

# **Spring Actuator.**

## **Настройка мониторинга с Prometheus и Grafana**









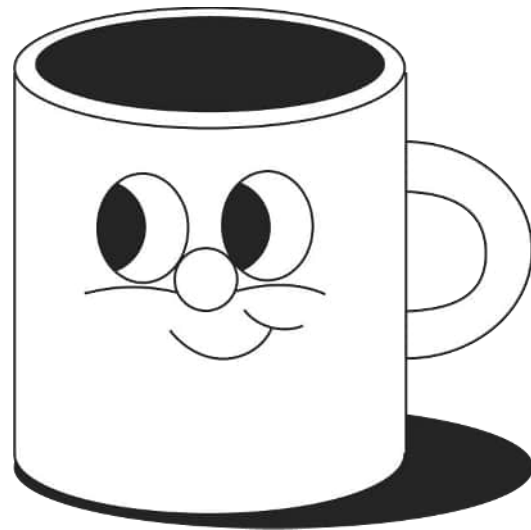
## Евгений Манько

Java-разработчик, создатель данного курса

- 🌟 Разрабатывал бэкенд для Яндекс, Тинькофф, МТС;
- 🌟 Победитель грантового конкурса от «Росмолодежь»;
- 🌟 Руководил IT-Департаментом «Студенты Москвы».

## Что будет на уроке сегодня

-  Spring Actuator
-  Эндпоинт Health, Metrics, Info, Loggers
-  JVM метрики в Spring Actuator
-  HTTP метрики
-  Метрики баз данных
-  Практика





# Spring Actuator

Spring Actuator — это подпроект Spring, который предоставляет готовые production-ready функции: метрики, мониторинг, даже некоторую информацию для отладки.





# Spring Actuator

Spring Actuator — это подпроект Spring, который предоставляет готовые production-ready функции: метрики, мониторинг, даже некоторую информацию для отладки.





# Spring Actuator

Чтобы начать, добавьте зависимость в ваш pom.xml  
или build.gradle:

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```





## Эндпоинт Health

Когда вы отправляете GET-запрос на /actuator/health, вы получаете ответ в формате JSON, который может выглядеть примерно так:

```
1 {  
2   "status": "UP",  
3   "components": {  
4     "db": {  
5       "status": "UP",  
6       "details": {  
7         // детали о состоянии базы данных  
8       }  
9     },  
10    "diskSpace": {  
11      "status": "UP",  
12      "details": {  
13        // детали о доступном дисковом пространстве  
14      }  
15    }  
16    // и так далее  
17  }  
18 }
```



## Кастомизация эндпоинта Health

```
1 @Component
2 public class MyHealthIndicator implements HealthIndicator {
3
4     @Override
5     public Health health() {
6         // ваша логика проверки здесь
7         if (someCheck()) {
8             return Health.up().build();
9         }
10        return Health.down().withDetail("reason", "Тут причина проблемы").build();
11    }
12
13    public boolean someCheck() {
14        // реализация проверки
15        return true;
16    }
17 }
```







## Эндпоинт Metrics. Как это работает?

```
1 {  
2   "name": "jvm.memory.used",  
3   "description": "The amount of used memory",  
4   "baseUnit": "bytes",  
5   "measurements": [  
6     {  
7       "statistic": "VALUE",  
8       "value": 550295  
9     }  
10  ]  
11 }
```



## Эндпоинт Metrics. Кастомизация метрик

```
1 @Autowired
2 private MeterRegistry meterRegistry;
3
4 public void someMethod() {
5     // Ваш код
6
7     // Увеличиваем счетчик на единицу
8     meterRegistry.counter("my.custom.counter").increment();
9 }
```



## Эндпоинт Info

```
7 {  
8   "app": {  
9     "name": "My Cool App",  
10    "version": "1.0.0",  
11    "description": "This app does something awesome!"  
12  }  
13 }
```



## Эндпоинт Loggers. Как этим управлять?

```
1 {  
2   "levels": ["OFF", "ERROR", "WARN", "INFO", "DEBUG", "TRACE"],  
3   "loggers": {  
4     "ROOT": {  
5       "configuredLevel": "INFO"  
6     },  
7     "com.example": {  
8       "configuredLevel": "DEBUG"  
9     },  
10    // ...  
11  }  
12 }
```



## JVM метрики в Spring Actuator

Как построить метрики?

```
1 {  
2     "name": "jvm.memory.used",  
3     "description": "The amount of used memory",  
4     // ... more metadata and values  
5 }
```



## HTTP метрики

Как это работает в Spring Actuator?

```
1 {  
2   "name": "http.server.requests",  
3   "description": "HTTP Server Requests",  
4   // ... more data and values  
5 }
```

# Метрики баз данных

Как это устроено в Spring Actuator?

```
1 {  
2   "name": "hikaricp.connections.active",  
3   "description": "Active DB connections",  
4   // ... more data and values  
5 }
```





# Логирование

## Как это сделать?

```
1 curl -i -X POST -H 'Content-Type: application/json' -d '{"configuredLevel": "DEBUG"}'  
http://localhost:8080/actuator/loggers/com.example.YourClass|
```

## Почему это круто?

Изменение уровня логирования на лету — это инструмент для быстрой отладки и оптимизации вашего кода.





## Хочу свой эндпоинт!

```
1 import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
2 import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
3
4 @Endpoint(id = "myCustomEndpoint")
5 public class MyCustomEndpoint {
6
7     @ReadOperation
8     public CustomResponse customMethod() {
9         return new CustomResponse("Everything is awesome!", 42);
10    }
11
12    public static class CustomResponse {
13        private String message;
14        private int number;
15
16        // getters and setters
17    }
18 }
```



## Практика

### Шаг 1. Добавляем зависимости

Открываем наш pom.xml и туда вставляем:

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>|
```

### Шаг 2. Включаем эндпоинты

Переходим в application.properties:

```
1 management.endpoints.web.exposure.include=*
```



## Практика

### Шаг 5. Информацию о приложении

Эндпоинт `/actuator/info` позволит вам добавить любую кастомную информацию о вашем проекте.

В `application.properties` просто добавьте что-то вроде:

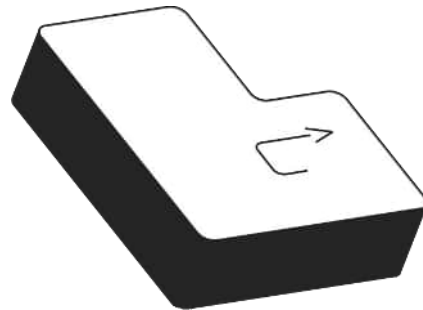
```
1 info.app.name=My Awesome App  
2 info.app.version=1.0.0|
```

И тадам! Вы видите это все на `/actuator/info`.

## Системы мониторинга: Prometheus и Grafana

В application.properties добавляем

```
1 management.metrics.export.prometheus.enabled=true
```





## Micrometer: Как измерять всё и сразу

Добавьте зависимость:

```
1 <dependency>
2   <groupId>io.micrometer</groupId>
3   <artifactId>micrometer-core</artifactId>
4 </dependency>
```

И в коде:

```
1 import io.micrometer.core.instrument.MeterRegistry;
2
3 @RestController
4 public class MyController {
5     private final MeterRegistry meterRegistry;
6
7     public MyController(MeterRegistry meterRegistry) {
8         this.meterRegistry = meterRegistry;
9     }
10
11     @GetMapping("/hello")
12     public String sayHello() {
13         meterRegistry.counter("requests_to_hello").increment();
14         // ваша логика
15         return "Hello, World!";
16     }
17 }
```

# Создание собственных метрик с Micromete

Счетчики и таймеры

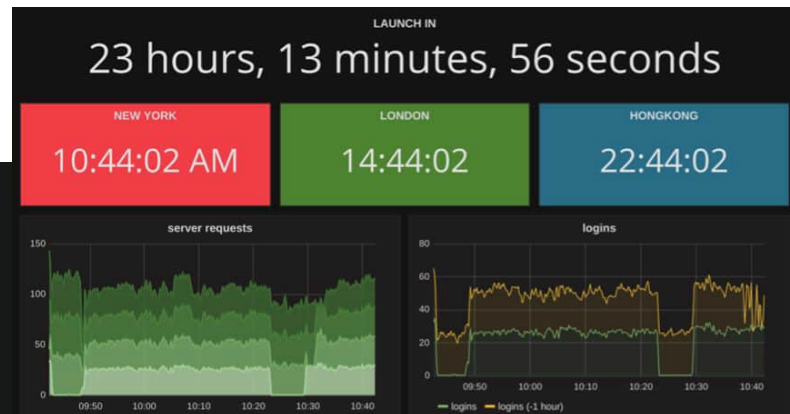
```
1 import io.micrometer.core.instrument.Counter;
2 import io.micrometer.core.instrument.MeterRegistry;
3
4 @Service
5 public class MyService {
6     private final Counter myCounter;
7
8     public MyService(MeterRegistry meterRegistry) {
9         myCounter = Counter.builder("my_custom_counter")
10             .description("Counts something very important")
11             .register(meterRegistry);
12     }
13
14     public void doSomethingImportant() {
15         // Тут какая-то важная логика
16         myCounter.increment();
17     }
18 }
```



## Создание собственных метрик с Micrometer

Таймеры:

```
1 import io.micrometer.core.instrument.Timer;
2
3 // ...
4
5 private final Timer myTimer;
6
7 public MyService(MeterRegistry meterRegistry) {
8     myTimer = Timer.builder("my_custom_timer")
9         .description("Timing something very important")
10        .register(meterRegistry);
11 }
12
13 public void doSomethingTimed() {
14     myTimer.record(() -> {
15         // Тут какая-то важная логика, время выполнения которой мы хотим замерить
16     });
17 }
```



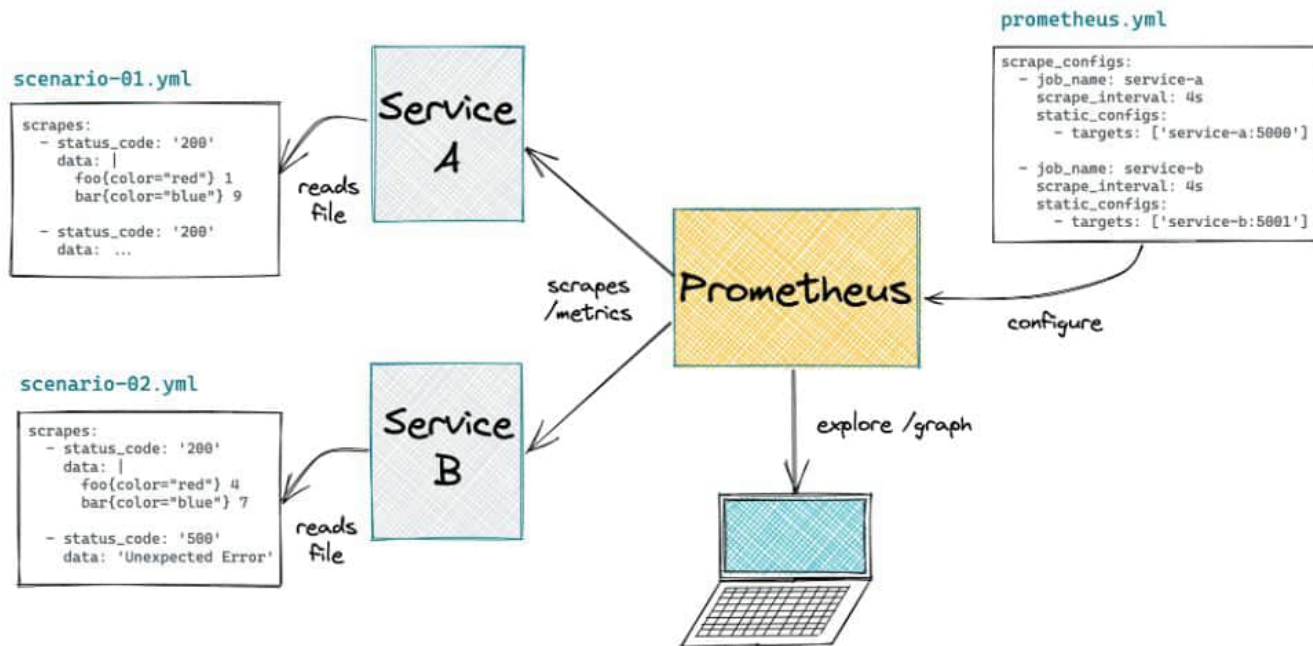
## Гистограммы и более сложные метрики





# Prometheus

## Скрапинг: Как Prometheus собирает данные





# Prometheus

Настроим Prometheus в нашем Spring проекте

Maven:

```
1 <dependency>
2     <groupId>io.micrometer</groupId>
3     <artifactId>micrometer-registry-prometheus</artifactId>
4 </dependency>
```

Gradle:

```
1 implementation 'io.micrometer:micrometer-registry-prometheus'
```



# Prometheus

## Включим экспозицию метрик

Теперь, в `application.properties` или `application.yml`, добавим следующую строчку:

```
1 management.endpoints.web.exposure.include=metrics,prometheus
```

## Настраиваем Prometheus

Скачайте Prometheus с официального сайта и распакуйте архив. В папке с Prometheus найдете файл `prometheus.yml`.

Добавьте в него конфигурацию для вашего Spring приложения.

```
1 scrape_configs:
2   - job_name: 'spring-actuator'
3     metrics_path: '/actuator/prometheus'
4     static_configs:
5       - targets: ['localhost:8080']
```

# Grafana для визуализации

## Запуск Grafana

Первое, что нужно сделать, это установить Grafana на ваш компьютер или сервер, где у вас уже крутится Prometheus.





# Grafana для визуализации

## Подключение к Prometheus

Теперь, открываем веб-интерфейс Grafana. По умолчанию это `http://localhost:3000/`. Входите с помощью логина `admin` и пароля `admin`.

Первым делом, давайте добавим Prometheus как источник данных.

Заходим в `Settings > Data Sources > Add data source` и выбираем Prometheus.

В поле `HTTP URL` вводим адрес, по которому доступен ваш Prometheus (обычно это `http://localhost:9090`).

Сохраняем и тестируем. Если все настроено правильно, Grafana скажет, что все отлично.



# Grafana для визуализации

## Первый Дашборд

Переходим в Create > Dashboard > Add new panel.

Давайте отобразим количество HTTP-запросов к нашему приложению.

В поле «Metrics» вбиваем что-то вроде:

```
1 rate(http_server_requests_seconds_count[5m])|
```



# Grafana для визуализации

## Создание Алерта

Переходим к уже созданному дашборду и выбираем нужный нам панель с графиком. Жмем на заголовок панели и выбираем «Edit». Далее переходим на вкладку «Alert».

Нажимаем «Create Alert» и дадим ему имя. Сразу под именем увидим блок «Conditions». Здесь мы настроим условия срабатывания алерта.

## Настройка Уведомлений

Под блоком «Conditions» будет раздел «Notifications». Здесь можно настроить, куда будет отправлено уведомление.

Создайте новый «Notification channel» в Settings > Notification channels > Add channel, если необходимо. Затем вернитесь к алерту и выберите этот канал для отправки уведомлений.



## Управление Конфигурациями и Dashboard-as-Code

Как Настроить?

- ✓ JSON-экспорт.
- ✓ Сохранение в Git.
- ✓ Автоматизация.
- ✓ Подключение к Grafana.

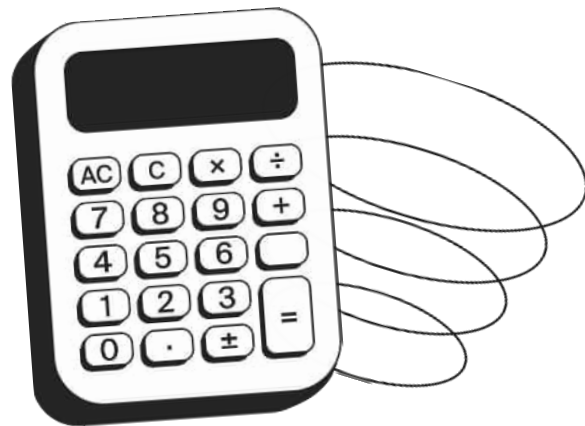




# Snapshot и Sharing Dashboard

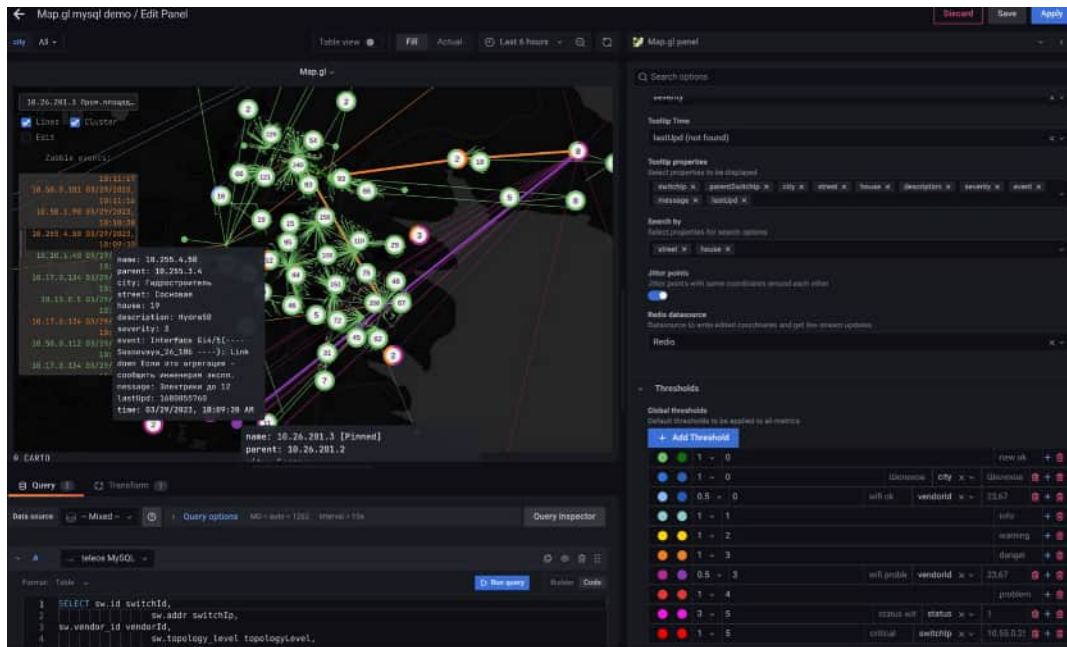
## Сохраняем момент с Snapshot

1. **Как сделать снимок?:** Просто перейдите на дашборд, который хотите сохранить, и выберите Share > Snapshot.
2. **Где хранить?:** Вы можете сохранить снимок либо на сервере Grafana, либо опубликовать на [snapshot.raintank.io](https://snapshot.raintank.io), чтобы поделиться с теми, у кого нет доступа к вашей Grafana.



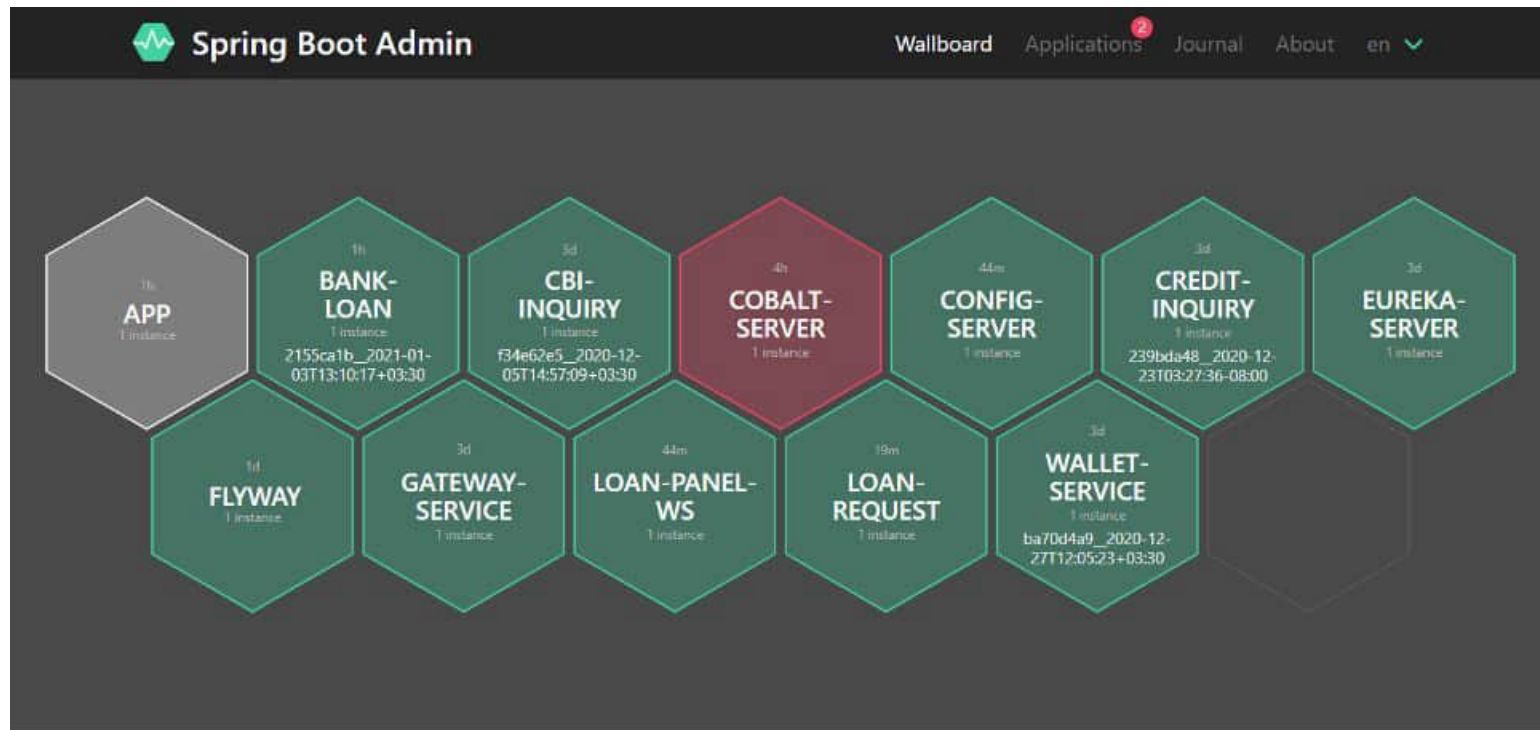
# Snapshot и Sharing Dashboard

## Sharing Dashboard: Быстрый способ поделиться крутой









## Spring Boot Admin





# Spring Boot Admin

Почему стоит обратить внимание?

-  Встроенный мониторинг.
-  Управление приложениями.
-  Связь с Actuator.
-  Уведомления.



# Spring Boot Admin

Как все это настроить?

1. Добавление зависимостей

```
1 <dependency>    <groupId>de.codecentric</groupId>
2 <artifactId>spring-boot-admin-starter-server</artifactId>
3 <version>2.5.1</version>    </dependency>
```




2. Настройка application.properties: Включаем админ-сервер.

```
1 spring.boot.admin.context-path=/admin|
```

3. Регистрация приложений.

## Spring Boot Admin

Оптимизация запросов, чтобы не ждать вечность

-  Сокращение временного диапазона.
-  Использование агрегации.
-  Кеширование.





## На практике



Проблема: Падение производительности в пиковые часы

Представьте, что вы работаете в стартапе, который разрабатывает социальное медиа приложение.

Все было бы хорошо, но в определенные часы (вечер пятницы, к примеру) нагрузка на сервера увеличивается в разы.

И вот приложение начинает тормозить, пользователи недовольны.



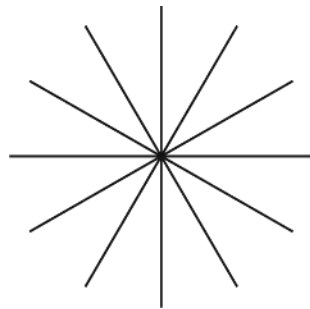
## На практике

### Диагностика

Сначала, давайте взглянем на метрики в Grafana, которые мы собрали с помощью Prometheus и Actuator.

Особенно интересны метрики, связанные с временем ответа сервера и загрузкой базы данных.

Ух ты, а тут оказывается, что время ответа API увеличилось в 4 раза, а CPU базы данных просто в огне.







## На практике



Решение: Оптимизация запросов и кеширование

- 1. Оптимизация SQL запросов:** Первым делом мы обратили внимание на медленные SQL запросы и оптимизировали их. Это сразу дало нам прирост в производительности.
- 2. Введение кеширования:** Мы внедрили кеширование для самых частых и тяжелых запросов, чтобы снизить нагрузку на базу данных.
- 3. Автоскейлинг:** И последний шаг — автоскейлинг наших серверов, чтобы увеличивать их количество в пиковые периоды.



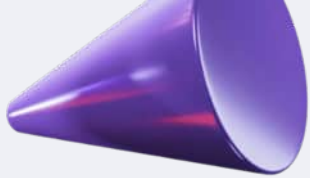
## Spring Actuator на страже порядка

И вот тут на помощь приходит Spring Actuator.

Этот набор полезных инструментов позволяет «под капот» заглянуть, а главное — сделать это быстро и без боли. Не нужно ничего дополнительно настраивать, всё уже есть «из коробки».

Понимание этой библиотеки и умение работать с ней могут сэкономить вам уйму времени и нервов.

Плюс, эта штука прекрасно интегрируется с такими мощными инструментами, как Prometheus и Grafana, которые открывают перед вами почти безграничные возможности в плане мониторинга и анализа данных.



**Спасибо за внимание**

