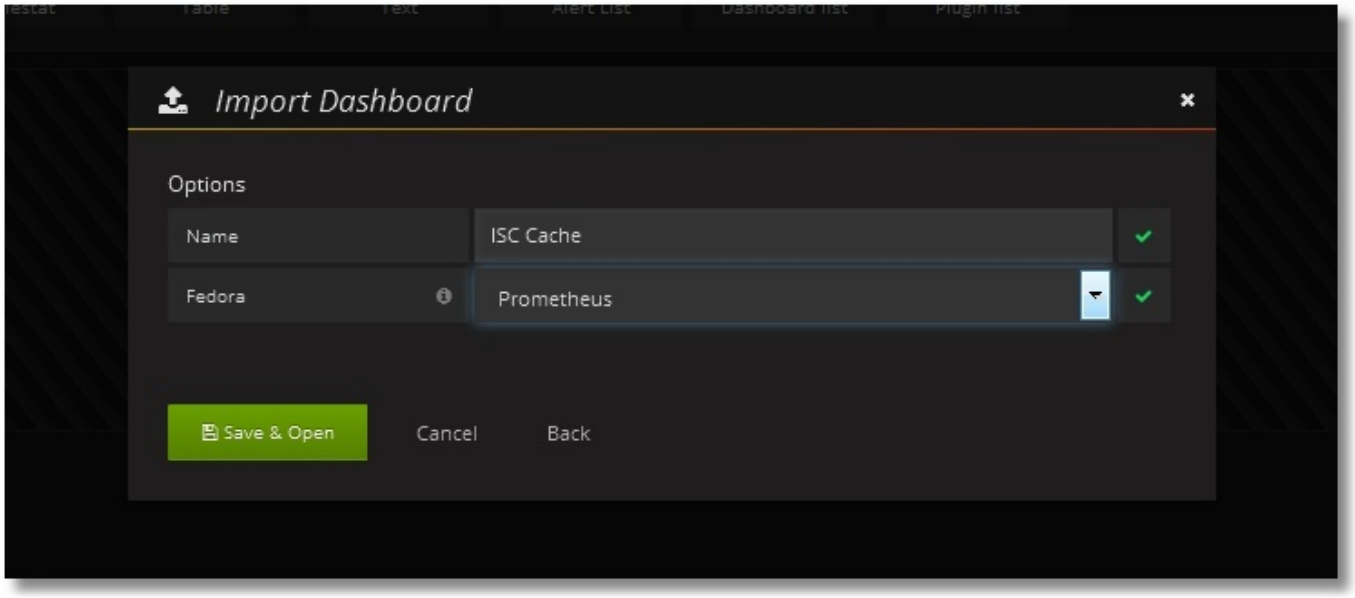
By default, Grafana ' s web interface is accessible via port 3000. Login/password: admin/admin.



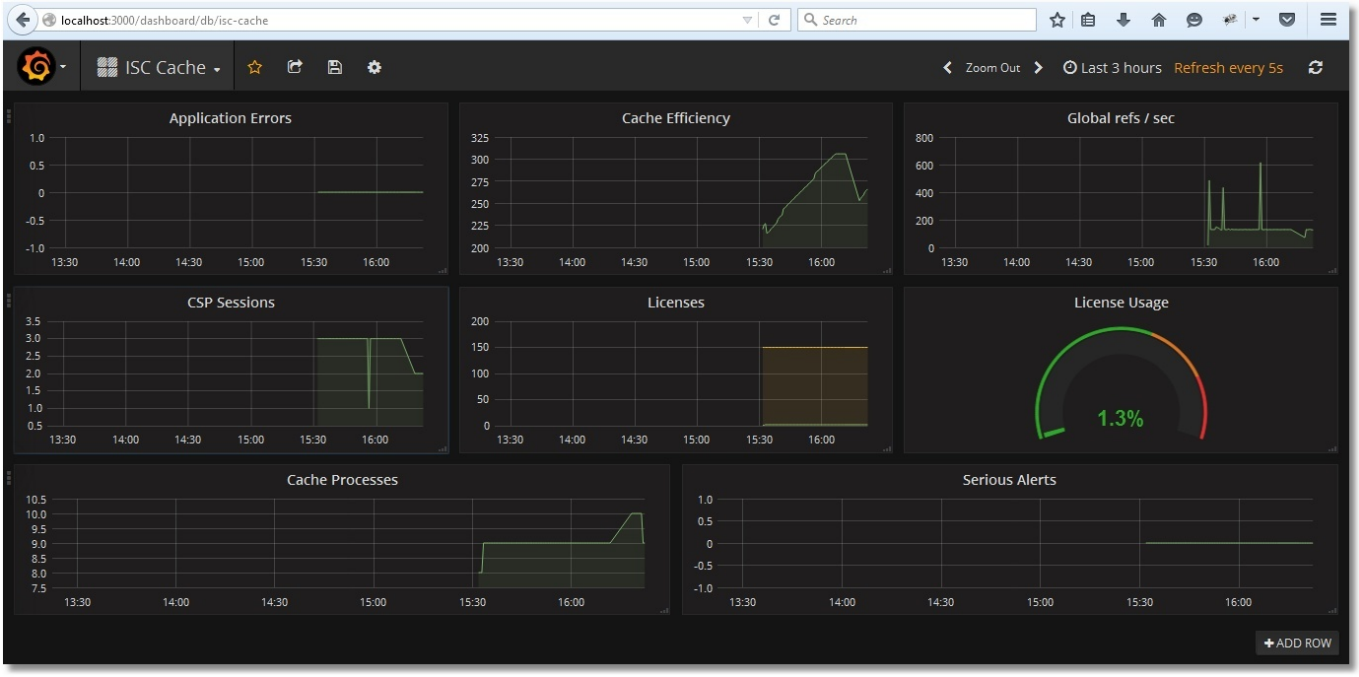
A detailed instruction on making Prometheus work with Grafana is available here. In short, we need to add a new Data Source of the Prometheus type. Select your option for direct/proxy access:



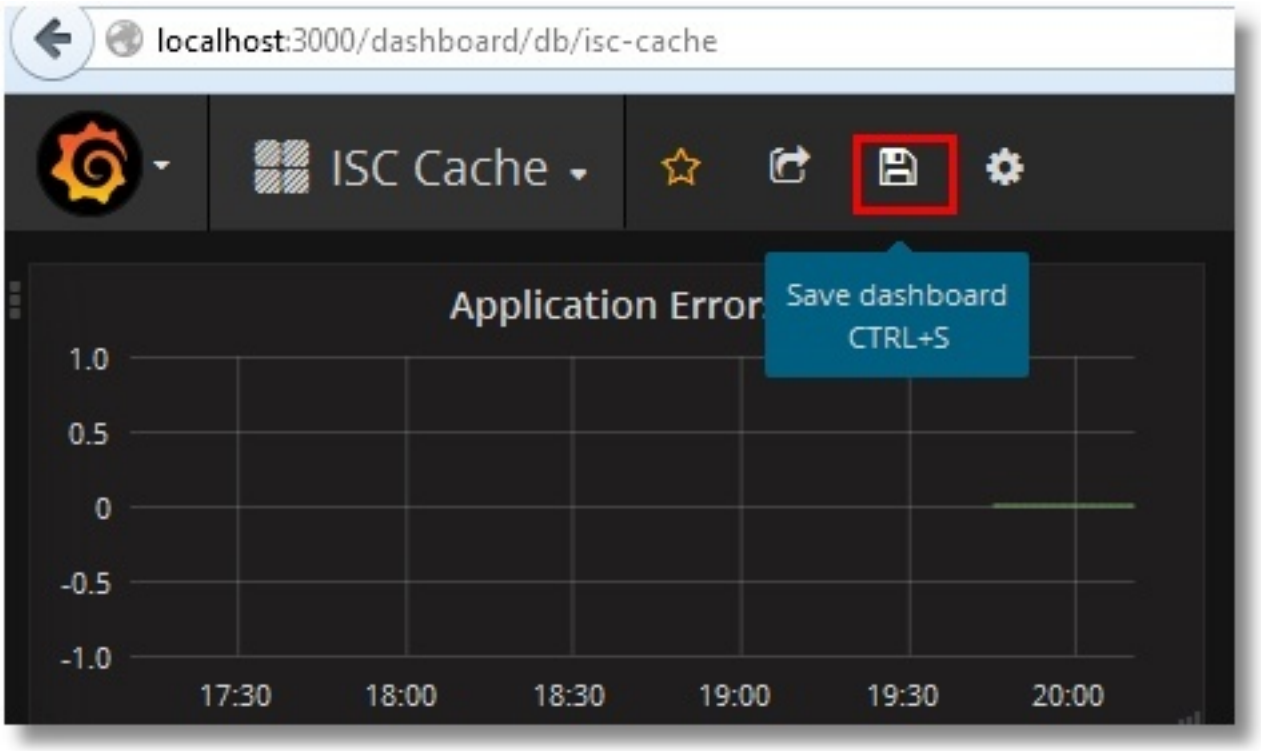
Once done, we need to add a dashboard with the necessary panels. The test sample of a dashboard is publicly available, along with the code of the metrics collection class. A dashboard can be simply imported to Grafana (Dashboards → Import):



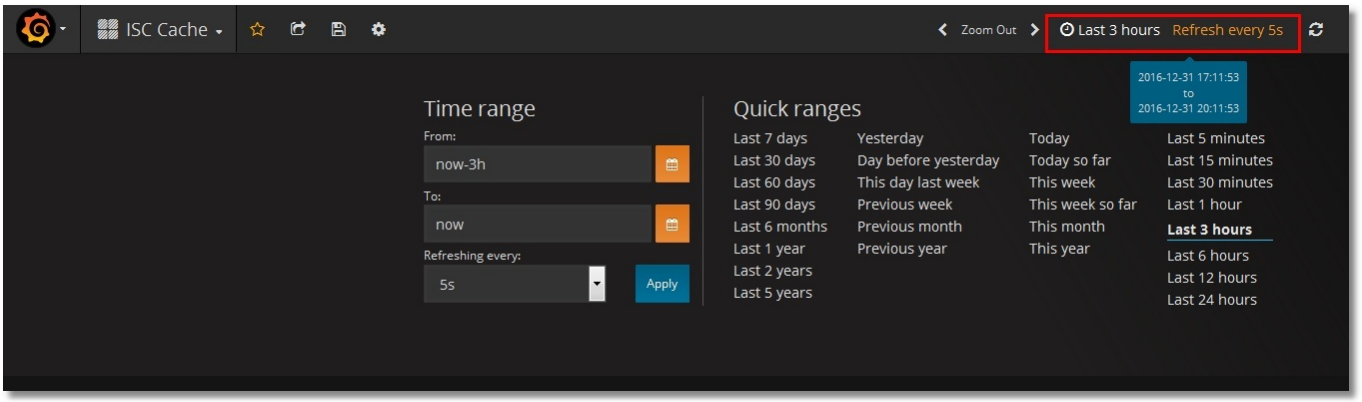
We'll get the following after import:



Save the dashboard:



Time range and update period can be selected in the top right corner:

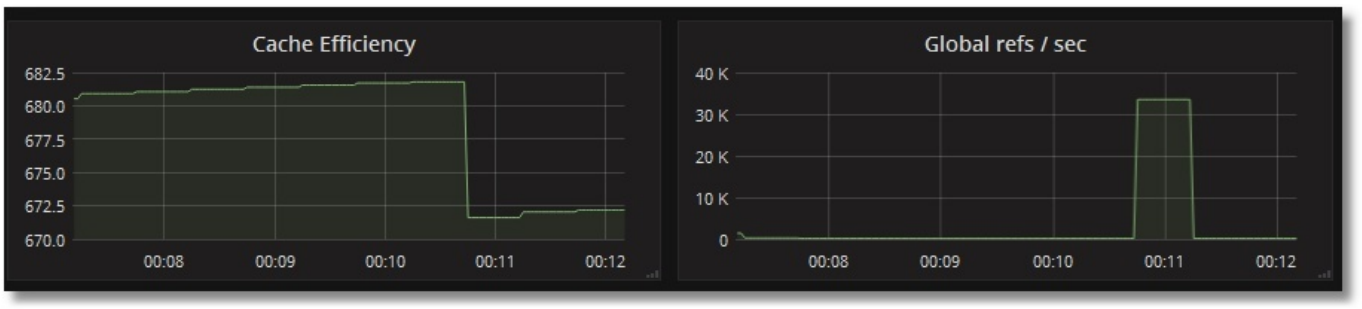


Examples of monitoring types

Let's test the monitoring of calls to globals:

```
USER>for i=1:1:1000000 {set ^prometheus(i) = i}
USER>kill ^prometheus
```

We can see that the number of references to globals per second has increased, while cache efficiency dropped (the ^Prometheus global hasn ' t been cached yet):



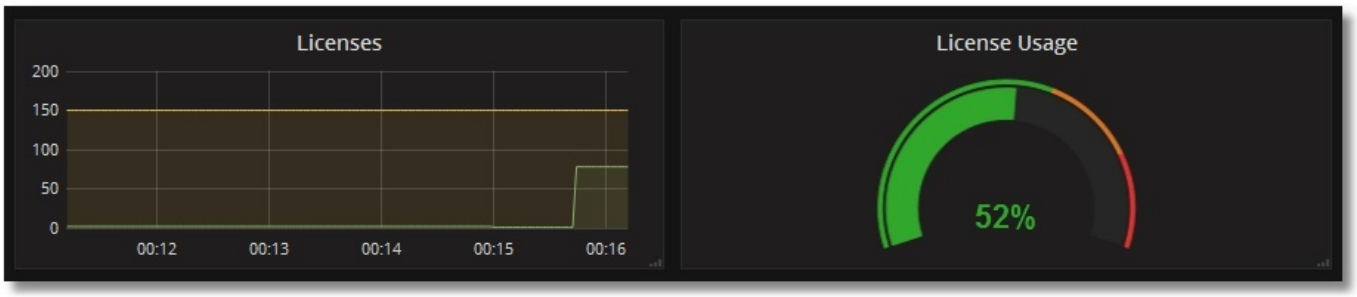
Let's check our license usage. To do this, let's create a primitive CSP page called PromTest.csp in the USER namespace:

```
<html>
<head><title>Prometheus Test Page</title></head>
<body>Monitoring works fine!</body>
</html>
```

And visit it so many times (we assume that the /csp/user application is not password-protected):

```
# ab -n77 http://localhost:57772/csp/user/PromTest.csp
```

We'll see the following picture for license usage:



Conclusions

As we can see, implementing the monitoring functionality is not hard at all. Even after a few initial steps, we can get important information about the work of the system, such as: license usage, efficiency of globals caching, application errors. We used the SYS.Stats.Dashboard for this tutorial, but other classes of SYS, %SYSTEM, %SYS packages also deserve attention. You can also write your own class that will supply custom metrics for your own application – for instance, the number of documents of a particular type. Some useful metrics will eventually be compiled into a separate template for Grafana.

To be continued

If you are interested in learning more about this, I will write more on the subject. Here are my plans:

1. Preparing a Grafana template with metrics for the logging daemon. It would be nice to make some sort of a graphical equivalent of the ^mgstat tool – at least for some of its metrics.
2. Password protection for web applications is good, but it would be nice to verify the possibility of using certificates.
3. Use of Prometheus, Grafana and some exporters for Prometheus as Docker containers.
4. Use of discovery services for automatically adding new Caché instances to the Prometheus monitoring list. It's also where I'd like to demonstrate (in practice) how convenient Grafana and its templates are. This is something like dynamic panels, where metrics for a particular selected server are shown, all on the same Dashboard.
5. Prometheus Alert Manager.
6. Prometheus configuration settings related to the duration of storing data, as well possible optimizations for systems with a large number of metrics and a short statistics collection interval.
7. Various subtleties and nuances that will transpire along the way.

Links

During the preparation of this article, I visited a number of useful sites and watched a great deal of videos:

- Prometheus project website
- Grafana project website
- Blog of one of Prometheus developers called Brian Brazil

- Tutorial on DigitalOcean
- Some videos from Robust Perception
- Many videos from a conference devoted to Prometheus

Thank you for reading all the way down to this line!

#Best Practices #Monitoring #System Administration #Visualization #Caché
