8. 액츄에이터

#1.인강/9. 스프링부트/강의#

프로덕션 준비 기능이란?

`전투에서 실패한 지휘관은 용서할 수 있지만 경계에서 실패하는 지휘관은 용서할 수 없다`라는 말이 있다. 이 말을 서비스를 운영하는 개발자에게 맞추어 보면 장애는 언제든지 발생할 수 있다. 하지만 모니터링(경계)은 잘 대응하는 것이 중요하다.

개발자가 애플리케이션을 개발할 때 기능 요구사항만 개발하는 것은 아니다. 서비스를 실제 운영 단계에 올리게 되면 개발자들이 해야하는 또 다른 중요한 업무가 있다. 바로 서비스에 문제가 없는지 모니터링하고 지표들을 심어서 감시하는 활동들이다.

운영 환경에서 서비스할 때 필요한 이런 기능들을 프로덕션 준비 기능이라 한다. 쉽게 이야기해서 프로덕션을 운영에 배포할 때 준비해야 하는 비 기능적 요소들을 뜻한다.

- 지표(metric), 추적(trace), 감사(auditing)
- 모니터링

좀 더 구제적으로 설명하자면, 애플리케이션이 현재 살아있는지, 로그 정보는 정상 설정 되었는지, 커넥션 풀은 얼마나 사용되고 있는지 등을 확인할 수 있어야 한다.

스프링 부트가 제공하는 액추에이터는 이런 프로덕션 준비 기능을 매우 편리하게 사용할 수 있는 다양한 편의 기능들을

제공한다. 더 나아가서 마이크로미터, 프로메테우스, 그라파나 같은 최근 유행하는 모니터링 시스템과 매우 쉽게 연동할 수 있는 기능도 제공한다.

참고로 액추에이터는 시스템을 움직이거나 제어하는 데 쓰이는 기계 장치라는 뜻이다.
여러 설명보다 한번 만들어서 실행해보는 것이 더 빨리 이해가 될 것이다.

프로젝트 설정

프로젝트 설정 순서

1. `actuator-start`의 폴더 이름을 `actuator`로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

```
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-actuator' //
actuator 추가

    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //test lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
- `Spring Boot Actuator`, `Spring Web`, `Spring Data JPA`, `H2 Database`, `Lombok`
- 테스트 코드에서 lombok을 사용할 수 있도록 설정을 추가했다.

Spring Boot Actuator 라이브러리를 추가한 부분을 확인하자.

동작 확인

- 기본 메인 클래스 실행(`ActuatorApplication.main()`)
- http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

액츄에이터 시작

액츄에이터가 제공하는 프로덕션 준비 기능을 사용하려면 스프링 부트 액츄에이터 라이브러리를 추가해야 한다.
참고로 앞의 프로젝트 설정에서 추가해두었다.

build.gradle - 추가

```
implementation 'org.springframework.boot:spring-boot-starter-actuator' //
actuator 추가
```

동작 확인

- 기본 메인 클래스 실행(`ActuatorApplication.main()`)
- http://localhost:8080/actuator 실행

실행 결과

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    }
  }
}
```

- 액츄에이터는 `/actuator` 경로를 통해서 기능을 제공한다.

화면에 보이는 `health` 결과를 제공하는 다음 URL도 실행해보자.

http://localhost:8080/actuator/health

```
{"status": "UP"}
```

- 이 기능은 현재 서버가 잘 동작하고 있는지 애플리케이션의 헬스 상태를 나타낸다.

지금 눈에 보이는 기능은 헬스 상태를 확인할 수 있는 기능 뿐이다. 액츄에이터는 헬스 상태 뿐만 아니라 수 많은 기능을 제공하는데, 이런 기능이 웹 환경에서 보이도록 노출해야 한다.

액츄에이터 기능을 웹에 노출

application.yml - 추가

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

동작 확인

- 기본 메인 클래스 실행(`ActuatorApplication.main()`)
- http://localhost:8080/actuator 실행

실행 결과


```
{
  "_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-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8080/actuator/conditions",
      "templated": false
    },
    "configprops-prefix": {
      "href": "http://localhost:8080/actuator/configprops/{prefix}",
      "templated": true
```

```
    },
    "configprops": {
      "href": "http://localhost:8080/actuator/configprops",
      "templated": false
    },
    "env": {
      "href": "http://localhost:8080/actuator/env",
      "templated": false
    },
    "env-toMatch": {
      "href": "http://localhost:8080/actuator/env/{toMatch}",
      "templated": true
    },
    "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": {
      "href": "http://localhost:8080/actuator/metrics",
      "templated": false
    },
    "metrics-requiredMetricName": {
      "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}",
      "templated": true
    },
    "scheduledtasks": {
      "href": "http://localhost:8080/actuator/scheduledtasks",
      "templated": false
    },
    "mappings": {
      "href": "http://localhost:8080/actuator/mappings",
```

```
        "templated": false
      }
    }
  }
```

액츄에이터가 제공하는 수 많은 기능을 확인할 수 있다.
액츄에이터가 제공하는 기능 하나하나를 엔드포인트라 한다. `health`는 헬스 정보를, `beans`는 스프링 컨테이너에 등록된 빈을 보여준다.

각각의 엔드포인트는 `/actuator/{엔드포인트명}`과 같은 형식으로 접근할 수 있다.

- `http://localhost:8080/actuator/health`: 애플리케이션 헬스 정보를 보여준다.
- `http://localhost:8080/actuator/beans`: 스프링 컨테이너에 등록된 빈을 보여준다.

엔드포인트 설정

엔드포인트를 사용하려면 다음 2가지 과정이 모두 필요하다.

1. 엔드포인트 활성화
2. 엔드포인트 노출

엔드포인트를 활성화 한다는 것은 해당 기능 자체를 사용할지 말지 `on`, `off`를 선택하는 것이다.
엔드포인트를 노출하는 것은 활성화된 엔드포인트를 HTTP에 노출할지 아니면 JMX에 노출할지 선택하는 것이다.
엔드포인트를 활성화하고 추가로 HTTP를 통해서 웹에 노출할지, 아니면 JMX를 통해서 노출할지 두 위치에 모두 노출할지 노출 위치를 지정해주어야 한다.
물론 활성화가 되어있지 않으면 노출도 되지 않는다.

그런데 **엔드포인트는 대부분 기본으로 활성화** 되어 있다.(`shutdown` 제외) 노출이 되어 있지 않을 뿐이다.
따라서 어떤 엔드포인트를 노출할지 선택하면 된다. 참고로 HTTP와 JMX를 선택할 수 있는데, 보통 JMX는 잘 사용하지 않으므로 HTTP에 어떤 엔드포인트를 노출할지 선택하면 된다.

application.yml - 모든 엔드포인트를 웹에 노출

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

`"*"` 옵션은 모든 엔드포인트를 웹에 노출하는 것이다. 참고로 `shutdown` 엔드포인트는 기본으로 활성화 되지 않기

때문에 노출도 되지 않는다.

엔드포인트 활성화 + 엔드포인트 노출이 둘다 적용되어야 사용할 수 있다.

엔드포인트 활성화

application.yml - shutdown 엔드포인트 활성화

```yaml
management:
  endpoint:
    shutdown:
      enabled: true
  endpoints:
    web:
      exposure:
        include: "*"
```

특정 엔드포인트를 활성화 하려면 `management.endpoint.{엔드포인트명}.enabled=true`를 적용하면 된다.

이제 Postman 같은 것을 사용해서 HTTP **POST**로 `http://localhost:8080/actuator/shutdown`를 호출하면 다음 메시지와 함께 실제 서버가 종료되는 것을 확인할 수 있다.

```
{"message": "Shutting down, bye..."}
```

참고로 HTTP GET으로 호출하면 동작하지 않는다.

물론 이 기능은 주의해서 사용해야 한다. 그래서 기본으로 비활성화 되어 있다.

엔드포인트 노출

스프링 공식 메뉴얼이 제공하는 예제를 통해서 엔드포인트 노출 설정을 알아보자

```yaml
management:
  endpoints:
    jmx:
      exposure:
        include: "health,info"
```

- `jmx`에 `health,info`를 노출한다.

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
        exclude: "env,beans"
```

- `web`에 모든 엔드포인트를 노출하지만 `env`, `beans`는 제외한다.

다양한 엔드포인트

각각의 엔드포인트를 통해서 개발자는 애플리케이션 내부의 수 많은 기능을 관리하고 모니터링 할 수 있다.

스프링 부트가 기본으로 제공하는 다양한 엔드포인트에 대해서 알아보자. 다음은 자주 사용하는 기능 위주로 정리했다.

엔드포인트 목록

- `beans` : 스프링 컨테이너에 등록된 스프링 빈을 보여준다.
- `conditions` : `condition` 을 통해서 빈을 등록할 때 평가 조건과 일치하거나 일치하지 않는 이유를 표시한다.
- `configprops` : `@ConfigurationProperties` 를 보여준다.
- `env` : `Environment` 정보를 보여준다.
- `health` : 애플리케이션 헬스 정보를 보여준다.
- `httpexchanges` : HTTP 호출 응답 정보를 보여준다. `HttpExchangeRepository` 를 구현한 빈을 별도로 등록해야 한다.
- `info` : 애플리케이션 정보를 보여준다.
- `loggers` : 애플리케이션 로거 설정을 보여주고 변경도 할 수 있다.
- `metrics` : 애플리케이션의 메트릭 정보를 보여준다.
- `mappings` : `@RequestMapping` 정보를 보여준다.
- `threaddump` : 쓰레드 덤프를 실행해서 보여준다.
- `shutdown` : 애플리케이션을 종료한다. 이 기능은 **기본으로 비활성화** 되어 있다.

> **전체 엔드포인트는 다음 공식 메뉴얼을 참고하자.**
> https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints

`health` , `info` , `loggers` , `httpexchanges` , `metrics` 는 뒤에서 더 자세히 알아보겠다.

헬스 정보

헬스 정보를 사용하면 애플리케이션에 문제가 발생했을 때 문제를 빠르게 인지할 수 있다.

http://localhost:8080/actuator/health

기본 동작

```
{"status": "UP"}
```

헬스 정보는 단순히 애플리케이션이 요청에 응답을 할 수 있는지 판단하는 것을 넘어서 애플리케이션이 사용하는 데이터베이스가 응답하는지, 디스크 사용량에는 문제가 없는지 같은 다양한 정보들을 포함해서 만들어진다.

헬스 정보를 더 자세히 보려면 다음 옵션을 지정하면 된다.

`management.endpoint.health.show-details=always`

```
management:
  endpoint:
    health:
      show-details: always
```

show-details 옵션

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "H2",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 994662584320,
        "free": 303418753024,
        "threshold": 10485760,
        "path": ".../spring-boot/actuator/actuator/.",
        "exists": true
      }
    },
    "ping": {
      "status": "UP"
    }
  }
```

```
}
```

각각의 항목이 아주 자세하게 노출되는 것을 확인할 수 있다.

이렇게 자세하게 노출하는 것이 부담스럽다면 `show-details` 옵션을 제거하고 대신에 다음 옵션을 사용하면 된다.

`management.endpoint.health.show-components=always`

```yaml
management:
  endpoint:
    health:
      show-components: always
```

show-components 옵션

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP"
    },
    "diskSpace": {
      "status": "UP"
    },
    "ping": {
      "status": "UP"
    }
  }
}
```

각 헬스 컴포넌트의 상태 정보만 간략하게 노출한다.

헬스 이상 상태

헬스 컴포넌트 중에 하나라도 문제가 있으면 전체 상태는 `DOWN`이 된다.

```
{
  "status": "DOWN",
  "components": {
    "db": {
      "status": "DOWN"
    },
    "diskSpace": {
      "status": "UP"
    },
    "ping": {
      "status": "UP"
```

```
        }
    }
}
```

여기서는 `db`에 문제가 발생했다. 하나라도 문제가 있으면 `DOWN`으로 보기 때문에 이 경우 전체 상태의 `status`도 `DOWN`이 된다.

참고로 액츄에이터는 `db`, `mongo`, `redis`, `diskspace`, `ping`과 같은 수 많은 헬스 기능을 기본으로 제공한다.

> **참고 - 자세한 헬스 기본 지원 기능은 다음 공식 메뉴얼을 참고하자**
> https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.health.auto-configured-health-indicators

> **참고 - 헬스 기능 직접 구현하기**
> 원하는 경우 직접 헬스 기능을 구현해서 추가할 수 있다. 직접 구현하는 일이 많지는 않기 때문에 필요한 경우 다음 공식 메뉴얼을 참고하자
> https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.health.writing-custom-health-indicators

애플리케이션 정보

`info` 엔드포인트는 애플리케이션의 기본 정보를 노출한다.

기본으로 제공하는 기능들은 다음과 같다.

- `java`: 자바 런타임 정보
- `os`: OS 정보
- `env`: `Environment`에서 `info.`로 시작하는 정보
- `build`: 빌드 정보, `META-INF/build-info.properties` 파일이 필요하다.
- `git`: `git` 정보, `git.properties` 파일이 필요하다.

`env`, `java`, `os`는 기본으로 비활성화 되어 있다.

실행

http://localhost:8080/actuator/info

처음에 실행하면 정보들이 보이지 않을 것이다. `java`, `os` 기능을 활성화해보자.

java, os

JAVA, OS 정보를 확인해보자.

application.yml - 내용 추가

```yaml
management:
  info:
    java:
      enabled: true
    os:
      enabled: true
```

`management.info.<id>.enabled`의 값을 `true`로 지정하면 활성화 된다.

주의!: management.endpoint 하위가 아니다. management 바로 다음에 info가 나온다.

실행 결과

http://localhost:8080/actuator/info


```
{
   "java":{
      "version":"17.0.3",
      "vendor":{
         "name":"JetBrains s.r.o.",
         "version":"JBR-17.0.3+7-469.37-jcef"
      },
      "runtime":{
         "name":"OpenJDK Runtime Environment",
         "version":"17.0.3+7-b469.37"
      },
      "jvm":{
         "name":"OpenJDK 64-Bit Server VM",
         "vendor":"JetBrains s.r.o.",
         "version":"17.0.3+7-b469.37"
      }
   },
   "os":{
      "name":"Mac OS X",
      "version":"12.5.1",
      "arch":"aarch64"
   }
}
```

실행해보면 `java`, `os` 관련 정보를 확인할 수 있다.

env

이번에는 `env`를 사용해보자.

`Environment`에서 `info.`로 시작하는 정보를 출력한다.

application.yml - 내용 추가

```
management:
  info:
    env:
      enabled: true

info:
  app:
    name: hello-actuator
    company: yh
```

`management.info.env.enabled`를 추가하고, `info..` 관련 내용을 추가했다.

실행 결과

http://localhost:8080/actuator/info

```
{
   "app":{
      "name":"hello-actuator",
      "company":"yh"
   }
   ...
}
```

`application.yml`에서 `info`로 시작하는 부분의 정보가 노출되는 것을 확인할 수 있다.

build

이번에는 빌드 정보를 노출해보자. 빌드 정보를 노출하려면 빌드 시점에 `META-INF/build-info.properties` 파일을 만들어야 한다.

`gradle`을 사용하면 다음 내용을 추가하면 된다.

build.gradle - 빌드 정보 추가

```
springBoot {
    buildInfo()
```

```
}
```

이렇게 하고 빌드를 해보면 `build` 폴더안에 `resources/main/META-INF/build-info.properties` 파일을 확인할 수 있다.

```
build.artifact=actuator
build.group=hello
build.name=actuator
build.time=2023-01-01T00\:000\:00.000000Z
build.version=0.0.1-SNAPSHOT
```

`build`는 기본으로 활성화 되어 있기 때문에 이 파일만 있으면 바로 확인할 수 있다.

실행 결과

http://localhost:8080/actuator/info

```
{
   "build":{
      "artifact":"actuator",
      "name":"actuator",
      "time":"2023-01-01T00:00:00.000Z",
      "version":"0.0.1-SNAPSHOT",
      "group":"hello"
   }
   ...
}
```

실행 결과를 통해서 애플리케이션의 기본 정보와 버전 그리고 빌드된 시간을 확인할 수 있다.

git

앞서본 `build`와 유사하게 빌드 시점에 사용한 `git` 정보도 노출할 수 있다. `git` 정보를 노출하려면 `git.properties` 파일이 필요하다.

build.gradle - git 정보 추가

```
plugins {
    ...
    id "com.gorylenko.gradle-git-properties" version "2.4.1" //git info
}
```

물론 프로젝트가 `git`으로 관리되고 있어야 한다. 그렇지 않으면 빌드시 오류가 발생한다.

프로젝트에 `git`을 적용하고 커밋해보자.

> **참고**
> `git`을 설명하는 것은 이 강의 범위를 넘어선다. `git`을 잘 모르는 분들은 별도로 `git`을 배운 다음에 이 부분을 다시 학습하자.

이렇게 하고 빌드를 해보면 `build` 폴더안에 `resources/main/git.properties` 파일을 확인할 수 있다.

```
git.branch=main
git.build.host=kim
git.build.user.email=zipkyh@mail.com
git.build.user.name=holyeye
git.build.version=0.0.1-SNAPSHOT
git.closest.tag.commit.count=
git.closest.tag.name=
git.commit.id=754bc78744107b6423352018e46367f5091b181e
git.commit.id.abbrev=754bc78
git.commit.id.describe=
git.commit.message.full=fitst commit\n
git.commit.message.short=fitst commit
git.commit.time=2023-01-01T00\:00\:00+0900
git.commit.user.email=zipkyh@mail.com
git.commit.user.name=holyeye
git.dirty=false
git.remote.origin.url=
git.tags=
git.total.commit.count=1
```

`git`은 기본으로 활성화 되어 있기 때문에 이 파일만 있으면 바로 확인할 수 있다.

실행 결과

http://localhost:8080/actuator/info

```
{
   "git":{
      "branch":"main",
      "commit":{
         "id":"754bc78",
         "time":"2023-01-01T00:00:00Z"
      }
   }
   ...
}
```

실행 결과를 통해서 이 빌드는 `main` 브랜치와 `754bc78` 커밋에서 만들어진 것을 확인할 수 있다.
애플리케이션을 배포할 때 가끔 기대와 전혀 다르게 동작할 때가 있는데,(특정 기능이 빠져있다던가) 확인해보면 다른

커밋이나 다른 브랜치의 내용이 배포된 경우가 종종 있다. 이럴 때 큰 도움이 된다.

`git`에 대한 더 자세한 정보를 보고 싶다면 다음 옵션을 적용하면 된다.

application.yml 추가

```
management:
  info:
    git:
      mode: "full"
```

> **info 사용자 정의 기능 추가**
>
> `info`의 사용자 정의 기능을 추가 하고 싶다면 다음 스프링 공식 메뉴얼을 참고하자
>
> https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html#actuator.endpoints.info.writing-custom-info-contributors

로거

`loggers` 엔드포인트를 사용하면 로깅과 관련된 정보를 확인하고, 또 실시간으로 변경할 수도 있다. 코드를 통해서 알아보자.

LogController 생성

```
package hello.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
public class LogController {

    @GetMapping("/log")
    public String log() {
        log.trace("trace log");
        log.debug("debug log");
        log.info("info log");
```

```
        log.warn("warn log");
        log.error("error log");
        return "ok";
    }
}
```

여러 레벨을 로그를 남기는 단순한 컨트롤러이다.

application.yml 설정

```
logging:
  level:
    hello.controller: debug
```

`hello.controller` 패키지와 그 하위는 `debug` 레벨을 출력하도록 했다. 이제 앞서 만든 `LogController` 클래스도 `debug` 레벨의 영향을 받는다.

실행

http://localhost:8080/log

결과 로그

```
DEBUG 53783 --- hello.controller.LogController  : debug log
 INFO 53783 --- hello.controller.LogController  : info log
 WARN 53783 --- hello.controller.LogController  : warn log
ERROR 53783 --- hello.controller.LogController  : error log
```

실행 결과를 보면 기대한 것 처럼 `DEBUG` 레벨까지 출력되는 것을 확인할 수 있다.

`loggers` 엔드포인트를 호출해보자.

실행

http://localhost:8080/actuator/loggers

실행 결과

```
{
   "levels":[
      "OFF",
      "ERROR",
      "WARN",
      "INFO",
      "DEBUG",
      "TRACE"
   ],
```

```
    "loggers":{
       "ROOT":{
          "configuredLevel":"INFO",
          "effectiveLevel":"INFO"
       },
       "_org.springframework":{
          "effectiveLevel":"INFO"
       },
       "hello":{
          "effectiveLevel":"INFO"
       },
       "hello.ActuatorApplication":{
          "effectiveLevel":"INFO"
       },
       "hello.controller":{
          "configuredLevel":"DEBUG",
          "effectiveLevel":"DEBUG"
       },
       "hello.controller.LogController":{
          "effectiveLevel":"DEBUG"
       }
    }
}
```


- 로그를 별도로 설정하지 않으면 스프링 부트는 기본으로 `INFO`를 사용한다. 실행 결과를 보면 `ROOT`의 `configuredLevel`가 `INFO`인 것을 확인할 수 있다. 따라서 그 하위도 모두 `INFO` 레벨이 적용된다.
- 앞서 우리는 `hello.controller`는 `DEBUG`로 설정했다. 그래서 해당 부분에서 `configuredLevel`이 `DEBUG`로 설정된 것을 확인할 수 있다. 그리고 그 하위도 `DEBUG` 레벨이 적용된다.

더 자세히 조회하기

다음과 같은 패턴을 사용해서 특정 로거 이름을 기준으로 조회할 수 있다.

`http://localhost:8080/actuator/loggers/{로거이름}`

실행

http://localhost:8080/actuator/loggers/hello.controller

결과


```
{
  "configuredLevel": "DEBUG",
  "effectiveLevel": "DEBUG"
}
```

실시간 로그 레벨 변경

개발 서버는 보통 `DEBUG` 로그를 사용하지만, 운영 서버는 보통 요청이 아주 많다. 따라서 로그도 너무 많이 남기 때문에 `DEBUG` 로그까지 모두 출력하게 되면 성능이나 디스크에 영향을 주게 된다. 그래서 운영 서버는 중요하다고 판단되는 `INFO` 로그 레벨을 사용한다.
그런데 서비스 운영중에 문제가 있어서 급하게 `DEBUG`나 `TRACE` 로그를 남겨서 확인해야 확인하고 싶다면 어떻게 해야할까? 일반적으로는 로깅 설정을 변경하고, 서버를 다시 시작해야 한다.

`loggers` 엔드포인트를 사용하면 애플리케이션을 다시 시작하지 않고, 실시간으로 로그 레벨을 변경할 수 있다.

다음을 Postman 같은 프로그램으로 POST로 요청해보자(**꼭! POST를 사용해야 한다.**)

POST http://localhost:8080/actuator/loggers/hello.controller

POST로 전달하는 내용 JSON , `content/type`도 `application/json`으로 전달해야 한다.

```
{
    "configuredLevel": "TRACE"
}
```

참고로 이것은 POST에 전달하는 내용이다. 응답 결과가 아니다.
요청에 성공하면 `204` 응답이 온다.(별도의 응답 메시지는 없다.)

GET으로 요청해서 확인해보면 `configuredLevel`이 `TRACE`로 변경된 것을 확인할 수 있다.

GET http://localhost:8080/actuator/loggers/hello.controller

호출 결과

```
{
    "configuredLevel": "TRACE",
    "effectiveLevel": "TRACE"
}
```

정말 로그 레벨이 실시간으로 변경되었는지 확인해보자.

실행

http://localhost:8080/log

결과 로그

```
TRACE 53783 --- [nio-8080-exec-6] hello.controller.LogController  : trace log
DEBUG 53783 --- [nio-8080-exec-6] hello.controller.LogController  : debug log
```

```
 INFO 53783 --- [nio-8080-exec-6] hello.controller.LogController  : info log
 WARN 53783 --- [nio-8080-exec-6] hello.controller.LogController  : warn log
ERROR 53783 --- [nio-8080-exec-6] hello.controller.LogController  : error log
```

실행 결과를 보면 `TRACE` 레벨까지 출력되는 것을 확인할 수 있다.

HTTP 요청 응답 기록

HTTP 요청과 응답의 과거 기록을 확인하고 싶다면 `httpexchanges` 엔드포인트를 사용하면 된다.

`HttpExchangeRepository` 인터페이스의 구현체를 빈으로 등록하면 `httpexchanges` 엔드포인트를 사용할 수 있다.
(주의! 해당 빈을 등록하지 않으면 `httpexchanges` 엔드포인트가 활성화 되지 않는다)

스프링 부트는 기본으로 `InMemoryHttpExchangeRepository` 구현체를 제공한다.

InMemoryHttpExchangeRepository 추가

```
@SpringBootApplication
public class ActuatorApplication {

    public static void main(String[] args) {
        SpringApplication.run(ActuatorApplication.class, args);
    }

    @Bean
    public InMemoryHttpExchangeRepository httpExchangeRepository() {
        return new InMemoryHttpExchangeRepository();
    }

}
```

이 구현체는 최대 100개의 HTTP 요청을 제공한다. 최대 요청이 넘어가면 과거 요청을 삭제한다. `setCapacity()`로 최대 요청수를 변경할 수 있다.

실행

http://localhost:8080/actuator/httpexchanges

실행해보면 지금까지 실행한 HTTP 요청과 응답 정보를 확인할 수 있다.

참고로 이 기능은 매우 단순하고 기능에 제한이 많기 때문에 개발 단계에서만 사용하고, 실제 운영 서비스에서는 모니터링 툴이나 핀포인트, Zipkin 같은 다른 기술을 사용하는 것이 좋다.

액츄에이터와 보안

보안 주의

액츄에이터가 제공하는 기능들은 우리 애플리케이션의 내부 정보를 너무 많이 노출한다. 그래서 외부 인터넷 망이 공개된 곳에 액츄에이터의 엔드포인트를 공개하는 것은 보안상 좋은 방안이 아니다.
액츄에이터의 엔드포인트들은 외부 인터넷에서 접근이 불가능하게 막고, 내부에서만 접근 가능한 내부망을 사용하는 것이 안전하다.

액츄에이터를 다른 포트에서 실행

예를 들어서 외부 인터넷 망을 통해서 8080 포트에만 접근할 수 있고, 다른 포트는 내부망에서만 접근할 수 있다면 액츄에이터에 다른 포트를 설정하면 된다.
액츄에이터의 기능을 애플리케이션 서버와는 다른 포트에서 실행하려면 다음과 같이 설정하면 된다. 이 경우 기존 8080 포트에서는 액츄에이터를 접근할 수 없다.

액츄에이터 포트 설정

```
management.server.port=9292
```

실행

http://localhost:9292/actuator

액츄에이터 URL 경로에 인증 설정

포트를 분리하는 것이 어렵고 어쩔 수 없이 외부 인터넷 망을 통해서 접근해야 한다면 `/actuator` 경로에 서블릿 필터, 또는 스프링 시큐티리를 통해서 인증된 사용자만 접근 가능하도록 추가 개발이 필요하다.

엔드포인트 경로 변경

엔드포인트의 기본 경로를 변경하려면 다음과 같이 설정하면 된다.

application.yml

```
management:
```

```
endpoints:
  web:
    base-path: "/manage"
```

- `/actuator/{엔드포인트}` 대신에 `/manage/{엔드포인트}`로 변경된다.

정리