Recording Custom Application Metrics

Elton Stoneman
CONSULTANT & TRAINER

@EltonStoneman | blog.sixeyed.com

Library

Package reference

Wiring

Plug into app runtime

Metrics

Record custom values

Manual

Set values in code

Manually setting metrics

example.code

```
// create a counter:
counter = new Counter("name", "help-text", "labels");


// increment the value:
counter.labels("label").inc();
```

Manual

Set values in code

Middleware

Set during pipeline

AOP

Set in code injection

Wiring up middleware

router.go

```
// middleware collects HTTP response times:
router.Use(middleware.Prometheus)


// metrics endpoint includes middleware metrics:
router.Path("/metrics").Handler(promhttp.Handler())
```

Using AOP

ProductsController.java

```
@RequestMapping("/products")
@Timed()
public List<Product> get() { // ... }
```

Manual

Set values in code

Fine-grained control

Middleware

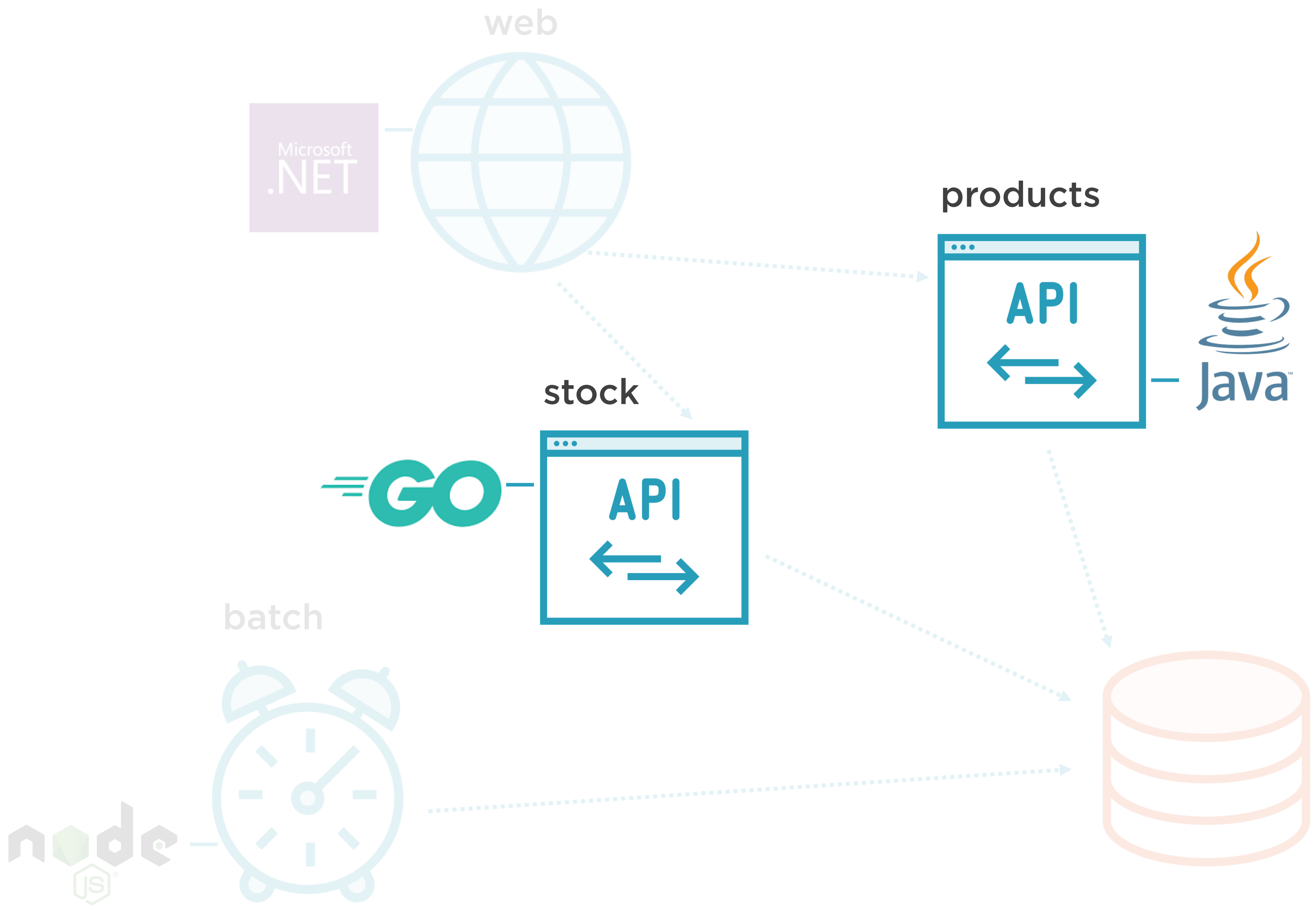
Set during pipeline

Common metrics

AOP

Set in code injection

Opt-in for key features


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

Demo

Adding metrics to the stock API

- Using the Go client library
- Wiring the metrics endpoint
- Recording the info metric

Demo

Collecting metrics with middleware

- Recording active users
- Wiring up middleware
- Recording HTTP response times

Referencing the client library

go.mod

```
module stock-api

go 1.14

require (
    github.com/prometheus/client_golang v1.7.1
)
```

Wiring up the metrics endpoint

router.go

```go
// Gorilla MUX supports the standard HTTP handler:
router := mux.NewRouter()
router.Path("/metrics").Handler(promhttp.Handler())
```

Setting the info metric

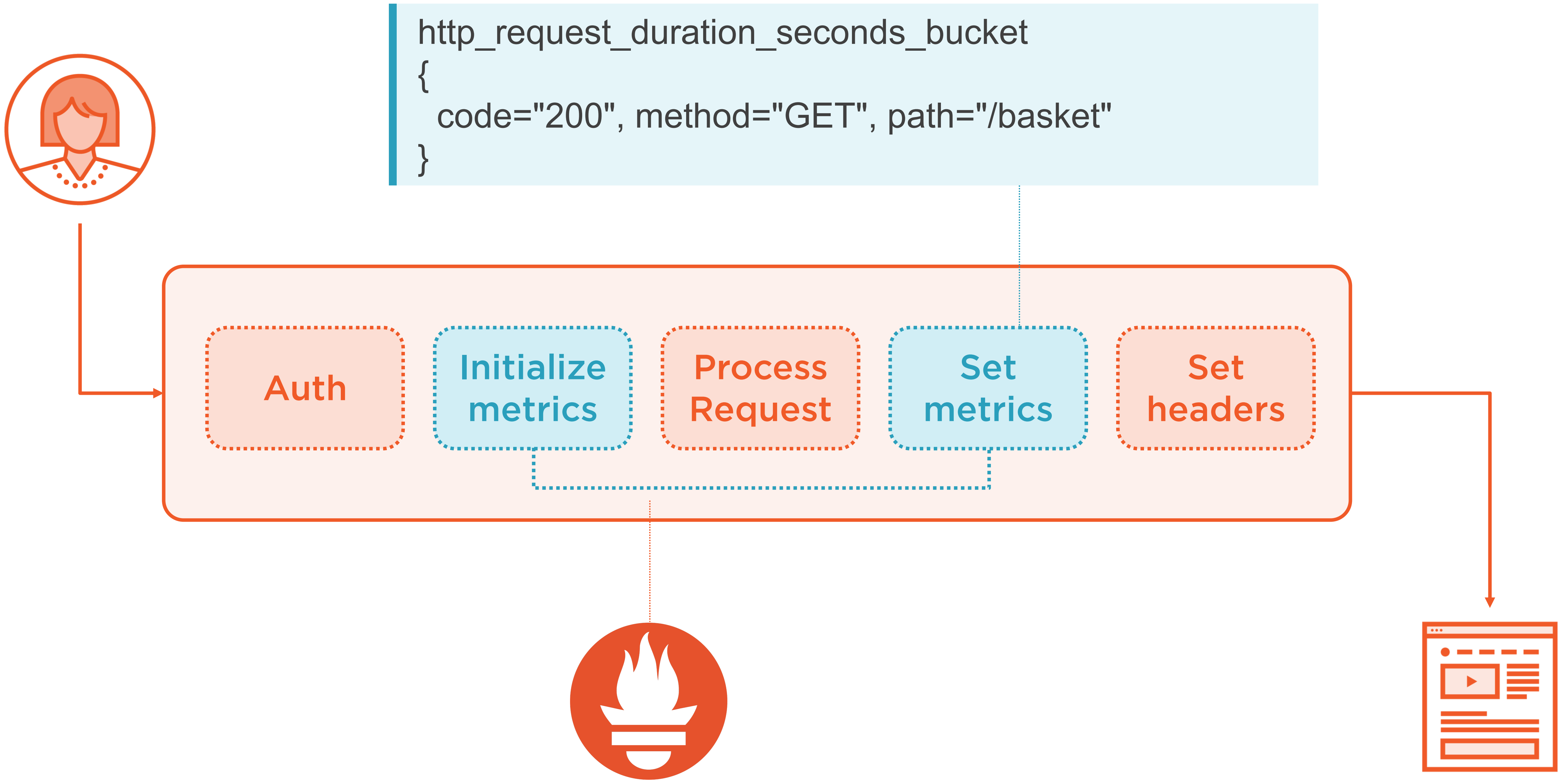
main.go

```go
var (
    appInfo = promauto.NewGaugeVec(prometheus.GaugeOpts{
      Name: "app_info", Help: "Application info",
      }, []string{"version", "goversion"})
)
// ...
appInfo.WithLabelValues("0.2.0", "1.14.4").Set(1)
```

Collecting metrics in middleware

prometheus.go

```go
activeRequests.Inc()
// ...
timer := prometheus.NewTimer(httpDuration.WithLabelValues(path))
next.ServeHTTP(w, r)
timer.ObserveDuration()
activeRequests.Dec()
```



```
http_request_duration_seconds_bucket
{
  code="200", method="GET", path="/basket"
}
```

Cardinality

http_request_durations_bucket

- le: 0.1..10
- method: GET,POST
- code: 200,400,500

$\Bigg\}$ 60x time series

Cardinality

http_request_durations_bucket

- le: 0.1..10
- method: GET,POST
- code: 200,400,500,401,403,503
- url: /,/a/1,/a/2,/b/1...

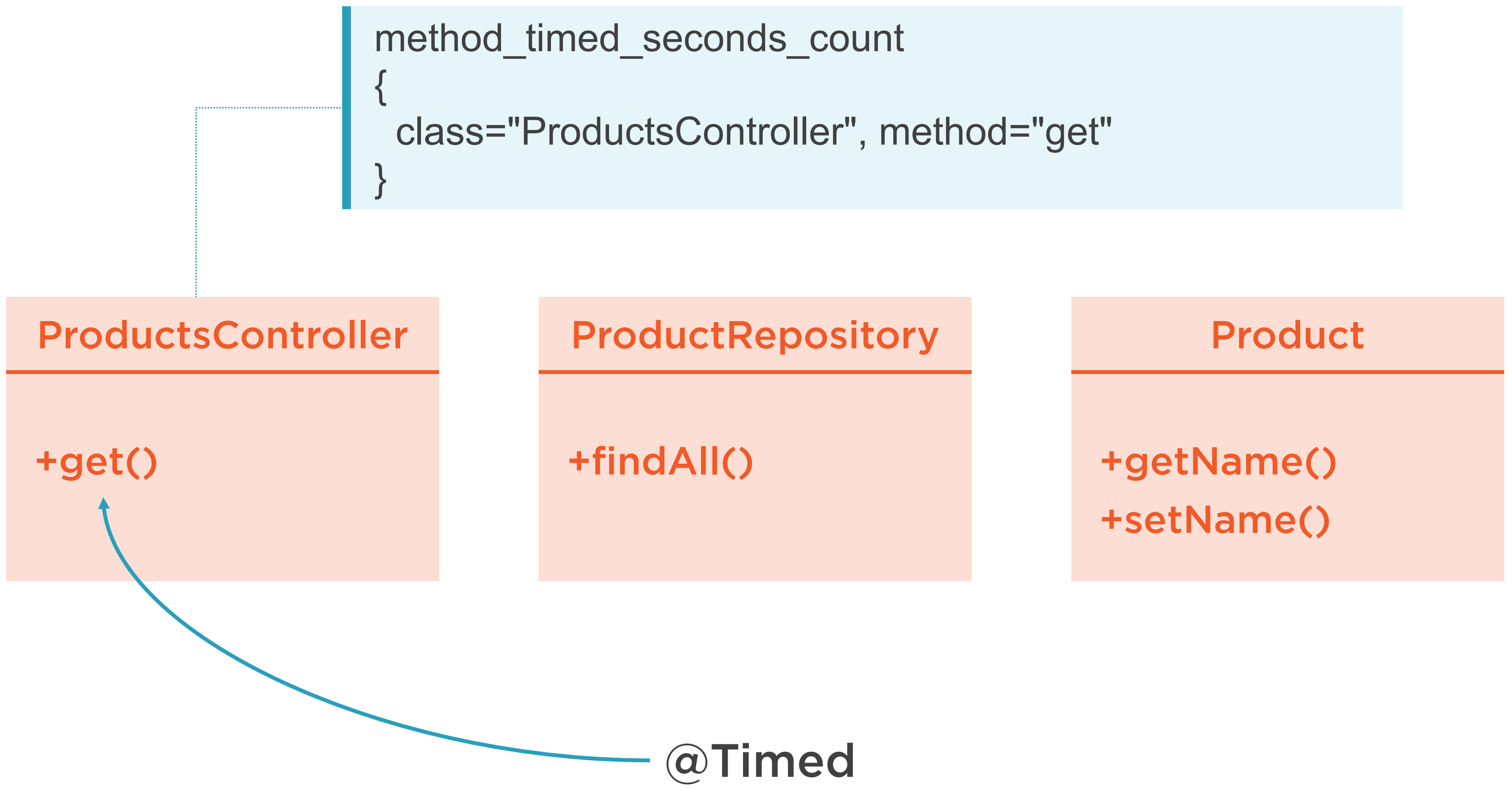
? time series

Cardinality

http_request_durations_bucket

- le: 0.1..10
- method: GET,POST
- code: 2xx,4xx,5xx
- path: /a,/b...

<100 time series


```
method_timed_seconds_count
{
  class="ProductsController", method="get"
}
```

Demo

Adding metrics to the products API

- Using the Java Micrometer library
- Wiring the metrics endpoint
- Recording the info metric

Demo

Collecting metrics with AOP

- Manually recording metrics
- Adding the AOP handler
- Recording method durations

Referencing the client library

pom.xml

```
<dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Wiring up the metrics endpoint

application.properties

```
management.endpoints.web.exposure.include=prometheus
```

Setting the info metric

ApplicationStartup.java

```java
private AtomicInteger appInfoGaugeValue = new AtomicInteger(1);

@Autowired
MeterRegistry registry;

@Override
public void run(ApplicationArguments args) throws Exception {
    registry.gauge("app.info", Tags.of("version", "0.2.0"), appInfoGaugeValue);
}
```

Collecting metrics manually

ProductsController.java

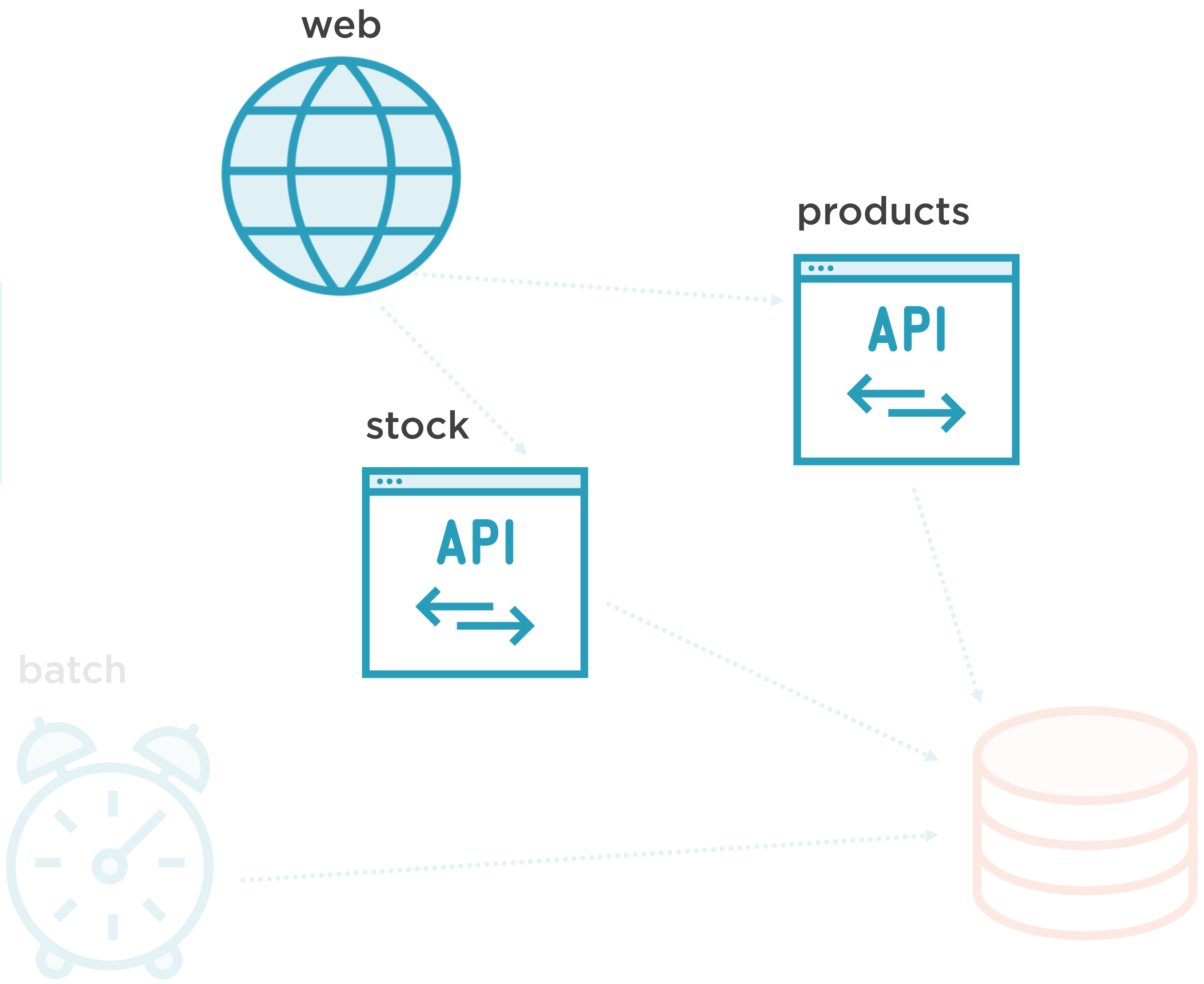
```
registry.counter("products_data_load_total",
                                    "status", "called").increment();
try { //... }
catch (Exception ex) {
  registry.counter("products_data_load_total",
                                          "status", "failure").increment();
}
```

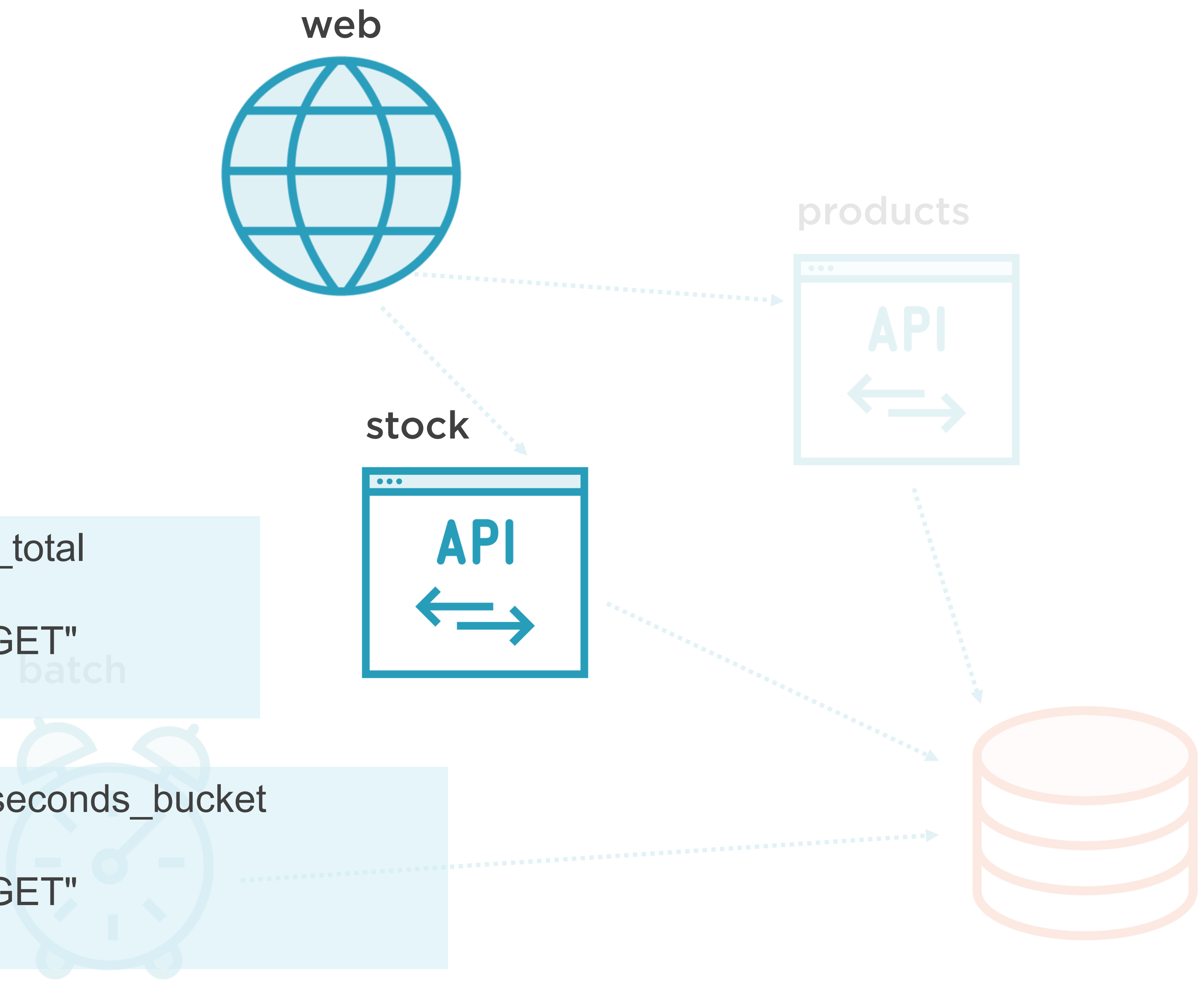
Collecting metrics with AOP

ProductsController.java

```
@RequestMapping("/products")
@Timed()
public List<Product> get() {
  //...
}
```

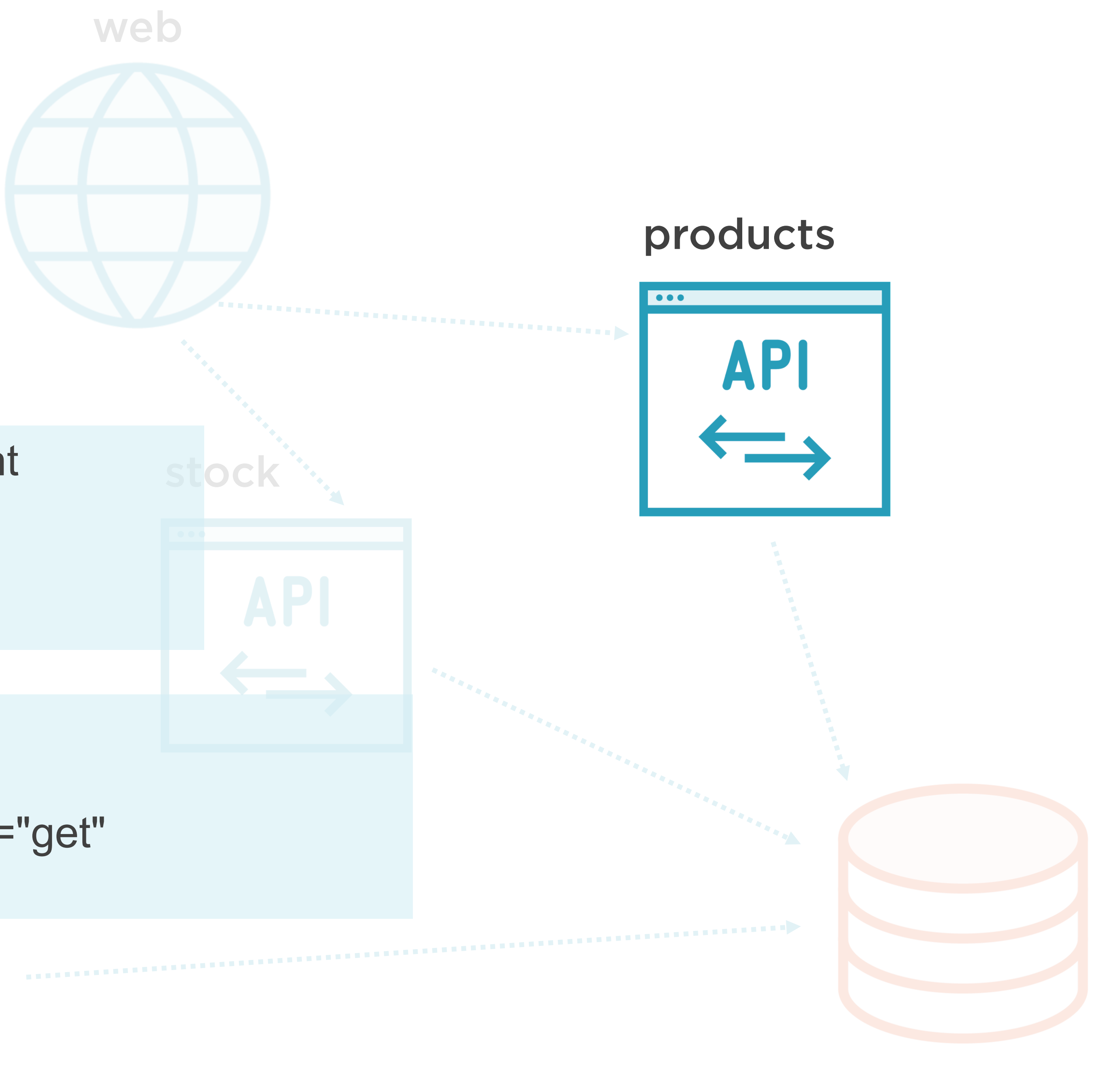
```
app_info
{
  version="0.2.0"
}
```





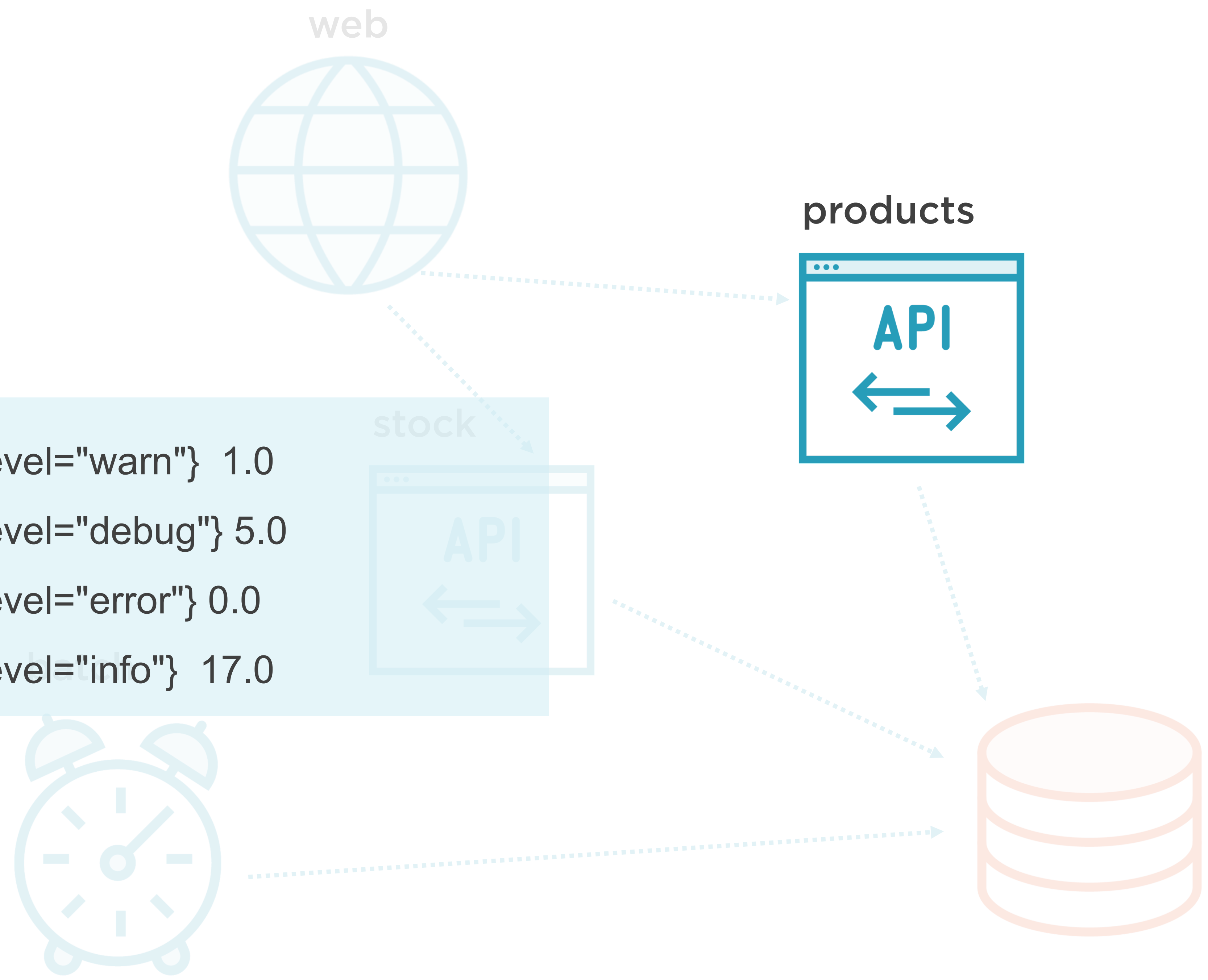
```
http_requests_received_total
{
  code="200",method="GET"
}
```

```
http_request_duration_seconds_bucket
{
  code="200",method="GET"
}
```



```
http_server_requests_seconds_count
{
  method="GET",status="200"
}
```

```
method_timed_seconds_count
{
  class="ProductsController",method="get"
}
```



```
logback_events_total{level="warn"}  1.0
logback_events_total{level="debug"} 5.0
logback_events_total{level="error"} 0.0
logback_events_total{level="info"}  17.0
```

Summary

Adding custom metrics

- Manually setting values
- Using middleware
- Aspect-Oriented Programming

Go and Java client libraries

- Standard integration pattern
- Implementation differences

Metrics best practices

- Use consistent names
- Capture low-level data
- Limit cardinality of labels

Up Next:
Pushing metrics from batch jobs