



LẬP TRÌNH JAVA SPRING BOOT

**Bài 6: Spring Boot Properties, Logging, AOP,
Validation**

- ❑ Kết thúc bài học này bạn có khả năng
 - ❖ Hiểu về file application.properties và file YAML trong Spring Boot
 - ❖ Biết cách sử dụng biến từ file properties vào code Java
 - ❖ Biết cách Logging để output hoặc ghi ra vào file log ứng dụng
 - ❖ Hiểu lập trình khía cạnh AOP
 - ❖ Biết cách sử dụng các annotation để validation trong Spring Boot
-


Nêu Vấn Đề?

- ❑ Giả sử, ứng dụng của tôi sẽ yêu cầu có một số giá trị toàn cục, mà thay vì cấu hình ở trong code, tôi muốn lưu nó ở bên ngoài, để tiện thay đổi và lấy các thông tin đó ra mỗi khi cần

Giải Pháp


- ❑ Spring Boot cho phép chúng ta **cấu hình** ứng dụng từ **bên ngoài** và lấy các thông tin đó ra một cách dễ dàng
- ❑ Mặc định Spring Boot sẽ đọc thông tin từ một file properties **chuẩn**
- ❑ File đó được đặt tại: ***src/main/resources/application.propterties***
- ❑ ***application.propterties*** là tên file properties tiêu chuẩn của Spring Boot
- ❑ Bạn có thể tự định nghĩa properties của bạn trong file này

- ❑ **Bước 1:** Định nghĩa các properties trong file `application.properties`



```
# Định nghĩa properties
coach.name=Mickey Mouse
team.name=The Mouse Club
```

- ❑ **Bước 2:** Sử dụng `@Value` để tiêm giá trị từ file properties vào code Java



```
import org.springframework.beans.factory.annotation.Value;
...
@RestController
@RequestMapping("/fun")
public class FunRestController {

    @Value("${coach.name}")
    private String coachName;
    @Value("${team.name}")
    private String teamName;

    @GetMapping("/getInfo")
    public String getInfo() {
        return "Coach name: " + coachName + ", Team Name: " +
teamName;
    }
}
```

- ❑ **Lưu ý:** `@Value` được import từ `org.springframework.beans.factory.annotation.Value`

- ❏ Request vào đường dẫn <http://localhost:8080/fun/getInfo> và xem kết quả

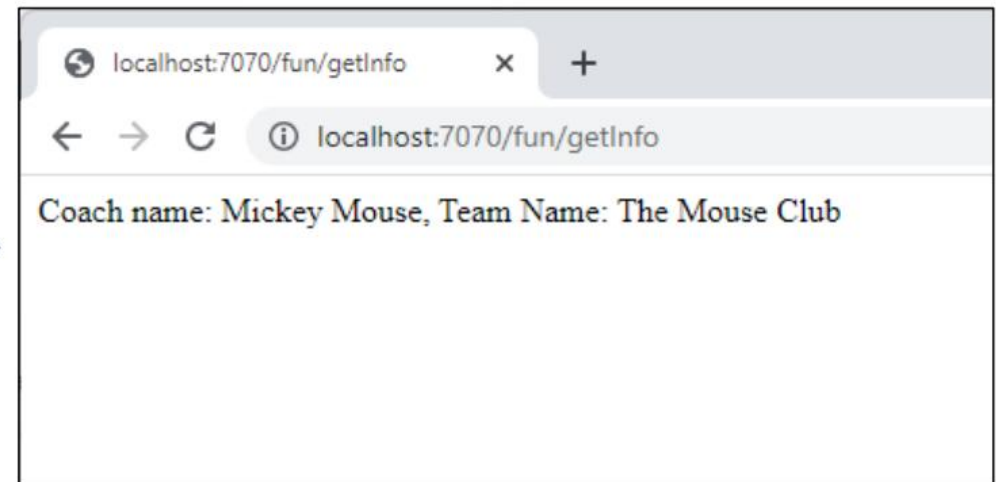


- ❑ Có thể cấu hình ứng dụng Spring Boot trong file **application.properties**
 - ❑ Ví dụ: Server port, context path, actuator, security etc...
 - ❑ Spring Boot có hơn +1000 properties...
 - ❑ Truy cập vào <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html> để có thể biết thêm những property phù hợp cấu hình cho ứng dụng
 - ❑ **Lưu ý:** đừng cố gắng sử dụng hết các properties, chỉ sử dụng những cấu hình phù hợp cho ứng dụng
-

- ❑ Spring Boot hỗ trợ gợi ý các cấu hình cho ứng dụng
- ❑ Cấu hình port cho ứng dụng là 7070 bằng properties `server.port=7070`

```
5 # HTTP server port
6 server.port=7070
7 ser
8 p server.address (Network address to which th... InetAddress
9 p server.compression.enabled=false (Whether respo... Boolean
10 p server.compression.excluded-user-agents (Comma... String[]
11 p server.compression.mime-types=text/html, text/... String[]
12 p server.compression.min-response-size=2KB (Mini... DataSize
13 p server.error.include-binding-errors=ne... IncludeAttribute
14 p server.error.include-exception=false (Include t... Boolean
15 p server.error.include-message=never (Wh... IncludeAttribute
16 p server.error.include-stacktrace=never ... IncludeAttribute
17 p server.error.path=/error (Path of the error cont... String
18 p server.error.whitelabel.enabled=true (Whether t... Boolean
19 p server.forward-headers-strategy ForwardHeadersStrategy
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip
```

- ❑ Run ứng dụng và truy cập vào <http://localhost:7070/fun/getInfo> để test port 7070



- ❑ YAML (YAML Ain't Markup Language) là một loại ngôn ngữ tương tự JSON được thiết kế với mục đích để người và máy cùng đọc được

- ❑ Ví dụ

```
message: Hello
```

```
menus:
```

- title: Home
name: Home
path: /
- title: Login
name: Login
path: /login

- ❑ Trong Spring Boot file YAML (.yml) được dùng để làm file config (tương đương với file .properties)
 - ❑ Ta có thể dùng file YAML hoặc file Properties hoặc kết hợp cả 2
 - ❑ Việc đọc file YAML trong Spring Boot giống với việc đọc file Properties. Chúng ta cũng sử dụng các annotation **@Value**, **@ConfigurationProperties**
-

- ❑ Spring boot cung cấp một annotation cho phép truy xuất các thuộc tính được đặt trong các tệp cấu hình một cách tự động là **@ConfigurationProperties**

 **Vậy @Value và @ConfigurationProperties khác nhau như thế nào?**

- ❑ Với **@ConfigurationProperties** sẽ chú thích class bên dưới là các properties tương ứng các thuộc tính sẽ được tự động nạp vào
- ❑ Với **@Value** chú thích trên từng thuộc tính trong code Java với tên tương ứng các properties trong tệp file cấu hình
- ❑ **Lưu ý:** khi dùng **@ConfigurationProperties**
 - ❖ Tạo các hàm Setter cho class
 - ❖ Kể từ spring boot **2.2**, nó sẽ tự động tìm kiếm và đăng ký
 - ❖ Nếu dùng phiên bản nhỏ hơn **2.2**. Sử dụng thêm annotation **@Configuration** để chú thích lên class
 - ❖ Hoặc sử dụng **@EnableConfigurationProperties** chú thích vào hàm main của spring application

- ❑ Tại class cấu hình properties: **LikelionProperties**

```
@Data // Lombok
@Component // Là 1 spring bean
// Đánh dấu để lấy config từ trong file likelion.properties
// @PropertySource("classpath:likelion.properties")
@ConfigurationProperties(prefix = "likelion") // Chỉ lấy các config có tiền tố là "likelion"
public class LikelionProperties {
    private String address;
    private String phone;
    private String website;
}
```

- ❑ Tại file **application.yml**
- ❑ **Không** phân biệt hoa, thường, gạch dưới hay gạch ngang

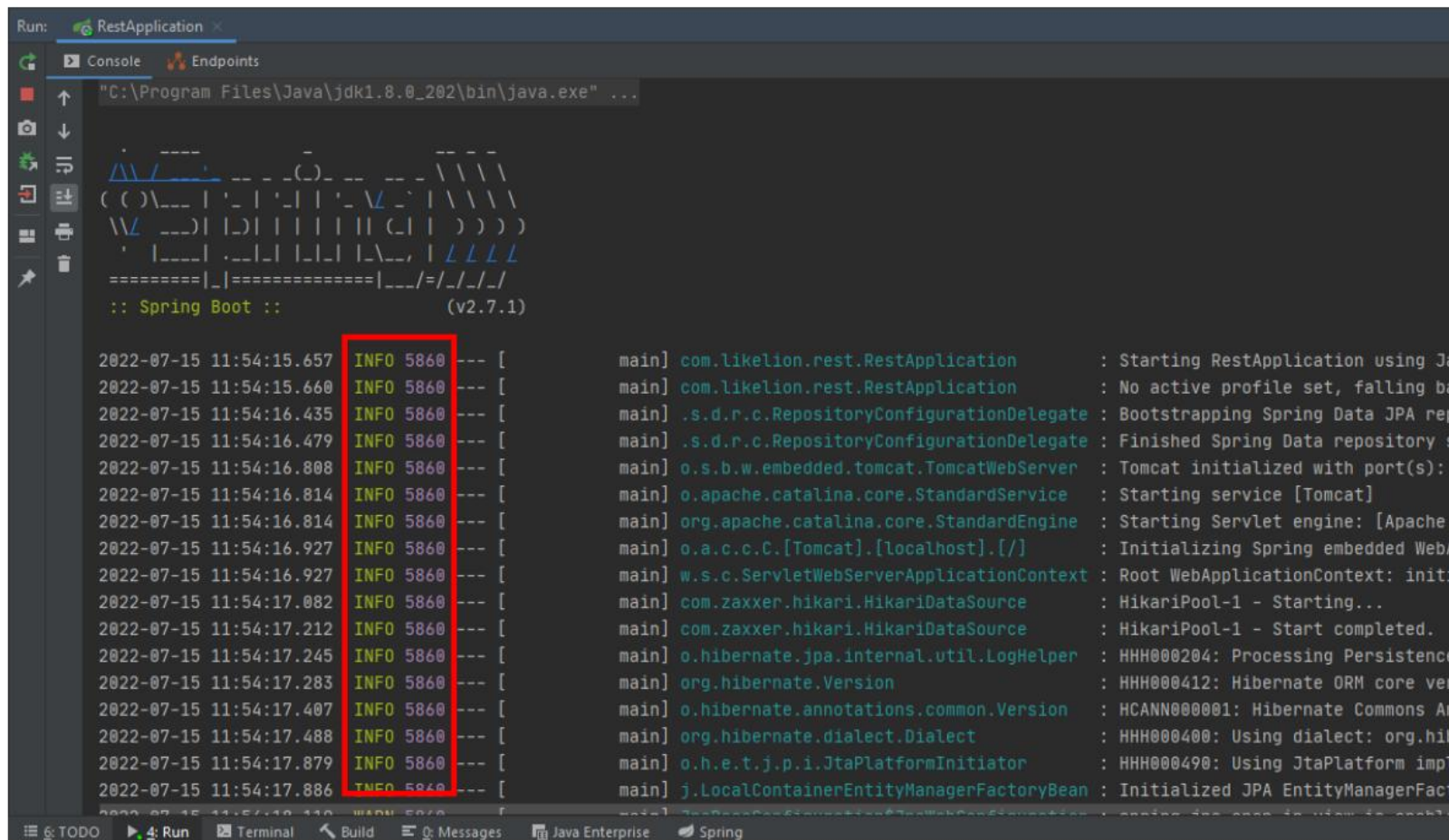
```
likelion:
  add-ress: 76 Le Lai
  PHONE: (+84) 90 8885 090
  web_site: likelion.vn@likelion.net
```

- ❑ **Console**



```
Address: 76 Le Lai
Phone: (+84) 90 8885 090
Website: likelion.vn@likelion.net
```

- Hiểu một cách đơn giản "Logging" là "ghi chép" lại các vấn đề trong quá trình ứng dụng hoạt động. Các vấn đề ở đây là các thông tin lỗi, các cảnh báo (warning), và các thông tin khác, ... Các thông tin này có thể được hiển thị trên màn hình Console hoặc ghi vào file



The screenshot shows the console output of a Spring Boot application named 'RestApplication'. The log displays a series of 'INFO' messages from the main thread, indicating the successful initialization of various components. A red box highlights the first 15 log entries, which are all 'INFO' messages with the logger name '5860'. The messages include the start of the application, the setup of Spring Data JPA, the initialization of Tomcat, the starting of the Servlet engine, the initialization of the Spring embedded WebApplicationContext, the starting of HikariPool-1, and the initialization of the JPA EntityManagerFactoryBean.

```
Run: RestApplication x
Console Endpoints
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...

:: Spring Boot :: (v2.7.1)

2022-07-15 11:54:15.657 INFO 5860 --- [main] com.likelion.rest.RestApplication : Starting RestApplication using Ja
2022-07-15 11:54:15.660 INFO 5860 --- [main] com.likelion.rest.RestApplication : No active profile set, falling ba
2022-07-15 11:54:16.435 INFO 5860 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA rep
2022-07-15 11:54:16.479 INFO 5860 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository s
2022-07-15 11:54:16.808 INFO 5860 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s):
2022-07-15 11:54:16.814 INFO 5860 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-07-15 11:54:16.814 INFO 5860 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache
2022-07-15 11:54:16.927 INFO 5860 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebA
2022-07-15 11:54:16.927 INFO 5860 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initi
2022-07-15 11:54:17.082 INFO 5860 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2022-07-15 11:54:17.212 INFO 5860 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2022-07-15 11:54:17.245 INFO 5860 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing Persistence
2022-07-15 11:54:17.283 INFO 5860 --- [main] org.hibernate.Version : HHH000412: Hibernate ORM core ver
2022-07-15 11:54:17.407 INFO 5860 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons An
2022-07-15 11:54:17.488 INFO 5860 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hib
2022-07-15 11:54:17.879 INFO 5860 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform imp
2022-07-15 11:54:17.886 INFO 5860 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFac
```


- ❑ Spring Boot có thể làm việc với một số thư viện Logging như **Logback**, **Log4j2**, **JUL**,...
- ❑ Mặc định Spring Boot đã tự động cấu hình và sử dụng thư viện **Logback** cho **logging**
- ❑ Các Thuộc tính (properties) mà bạn có thể tùy biến

logging.config

logging.exception-conversion-word

logging.file

logging.level.*

logging.path

logging.pattern.console

logging.pattern.file

logging.pattern.level

logging.register-shutdown-hook

```
@RestController
@RequestMapping("/fun")
public class FunRestController {

    private static final Logger LOGGER = LoggerFactory.getLogger(FunRestController.class);

    @GetMapping("/getLogging")
    public String home() {
        LOGGER.trace("This is TRACE");
        LOGGER.debug("This is DEBUG");
        LOGGER.info("This is INFO");
        LOGGER.warn("This is WARN");
        LOGGER.error("This is ERROR");
        return "Hi, show loggings in the console or file!";
    }
}
```

- ❑ Chạy ứng dụng, sau đó truy cập vào đường dẫn <http://localhost:8080/>
- ❑ Trên cửa sổ Console bạn có thể nhìn thấy thông tin Logs như sau

```
INFO 13268 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is INFO
WARN 13268 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is WARN
ERROR 13268 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is ERROR
```


- ❑ Dựa trên mức độ nghiêm trọng của vấn đề, **Logback** chia các thông tin cần ghi chép thành **5 mức độ** (Level), loại ít nghiêm trọng nhất là **TRACE**, và loại nghiêm trọng nhất là **ERROR**
 - ❑ Chú ý: Có một số thư viện **Logging** phân chia các thông tin cần ghi chép thành 7 mức độ khác nhau
 1. TRACE
 2. DEBUG
 3. INFO
 4. WARN
 5. ERROR
 6. FATAL
 7. OFF
 - ❑ Theo mặc định Spring Boot chỉ ghi lại các thông tin có độ nghiêm trọng từ mức **INFO** trở lên
-

Thay đổi Logging Level

```
# Logging
logging.level.root=WARN
...
```

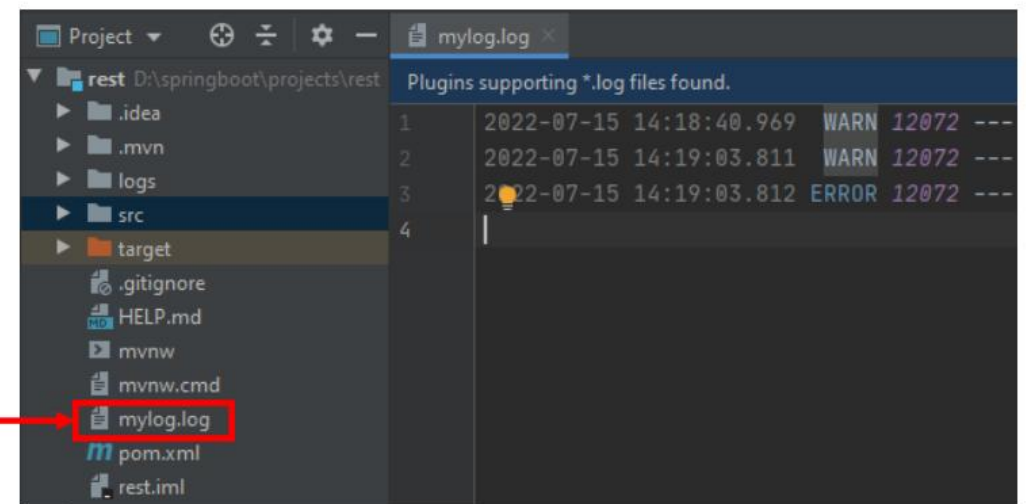
- ❑ Chạy lại ứng dụng và xem kết quả trên cửa sổ Console

```
WARN 10736 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is WARN
ERROR 10736 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is ERROR
```

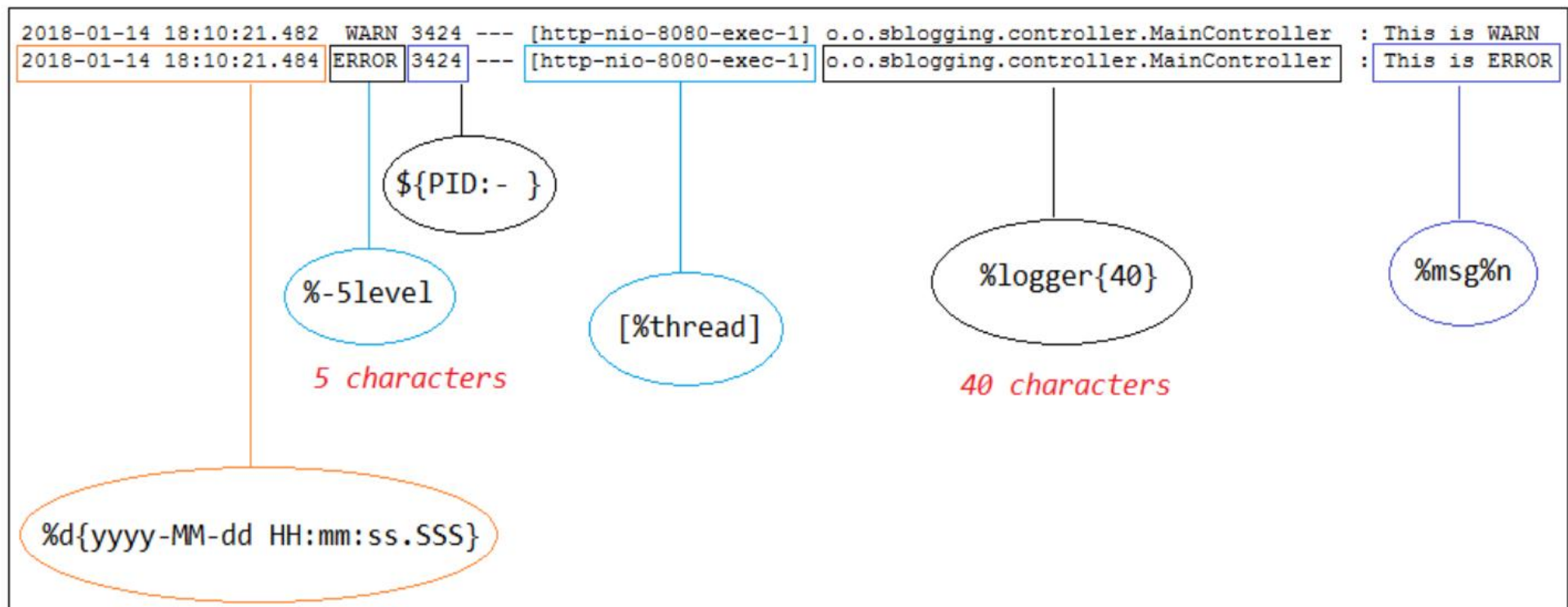
Logging File

- ❑ Cấu hình để ghi log ra file

```
# Logging
...
logging.file.name=mylog.log
...
```



- ❑ Các bản ghi Log (Log record) được ghi ra theo một mẫu (pattern), dưới đây là một mẫu (pattern) mặc định



- ❑ Thay đổi "**Logging pattern**" bằng cách tùy biến các thuộc tính (properties) dưới đây
 - ❖ **logging.pattern.console**
 - ❖ **logging.pattern.file**

Pattern:

```
logging.pattern.console= %d{yyyy-MMM-dd HH:mm:ss.SSS} %-5level [%thread] %logger{15} - %msg%n
```

Output:

```
2022-Jul-15 14:33:08.289 WARN [http-nio-8080-exec-1] c.l.r.c.FunRestController - This is WARN
```

```
2022-Jul-15 14:33:08.289 ERROR [http-nio-8080-exec-1] c.l.r.c.FunRestController - This is ERROR
```

Pattern:

```
logging.pattern.console=%d{yy-MMMM-dd HH:mm:ss:SSS} %5p %t %c{2}:%L - %m%n
```

Output:

```
22-July-15 14:34:37:483 WARN http-nio-8080-exec-1 c.l.r.c.FunRestController:20 - This is WARN
```

```
22-July-15 14:34:37:483 ERROR http-nio-8080-exec-1 c.l.r.c.FunRestController:21 - This is ERROR
```



Vấn Đề: Ứng dụng của chúng ta thường được phát triển với nhiều layer (lớp). Một ứng dụng Java điển hình sẽ có các layer như

- ❖ **Web Layer:** Nó hiển thị các service bằng cách sử dụng REST hoặc ứng dụng Web
- ❖ **Business Layer:** Nó thực hiện các logic nghiệp vụ của một ứng dụng
- ❖ **Data Layer:** Nó thực hiện các logic bền vững của ứng dụng

- ❑ Trách nhiệm của mỗi layer là khác nhau, nhưng có một số khía cạnh chung được áp dụng cho tất cả các layer là **Logging, Security, validation, caching**. Những khía cạnh chung này được gọi là những mối quan tâm xuyên suốt (**cross-cutting concern**)
 - ❑ Nếu thực hiện những mối quan tâm này trong từng layer khác nhau, đoạn code của chúng ta sẽ trở nên khó bảo trì hơn
 - ❑ Và để giải quyết vấn đề này, lập trình hướng khía cạnh (AOP) cung cấp cho ta một giải pháp để thực hiện các mối quan tâm xuyên suốt
-



Giải Pháp

- ❑ AOP (hay còn gọi là Aspect-Oriented Programming) là một mẫu lập trình làm tăng tính module bằng cách cho phép phân tách các mối quan tâm xuyên suốt
 - ❖ Thực hiện mối quan tâm xuyên suốt như một khía cạnh
 - ❖ Xác định các **pointcut** để chỉ ra nơi mà khía cạnh phải được áp dụng
 - ❖ Thêm các hành vi bổ sung vào đoạn code mà không cần phải chỉnh sửa tới đoạn code của mình
 - ❖ Mối quan tâm xuyên suốt giờ đây đã được module hóa thành các class đặc biệt và được gọi là **aspect**
 - ❑ Có 2 lợi ích của **aspect** đó là
 - ❖ Logic cho các mối quan tâm giờ đây ở một nơi
 - ❖ Các module nghiệp vụ chỉ chứa mã cho mối quan tâm chính của chúng. Mối quan tâm thứ cấp đã được chuyển sang cho **aspect**
 - ❑ Các **aspect** có trách nhiệm được thực hiện được gọi là **advice**. Chúng ta có thể triển khai chức năng của một **aspect** vào trong chương trình tại một hoặc nhiều điểm tham gia (**join point**)
-

- ❑ Chèn log khi chạy các service mà không sửa các method đó. Ví dụ có 1 method như thế này

```
public String callDaoSuccess(){  
    return "dao1";  
}
```

- ❑ Muốn chèn log khi method đó được gọi. Theo logic thông thường thì sẽ vào sửa method đó
 - 1) Phải sửa code trong method
 - 2) Nếu 1 Class có nhiều method mà muốn sửa thì phải sửa tất cả các method đó

```
public String callDaoSuccess(){  
    logger.info("callDaoSuccess is called");  
    return "dao1";  
}
```

- ❑ Hoặc tìm tất cả những chỗ nào method được gọi, insert log vào trước => Tốn thời gian nếu method được dùng nhiều chỗ

- ❑ Thêm thư viện AOP vào file pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

- ❑ Từ thư mục domain của project → tạo package **aspect**
- ❑ Trong package **aspect** → tạo class **DemoServiceAspect** như sau

```
@Configuration
@Aspect
public class DemoServiceAspect {
    private Logger logger = LoggerFactory.getLogger(DemoServiceAspect.class);

    @Before("execution(* com.likelion.rest.controller.*.*(..))")
    public void before(JoinPoint joinPoint){
        logger.info(" before called " + joinPoint.toString());
    }
}
```

- ❑ **@Aspect**: Chỉ ra rằng class này là 1 Aspect
- ❑ **@Before**: Chạy hàm này trước khi chạy hàm cần chèn
- ❑ **("execution(* com.likelion.rest.controller.*.*(..))")**: Là một regex để lựa chọn nơi sẽ áp dụng. Dấu * thứ nhất chỉ rằng bất kỳ class nào trong package controller. Dấu * thứ hai chỉ ra bất cứ method nào. Chúng ta có thể chỉ chính xác method

```

.controller.FunRestController.*.*(..))"
) {
inPoint.toString());
    equals()
    home()
    clone()
    hashCode()
    Object()
    toString()
    finalize()
    getClass()
    notify()
    notifyAll()
    registerNatives()
    wait()
    Press Enter to insert. Tab to replace. Next Tip

```

❑ Output

```

INFO 11500 --- [nio-8080-exec-1] c.l.rest.aspect.DemoServiceAspect : before called execution(String com.likelion.rest.controller.FunRestController.home())
INFO 11500 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is INFO
WARN 11500 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is WARN
ERROR 11500 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is ERROR

```


- ❑ **Pointcut:** Điểm cắt, dùng để khai báo rằng Aspect đó sẽ được gọi khi nào. Ở ví dụ trên ("execution(* com.likelion.rest.controller.*(..))") nó xảy ra ở tất cả các method trong class trong package com.likelion.rest.controller
 - ❑ **Advice:** Xử lý khi xảy ra điểm cắt đó. Advice là logic chúng ta muốn thực hiện, chính là đoạn code bên trong
 - ❑ **Join Point:** Khi code chạy và điều kiện pointcut đạt được, advice được chạy. Join Point là 1 instance của advice
 - ❑ Đây là định nghĩa khi nào code của advice được chạy
 - ❖ **@Before** : Chạy trước method
 - ❖ **@After**: Chạy trong 2 trường hợp method chạy thành công hay có exception
 - ❖ **@AfterReturning**: Chạy khi method chạy thành công
 - ❖ **@AfterThrowing**: Chạy khi method có exception
-


```
@Configuration
@Aspect
public class DemoServiceAspect {
    private Logger logger = LoggerFactory.getLogger(DemoServiceAspect.class);

    @Before("execution(* com.likelion.rest.controller.FunRestController.home())")
    public void beforeController(JoinPoint joinPoint) {
        logger.info(" before called " + joinPoint.toString());
    }

    @Before("execution(* com.likelion.rest.service.FunService.*(..))")
    public void beforeService(JoinPoint joinPoint) {
        logger.info(" before called service" + joinPoint.toString());
    }

    // package dao
    // ...

}
```

Output

```
2022-07-15 15:38:04.237 INFO 10700 --- [nio-8080-exec-1] c.l.rest.aspect.DemoServiceAspect : before called execution(String com.likelion.rest.controller.FunRestController.home())
2022-07-15 15:38:04.245 INFO 10700 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is INFO
2022-07-15 15:38:04.245 WARN 10700 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is WARN
2022-07-15 15:38:04.245 ERROR 10700 --- [nio-8080-exec-1] c.l.rest.controller.FunRestController : This is ERROR
2022-07-15 15:38:04.245 INFO 10700 --- [nio-8080-exec-1] c.l.rest.aspect.DemoServiceAspect : before called serviceexecution(void com.likelion.rest.service.FunService.aspectFunService())
```

- ❑ **Aspect:** Aspect(khía cạnh) là một module đóng gói những advice và các pointcut và cung cấp tính xuyên suốt. Một ứng dụng có thể có bất kỳ khía cạnh nào. Chúng ta có thể triển khai một khía cạnh bằng cách sử dụng class thông thường với annotation **@Aspect**
- ❑ **Pointcut:** pointcut là một biểu thức chọn một hoặc nhiều điểm nối(**join poin**) nơi **advice** được thực thi
- ❑ **Join point:** Join point là một điểm trong ứng dụng nơi chúng ta áp dụng khía cạnh AOP
- ❑ **Advice:** Advice là một hành động mà chúng ta thực hiện trước hoặc sau khi method thực hiện. Có 5 loại **advice** trong Spring AOP framework đó là: **before**, **after**, **after-returning**, **after-throwing**, **around advice**. Các advice được sử dụng cho một **join point** cụ thể
- ❑ **Target object:** Đối tượng mà được các advice sử dụng thì được gọi là target object. Target object thì luôn là một proxy. Nó có nghĩa là một lớp con được tạo ra tại thời điểm chạy, trong đó phương thức đích bị ghi đè và các advice được đưa vào dựa trên cấu hình của chúng
- ❑ **Weaving:** Nó là một quá trình liên kết các khía cạnh với các loại ứng dụng khác. Chúng ta có thể thực hiện nó tại các thời điểm như: **run time**, **load time**, và **compile time**
- ❑ **Proxy:** Là một đối tượng được tạo ra sau khi áp dụng **advice** cho target object

- ❑ Spring Boot có package **spring-boot-starter-validation** dùng cung cấp các validation API dưới dạng annotation để tiện sử dụng. Thêm vào file pom.xml

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- ❑ Trong Spring Boot, việc validation gồm 2 bước
 - ❖ Thêm các annotation ràng buộc trên các field của class nào đó. Mỗi annotation có ý nghĩa riêng, ví dụ **@NotNull**, **@NotEmpty**, **@Email**,...
 - ❖ Class đó được dùng ở vị trí nào cần đảm bảo hợp lệ, ví dụ truyền cho method làm tham số, mà tham số phải hợp lệ rồi mới thực hiện method, thì thêm **@Valid** hoặc **@Validated** trên tham số (thuộc class đó)
-

❑ Trong class Tutorial

```
@Data
public class Tutorial {

    @NotEmpty(message = "Thiếu title")
    private String title;

    @NotEmpty(message = "Thiếu description") private String description;
}
```

❑ Trong controller

```
@RestController
@RequestMapping("/fun")
public class FunRestController {
    @PostMapping("/getLogging")
    public String home(@RequestBody @Valid Tutorial tutorial) {
        return "Hi, show loggings in the console or file!";
    }
}
```


Postman

POST http://localhost:8080/fun/getLogging

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   ... "title": "",
3   ... "description": ""
4 }
```

Body Cookies Headers (4) Test Results Status: 400 Bad Request

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2022-07-15T08:58:17.741+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "path": "/fun/getLogging"
6 }
```

Console

```
codes [tutorial.description,description]; arguments []; default message [description]]; default message [Thiếu description]] ]
tutorial.description,description]; arguments []; default message [description]]; default message [Thiếu description]] ]
```


- ❑ Nếu dữ liệu không hợp lệ, thì method sẽ không được gọi. Lúc này là validation đã bị fail
- ❑ Dùng tham số cuối cùng là **BindingResult**. Nếu validation fail, method vẫn sẽ được gọi vào, và chúng ta có thể check tham số BindingResult kia có chứa lỗi hay không, từ đó xử lý phù hợp

```
@RestController
@RequestMapping("/fun")
public class FunRestController {
    @PostMapping("/getLogging")
    public String home(@RequestBody @Valid Tutorial tutorial
        , BindingResult bindingResult) {

        if (bindingResult.hasErrors()) {
            return "Exception ...";
        }
        ...
        return "Hi, show loggings in the console or file!";
    }
}
```

- ❑ Sử dụng **@Value** để **tiêm** giá trị từ file properties vào code Java
 - ❑ Có thể cấu hình ứng dụng Spring Boot trong file **application.properties**
 - ❑ Ví dụ: Server port, context path, actuator, security etc...
 - ❑ Trong Spring Boot file **YAML** (.yml) được dùng để làm file **config** (tương đương với file **.properties**)
 - ❑ **"Logging"** là "ghi chép" lại các vấn đề trong quá trình ứng dụng hoạt động
 - ❑ Mặc định Spring Boot đã tự động cấu hình và sử dụng thư viện **Logback** cho logging
 - ❑ Logging phân chia các thông tin cần ghi chép thành 7 mức độ khác nhau: **TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF**
 - ❑ Lập trình hướng khía cạnh (**AOP**) cung cấp cho ta một giải pháp để thực hiện các **mối quan tâm xuyên suốt**
 - ❑ Spring Boot có package **spring-boot-starter-validation** dùng cung cấp các validation **API** dưới dạng annotation
 - ❑ Thêm các annotation ràng buộc trên các field của class nào đó. Mỗi annotation có ý nghĩa riêng, ví dụ **@NotNull, @NotEmpty, @Email,...**
 - ❑ Class đó được dùng ở vị trí nào cần đảm bảo hợp lệ, thì thêm **@Valid** hoặc **@Validated** trên tham số (thuộc class đó)
-

Cảm Ơn Bạn Đã Chăm Chỉ!

