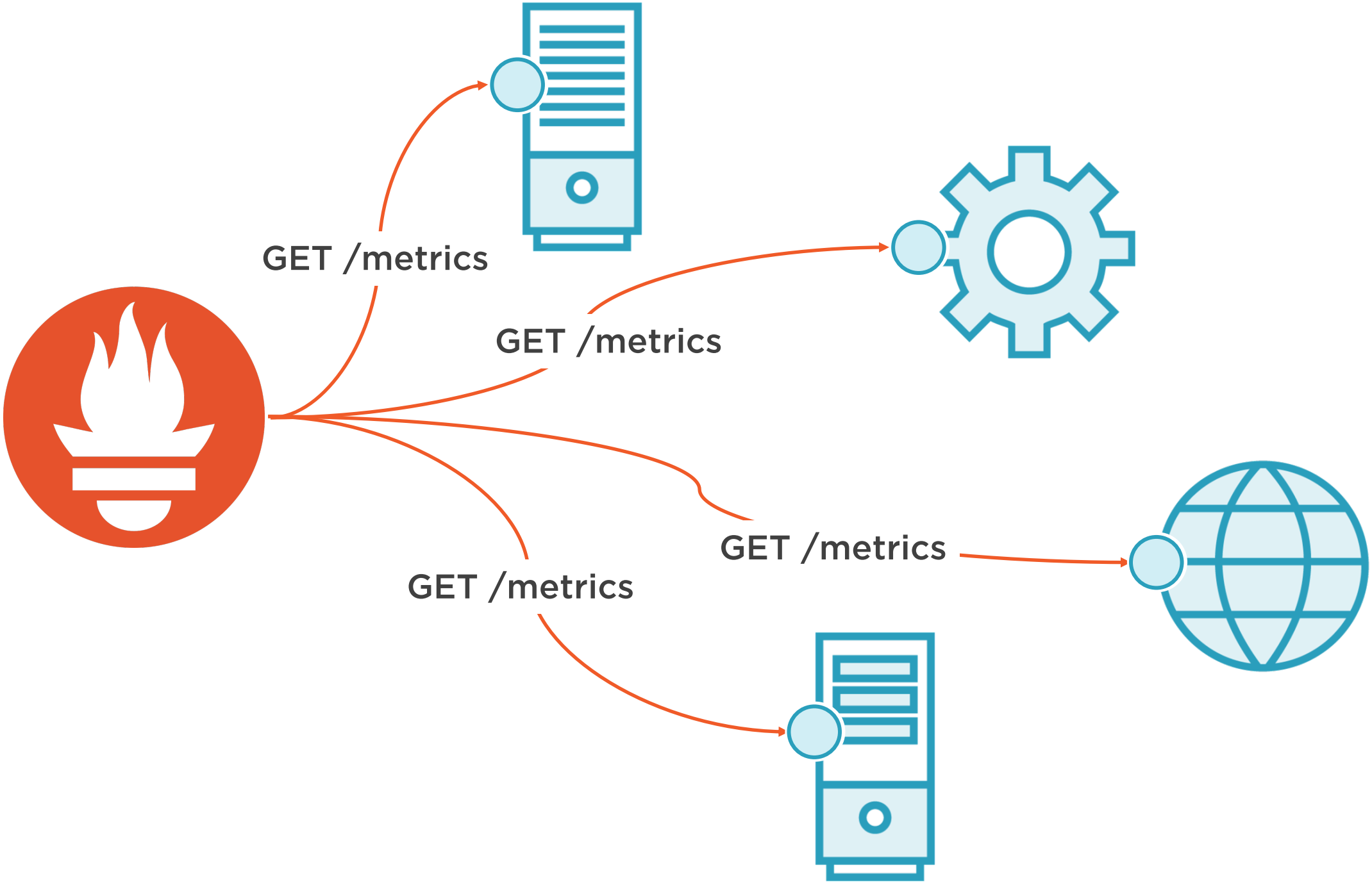
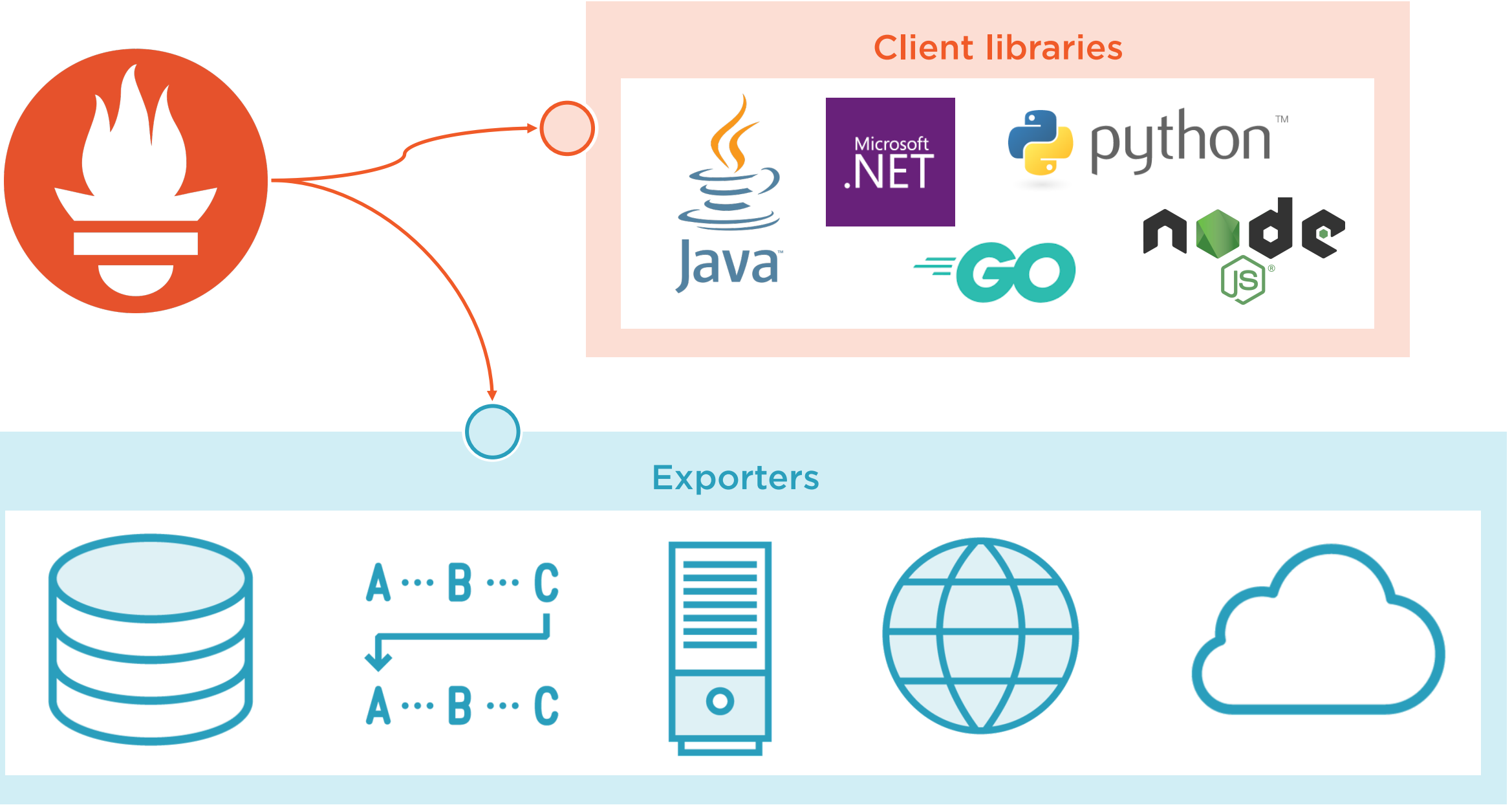
Instrumenting Applications with Metrics for Prometheus

ADDING INSTRUMENTATION WITH CLIENT LIBRARIES

Elton Stoneman
CONSULTANT & TRAINER

@EltonStoneman | blog.sixeyed.com


GET /metrics
GET /metrics
GET /metrics
GET /metrics


Client libraries
Java
Microsoft
.NET
python
GO
node
js
Exporters
A ··· B ··· C
A ··· B ··· C

Getting Started with Prometheus

★★★★☆ By Elton Stoneman

Prometheus is the preferred monitoring tool for containers, but it works just as well in any environment. This course will teach you how to get up and running with Prometheus and add a consistent monitoring approach to all your apps and servers.

Understanding How Prometheus Works

Demo: Counters and Gauges in Linux Apps	5m
Demo: Summaries and Histograms in Windows Apps	7m
Exploring the Prometheus Architecture	3m
Module Summary	2m
Understanding Labels and Data Granularity	5m
Understanding the Prometheus Metric Types	6m
What Makes Prometheus so Awesome?	4m

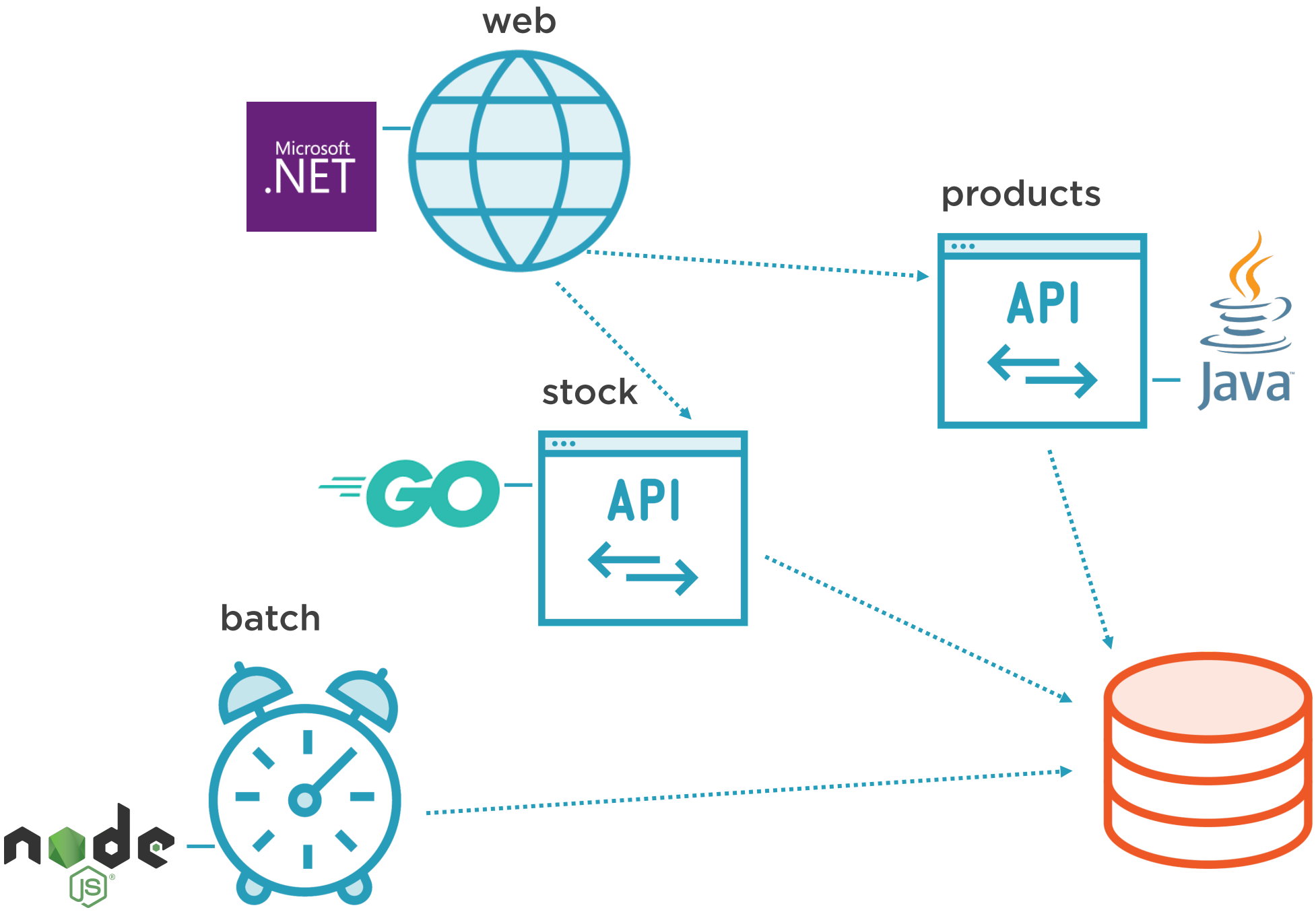
Course info

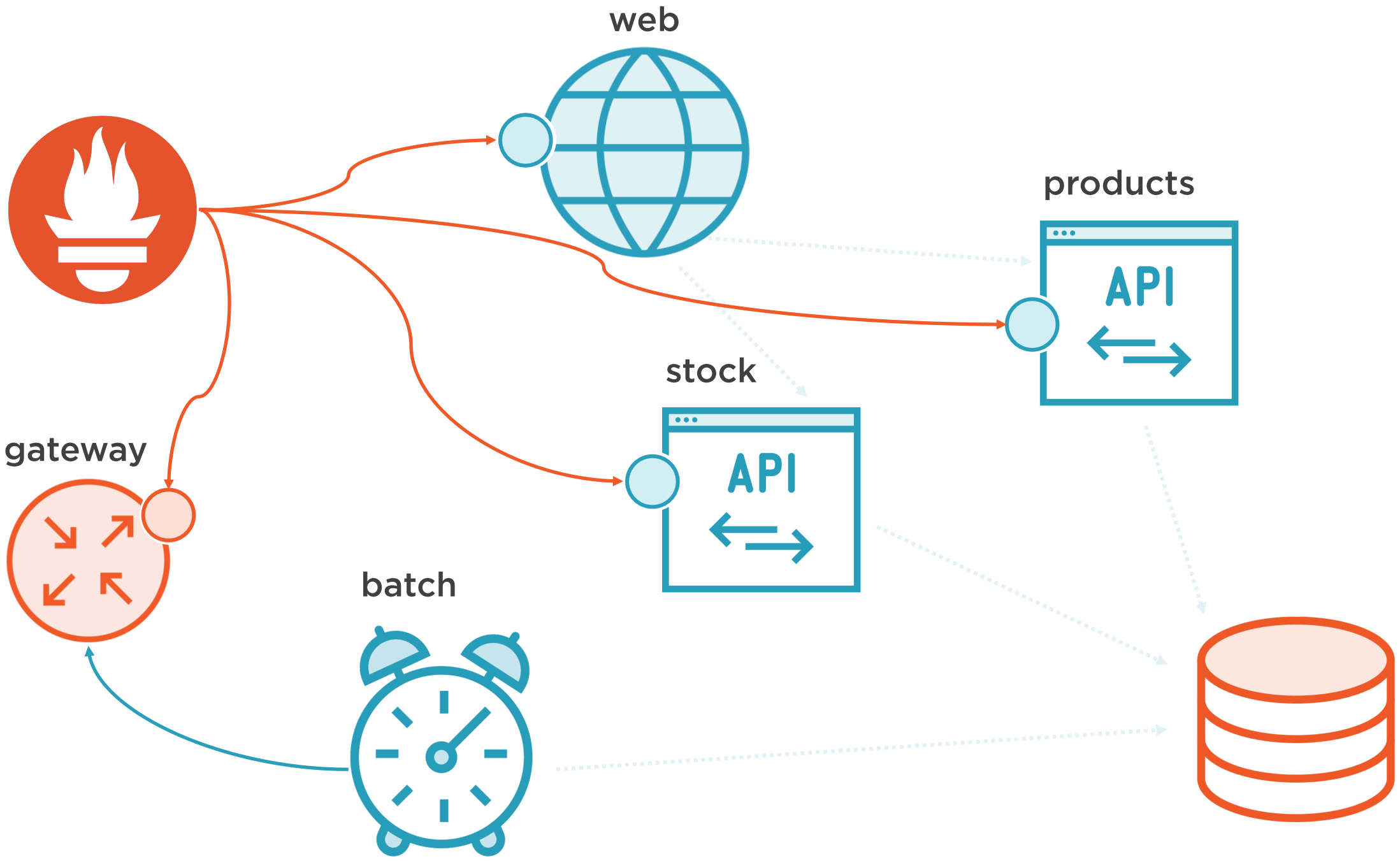
Rating	★★★★☆ (19)
Level	Beginner
Updated	Jun 24, 2020
Duration	1h 49m

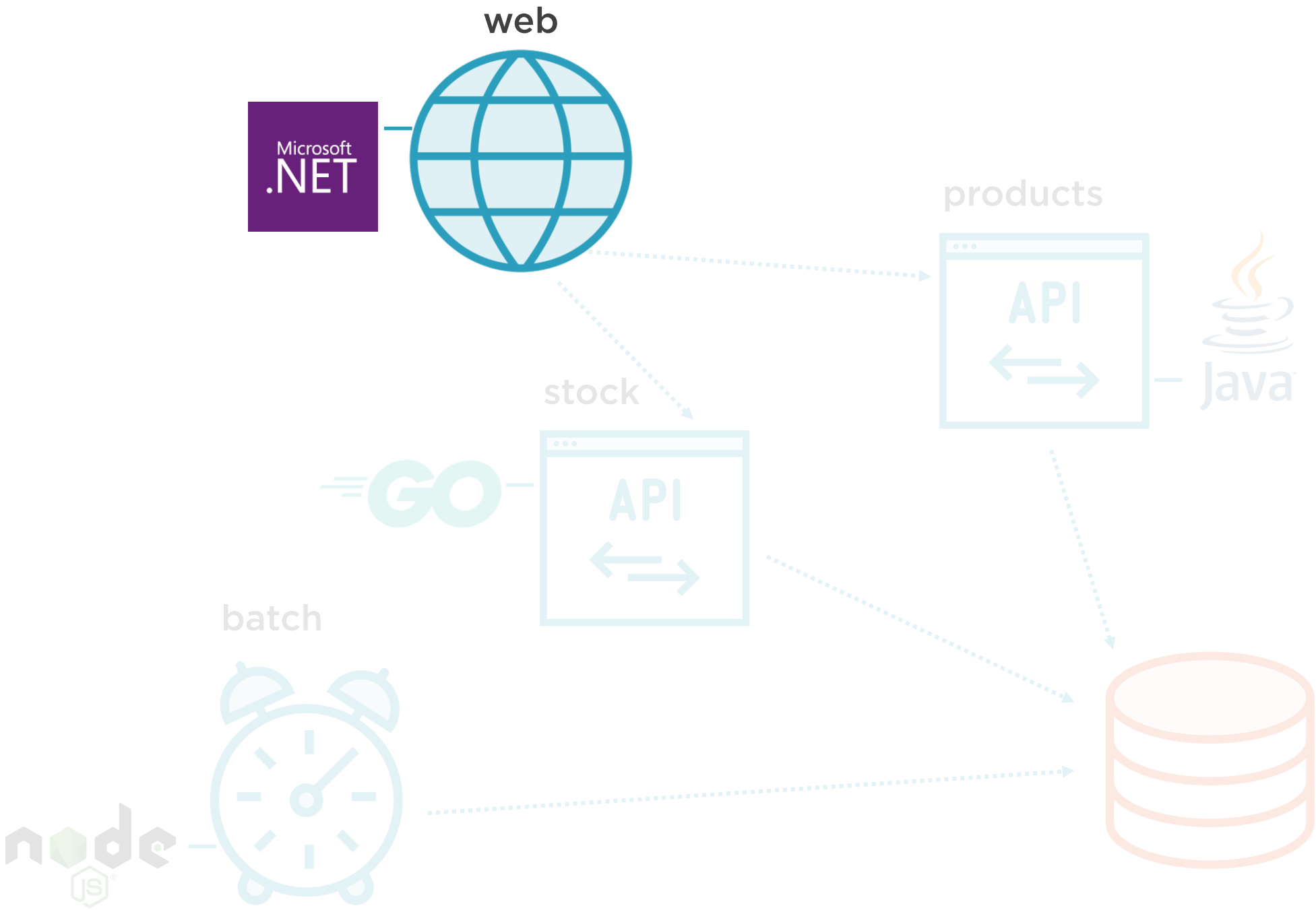
Description

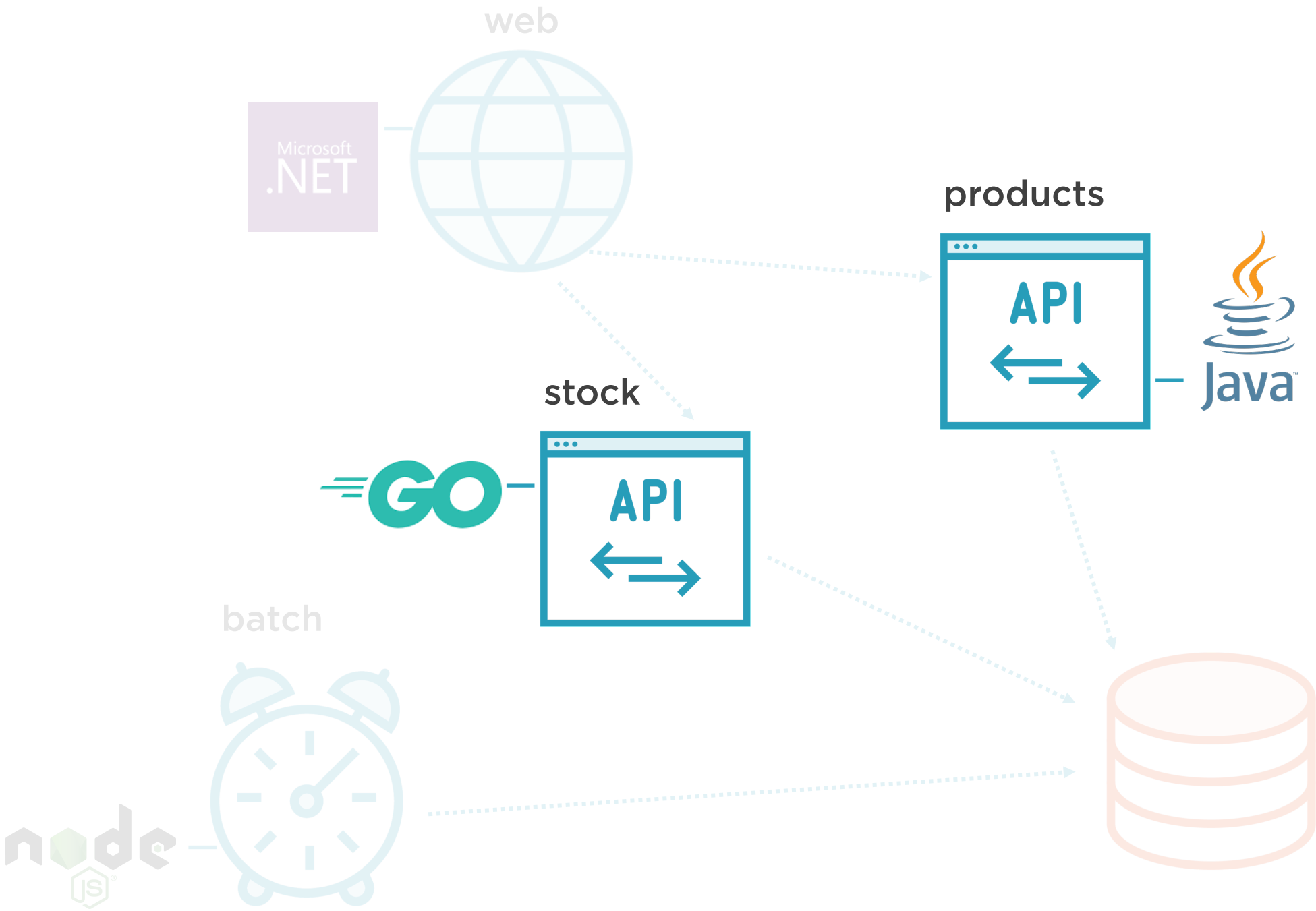
Prometheus is a cross-platform monitoring tool that lets you collect metrics from servers, containers, and applications and work with them all in the same way. In this course, Getting Started with Prometheus, you'll learn why it's such a popular approach to monitoring and how you can start bringing it into your organization. First, you'll learn about the architecture of Prometheus and how it uses a pull model to collect metrics from many targets. Then, you'll explore how to produce metrics from Linux and Windows servers using an exporter utility and from applications using a client library, and how to configure Prometheus to fetch those metrics. Finally, you'll discover the query language PromQL, how you can use it to track the changes in metrics over time, and visualize all the metrics in a dashboard. When you've finished with the course, you'll have the basic skills and knowledge of Prometheus needed to run a trial and evaluate it for your organization.

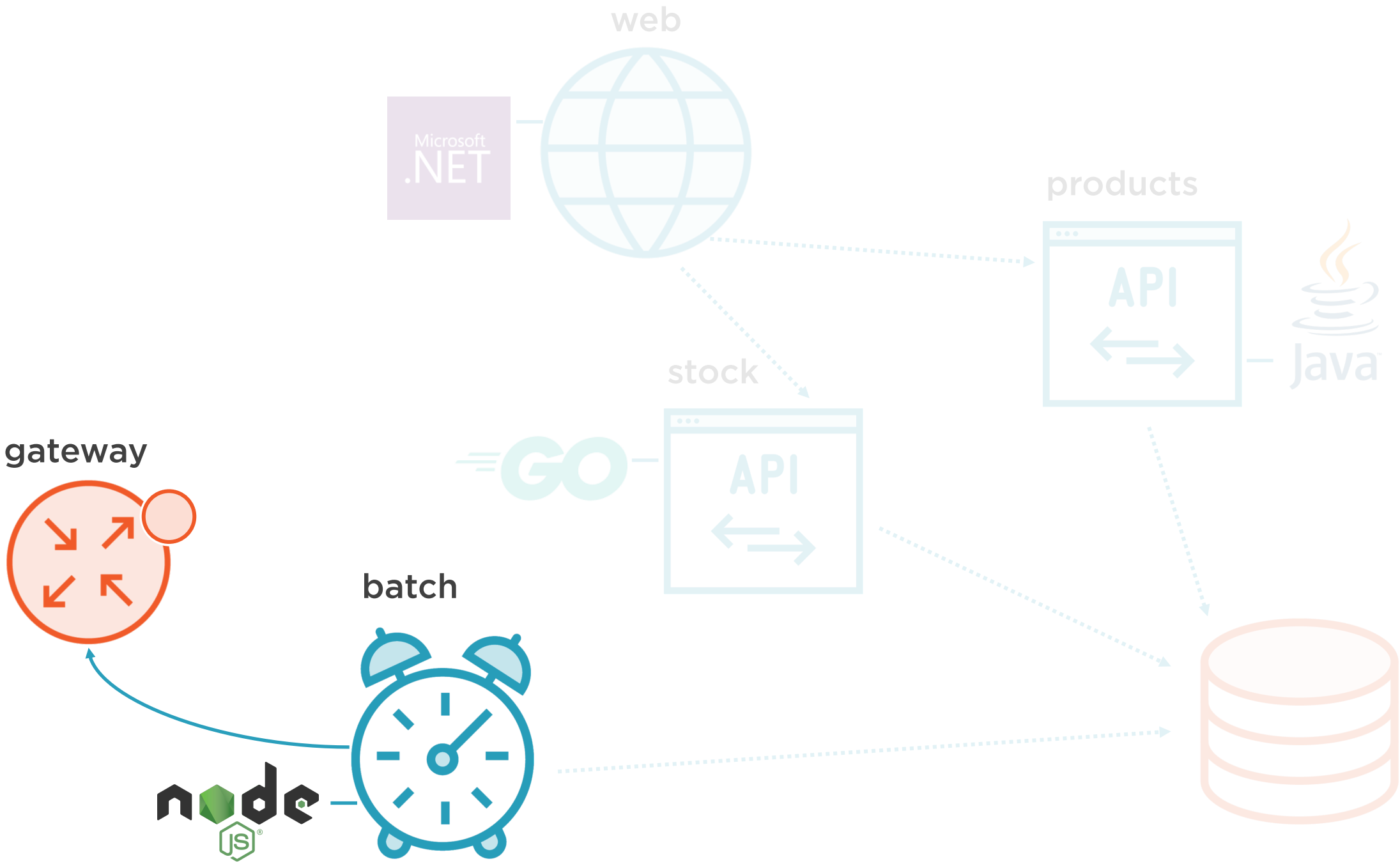
https://is.gd/yitezi

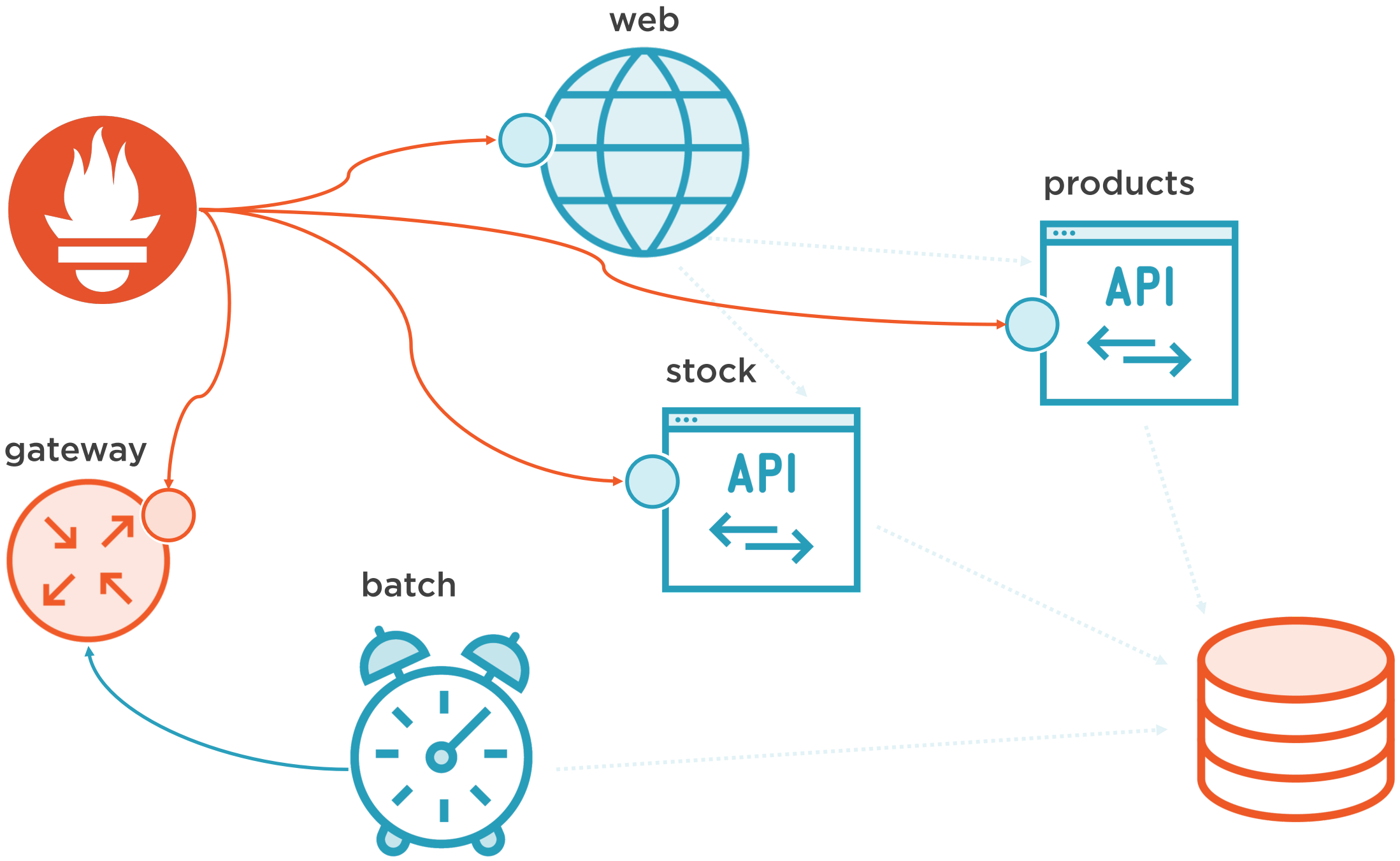

web
Microsoft
.NET
products
API
Java
stock
GO
API
batch
node
JS


web
products
API
stock
API
gateway
batch


web
Microsoft
.NET
products
API
Java
stock
GO
API
batch
node
JS


web
Microsoft
.NET
products
API
Java
stock
GO
API
batch
node
JS


web
Microsoft
.NET
products
API
Java
stock
GO
API
gateway
batch
node
JS


web
products
API
stock
API
gateway
batch

Demo

Adding metrics to a web app

- Using the .NET client library
- Wiring the metrics endpoint
- Verifying the metrics

Library

Package reference

Wiring

Plug into app runtime

Metrics

Record custom values

Reference the client library

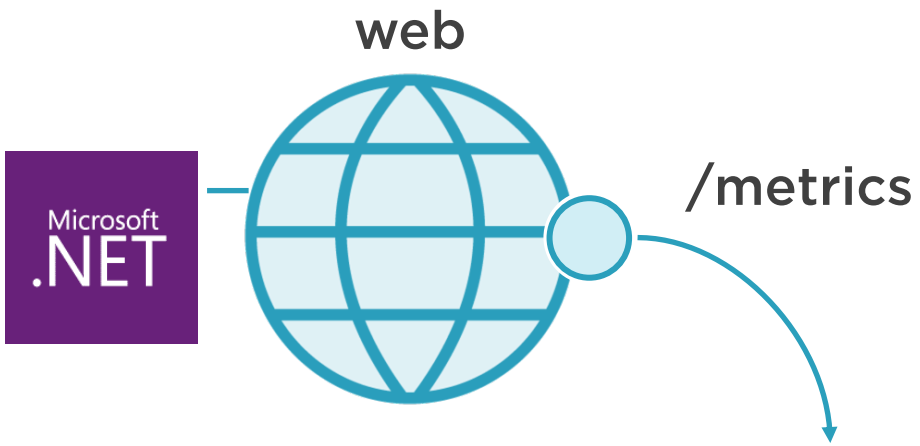
WiredBrain.Web.csproj

```
<PackageReference
    Include="prometheus-net.AspNetCore"
    Version="3.6.0"/>
```

Wire up the metrics

Startup.cs

```
// after app.UseRouting();
app.UseMetricServer();
app.UseHttpMetrics();
```

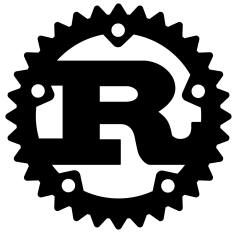



```
# HELP process_cpu_seconds_total Total user and system CPU time
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 1.9

# HELP dotnet_total_memory_bytes Total known allocated memory
# TYPE dotnet_total_memory_bytes gauge
dotnet_total_memory_bytes 11622608

# HELP http_request_duration_seconds The duration of HTTP requests
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{code="200",method="GET",le="0.064"} 3
```

Client libraries

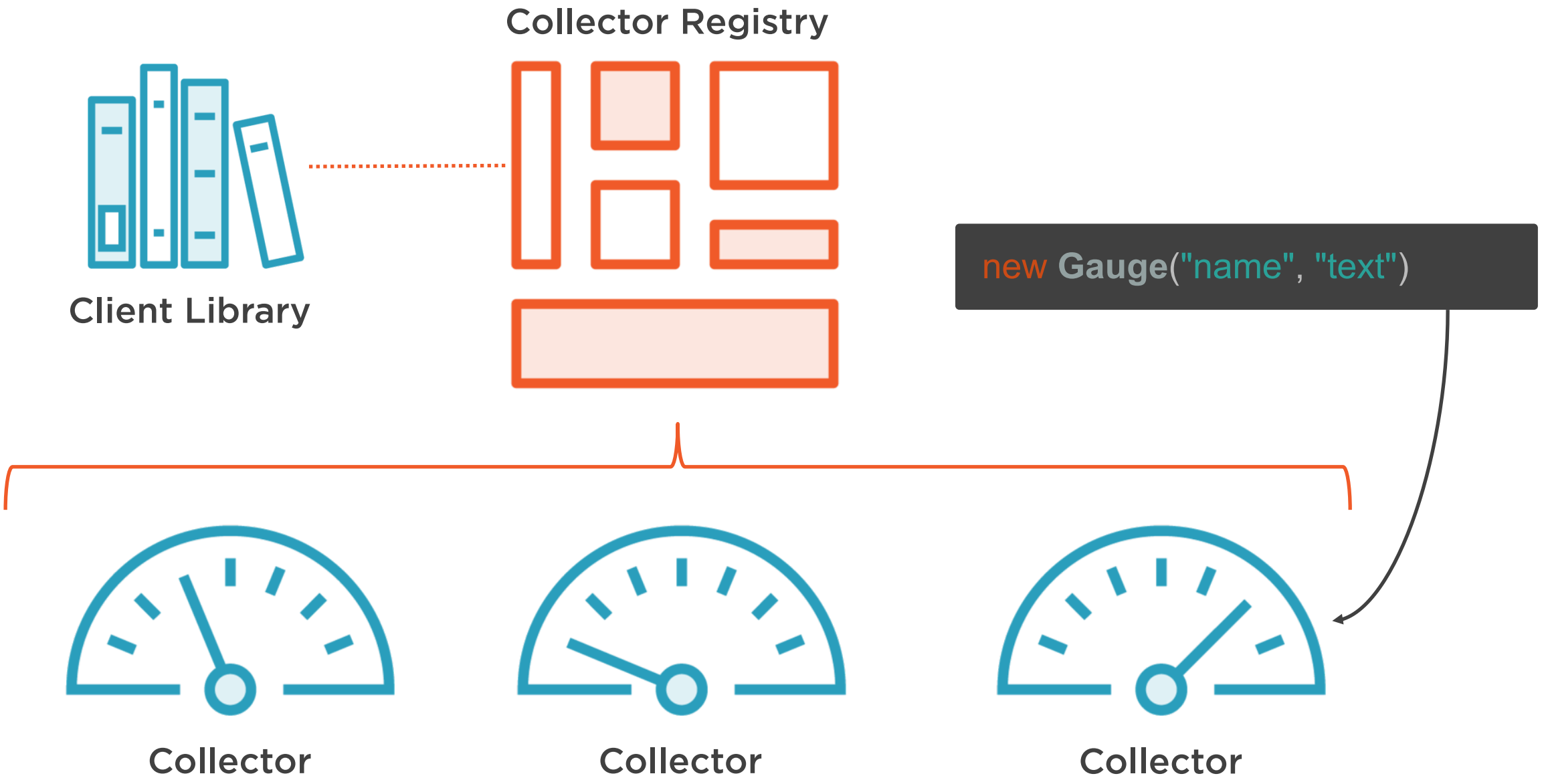
Official

Community



...

https://is.gd/awoded


Collector Registry
Client Library
new Gauge("name", "text")
Collector
Collector
Collector

Counter

Must be supported

Gauge

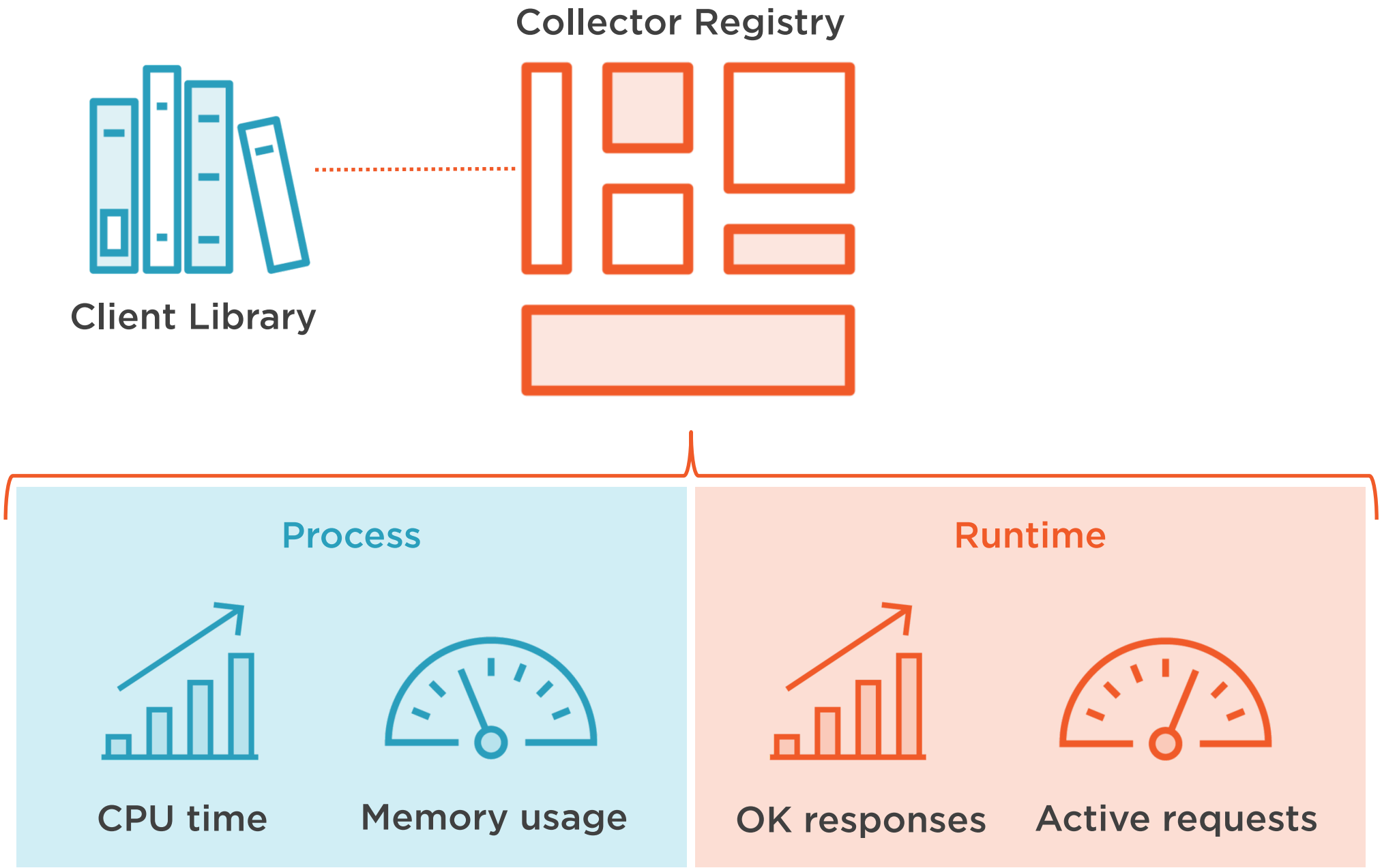
Must be supported

Summary

Should be supported

Histogram

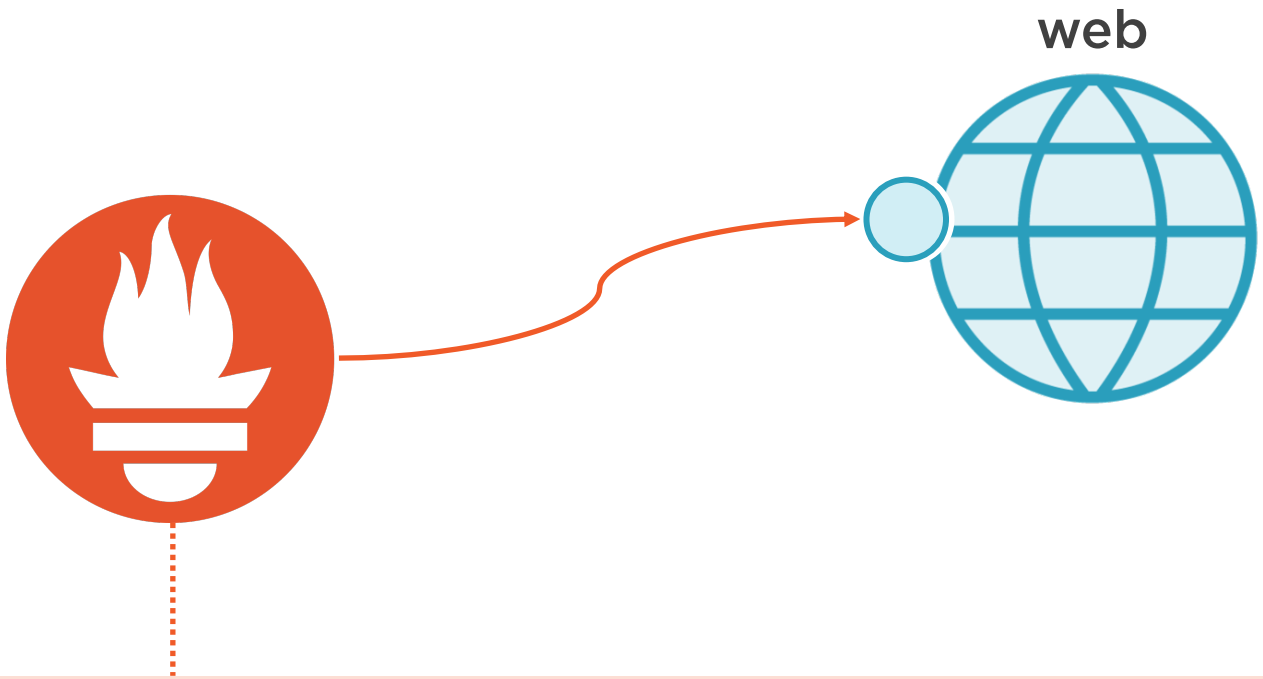
Should be supported


Collector Registry
Client Library
Process
CPU time
Memory usage
Runtime
OK responses
Active requests

Demo

Exploring client library metrics

- Scrape the app with Prometheus
- Build simple graphs
- Run a load test



http_requests_in_progress{method="GET",controller="Home",action="Index"} 15

dotnet_total_memory_bytes 11622608

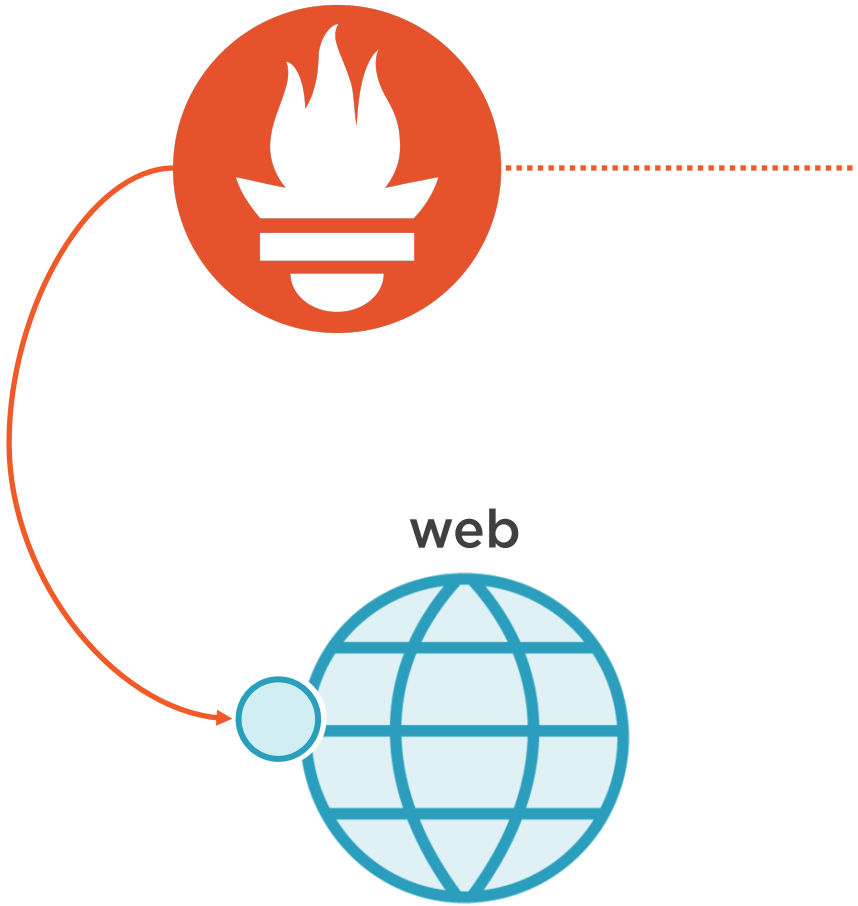
http_request_duration_seconds_bucket
{code="200",method="GET",controller="Home",action="Index",le="0.016"} 0

http_request_duration_seconds_bucket
{code="200",method="GET",controller="Home",action="Index",le="0.032"} 2

Wire up the metrics

Startup.cs

```
// the client library collects HTTP metrics
app.UseHttpMetrics();
```



process_cpu_seconds_total 1.9

process_start_time_seconds 1599471459.24

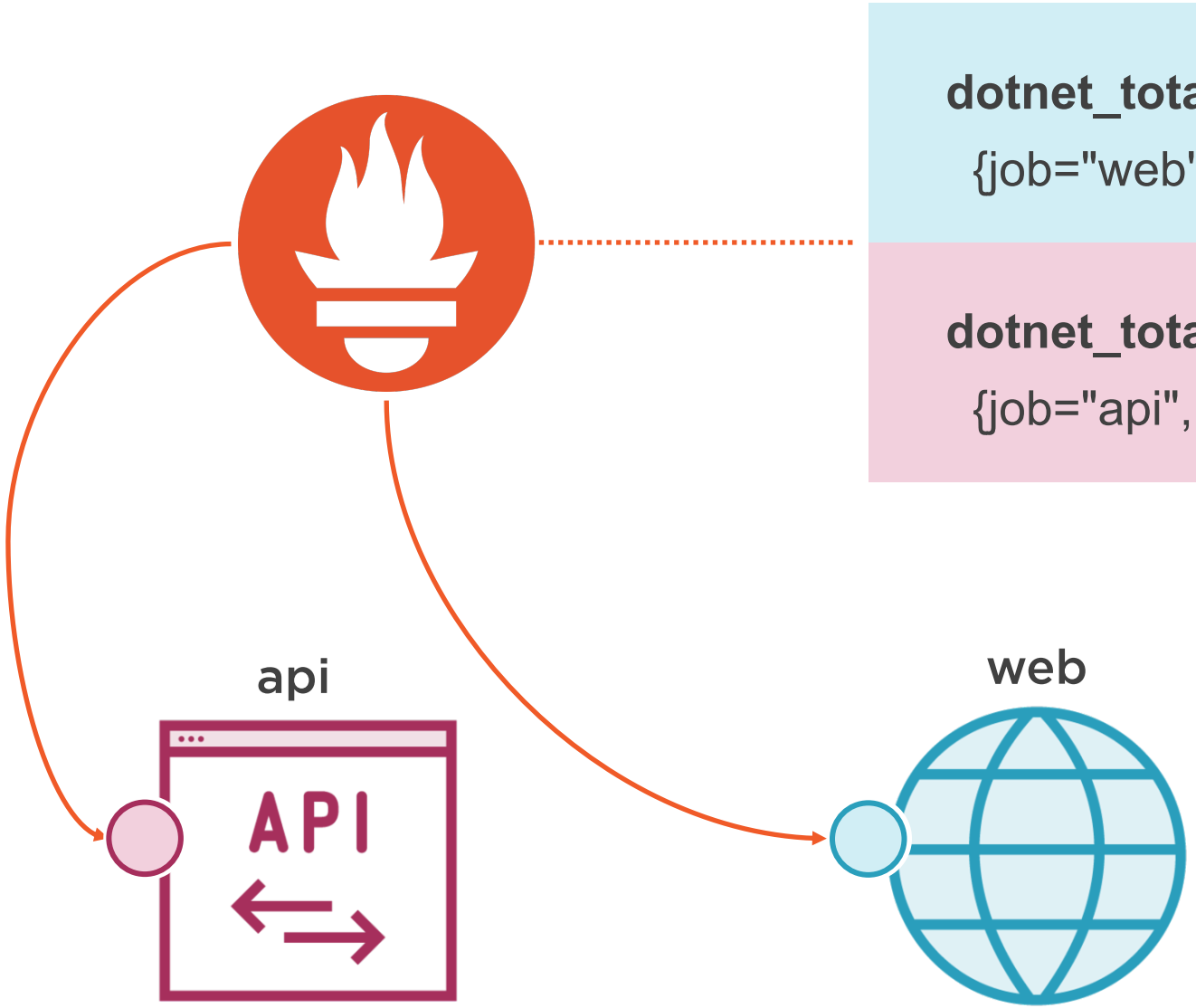
process_num_threads 26

process_open_handles 180

dotnet_collection_count_total{gen="0"} 6

dotnet_collection_count_total{gen="1"} 4

dotnet_collection_count_total{gen="2"} 1

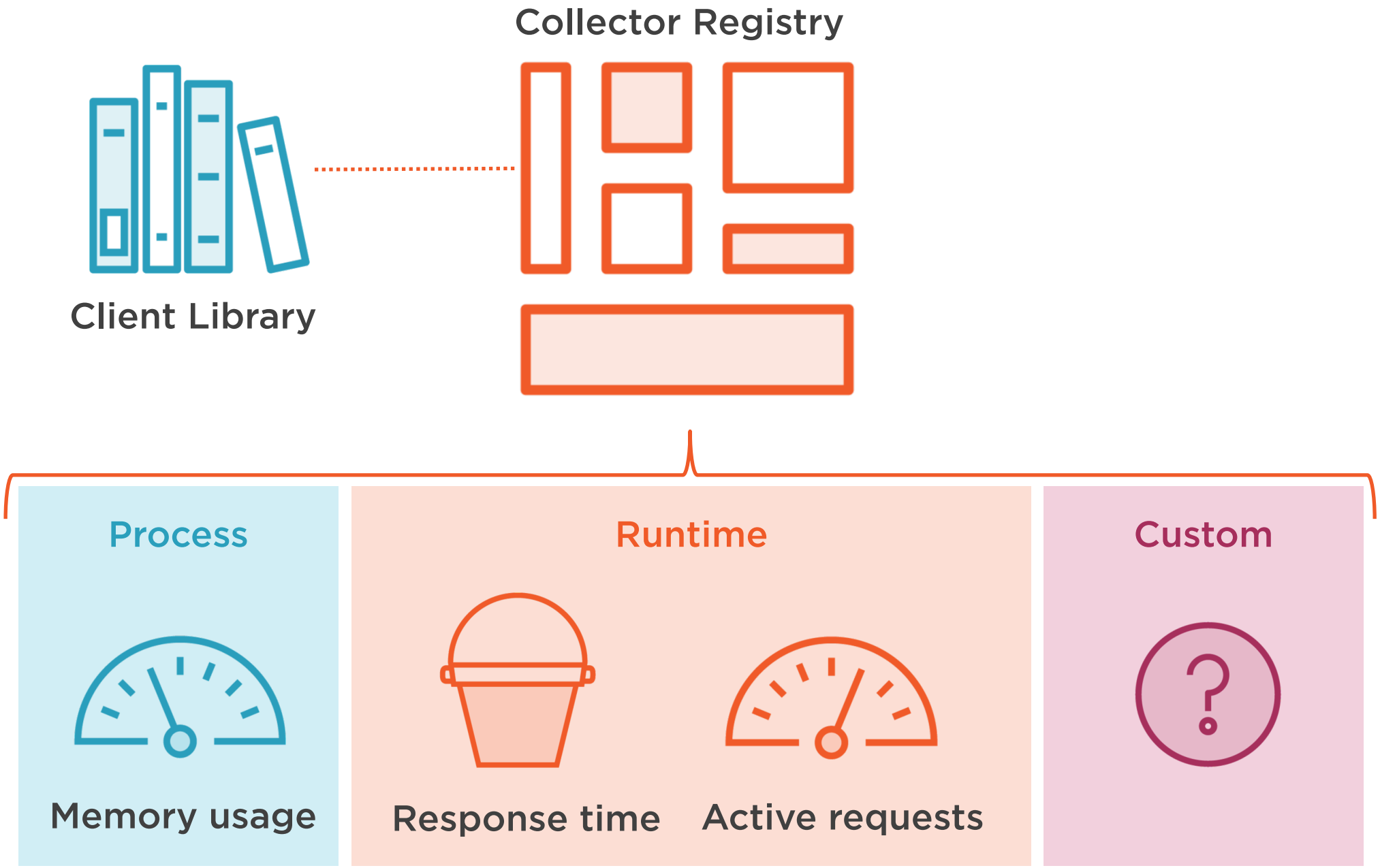


dotnet_total_memory_bytes

{job="web",instance="w1"} 11622608

dotnet_total_memory_bytes

{job="api",instance="a1"} 43102213


Collector Registry
Client Library
Process
Memory usage
Runtime
Response time
Active requests
Custom

Site Reliability Engineering (SRE): The Big Picture

★★★★★ By Elton Stoneman

Site Reliability Engineering (SRE) is how Google runs production systems, promoting high availability with high velocity and removing operational toil. It achieves the same goals as DevOps without the culture shift.

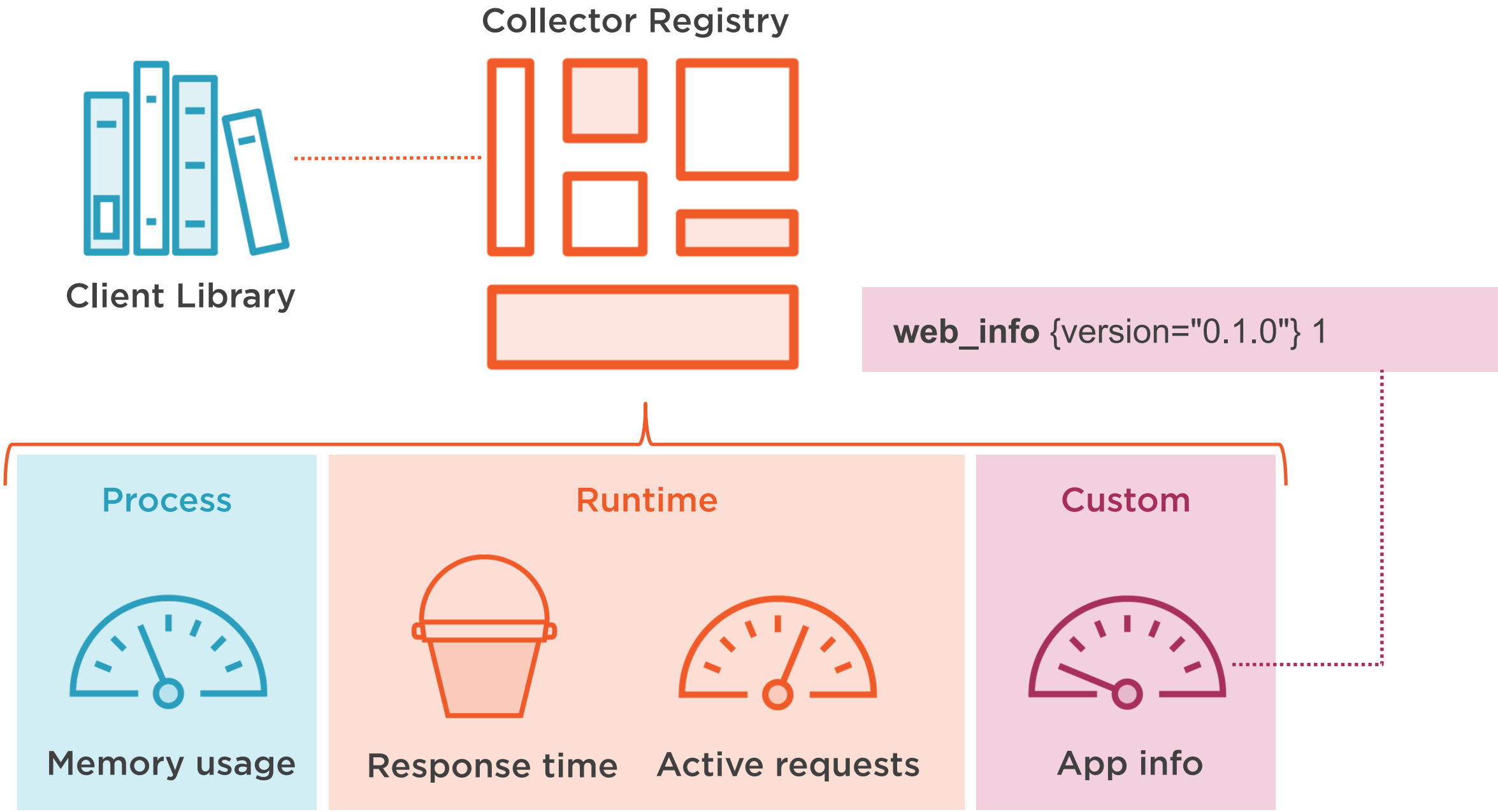
Course info

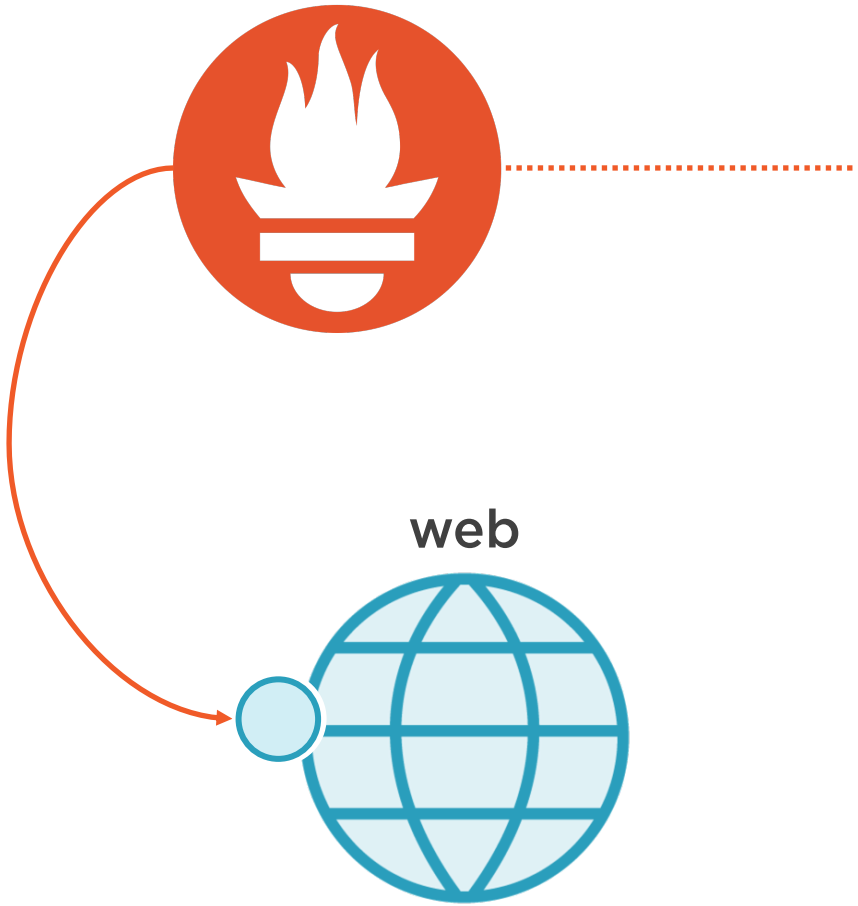
Rating	★★★★★ (29)
Level	Beginner
Updated	Mar 5, 2020
Duration	1h 41m

Description

Site Reliability Engineering (SRE) is a set of principles and practices that supports software delivery - keeping production systems stable and still delivering new features at speed. In this course, Site Reliability Engineering (SRE): The Big Picture, you'll get a thorough overview of how SRE works and why it's a good choice for many organisations. First, you'll learn the differences between SRE, DevOps, and traditional operations. Next, you'll discover how engineering practices help to reduce toil and provide more time to focus on high value tasks. Finally, you'll learn how SRE approaches monitoring and alerting, and about the SRE approach to managing incidents. When you're finished with this course, you'll be able to evaluate SRE and see if it's a good fit for your organisation.

https://is.gd/veroto


Collector Registry
Client Library
web_info {version="0.1.0"} 1
Process
Memory usage
Runtime
Response time
Active requests
Custom
App info



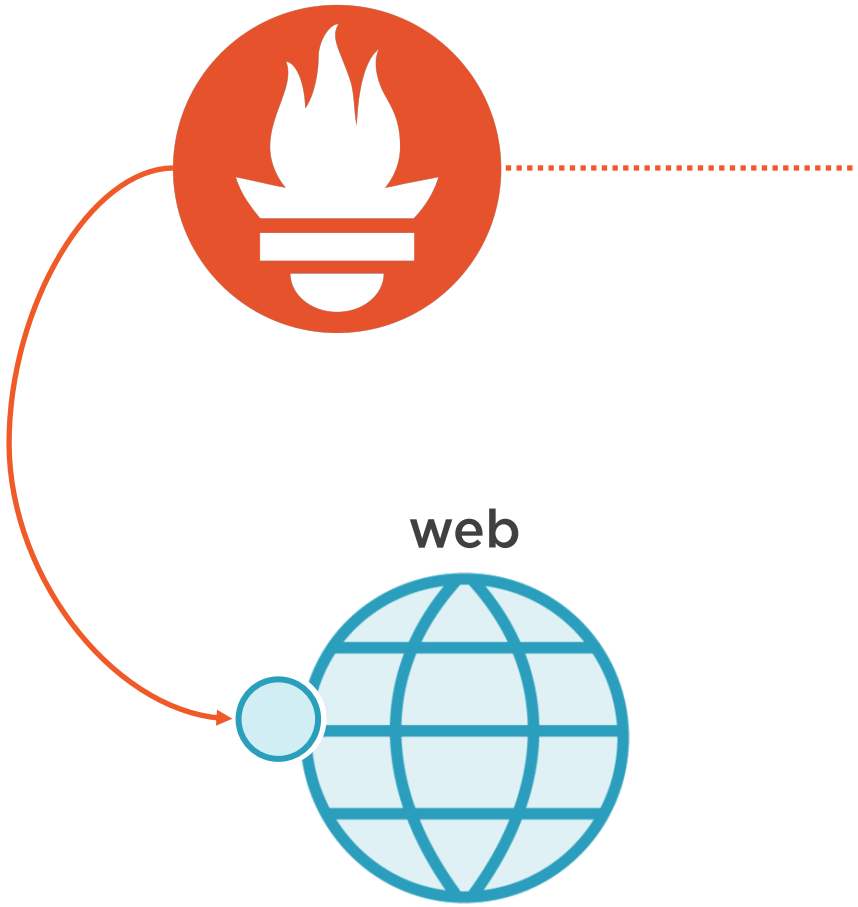
dotnet_total_memory_bytes
{version="0.1.0"} 28413694

dotnet_total_memory_bytes
{version="0.2.0"} 10028317

Demo

Recording application information

- Adding a custom metric
- Using labels for version info
- Joining the info metric in queries



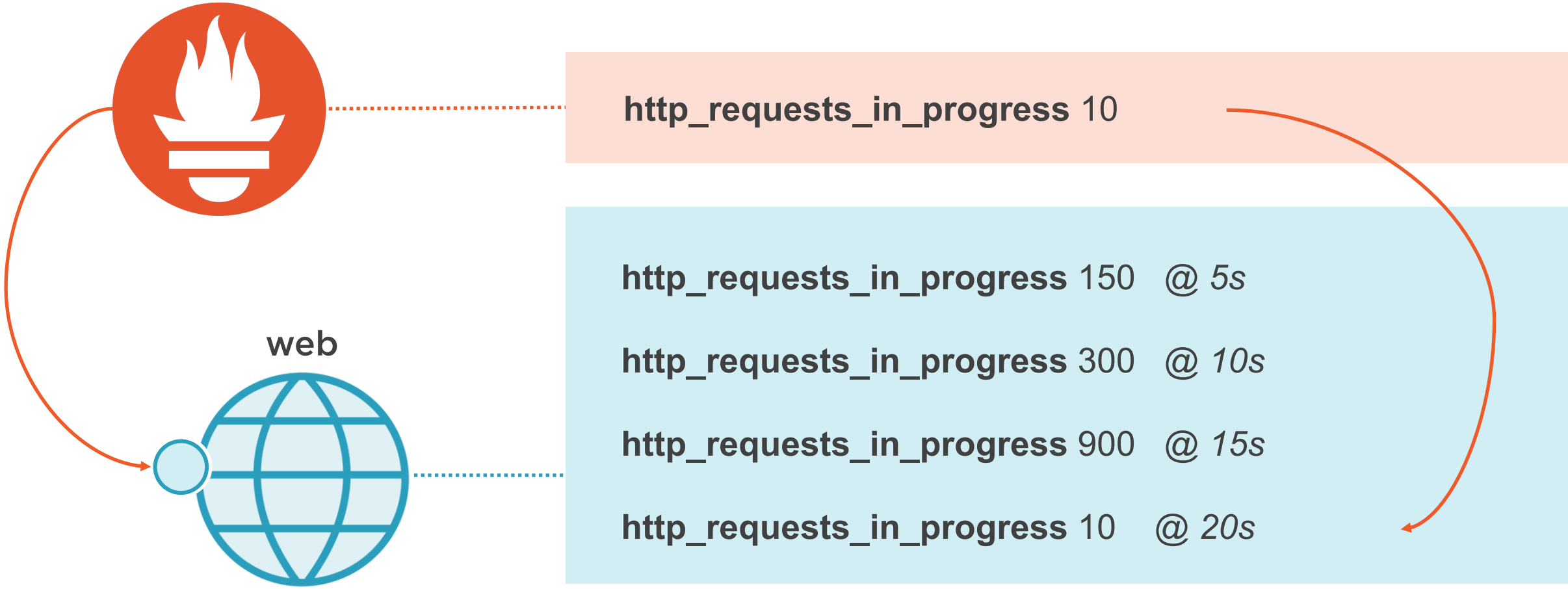
web_info

```
{dotnet_version="3.1.7",
 assembly_name="WiredBrain.Web",
 app_version="0.1.0"} 1
```

Recording info metrics

Program.cs

```
private static readonly Gauge _InfoGauge =
    Metrics.CreateGauge("web_info", "Web app info",
    "dotnet_version", "assembly_name", "app_version");
//...
_InfoGauge.Labels("3.1.7", "WiredBrain.Web", "0.1.0").Set(1);
```


http_requests_in_progress 10
web
http_requests_in_progress 150 @ 5s
http_requests_in_progress 300 @ 10s
http_requests_in_progress 900 @ 15s
http_requests_in_progress 10 @ 20s

http_requests_in_progress
{app_version="0.1.0"} 10

http_requests_in_progress 10

web_info
{dotnet_version="3.1.7",
assembly_name="WiredBrain.Web",
app_version="0.1.0"} 1

Summary

Instrumenting with client libraries

- Package reference
- Endpoint publishing
- Default metrics collection

Custom metrics

- Standard Prometheus types
- Labels

Info metrics

- Record application versions
- Surface in PromQL results

Up Next:
Recording custom application metrics