Spring Boot Validation

打造堅固的 ATM API 實戰教學

`@Valid` `@Validated` `Exception Handling`

現況分析：繁瑣的手動 if-else

AtmService.java 的痛點

為了確保資料正確，我們被迫撰寫大量的防禦性程式碼 (Defensive Programming)。

- **程式碼冗長**：真正的業務邏輯被驗證邏輯淹沒。
- **維護困難**：每個 Service 方法都要重複檢查 DTO。
- **可讀性差**：像一團糾結的毛線球。

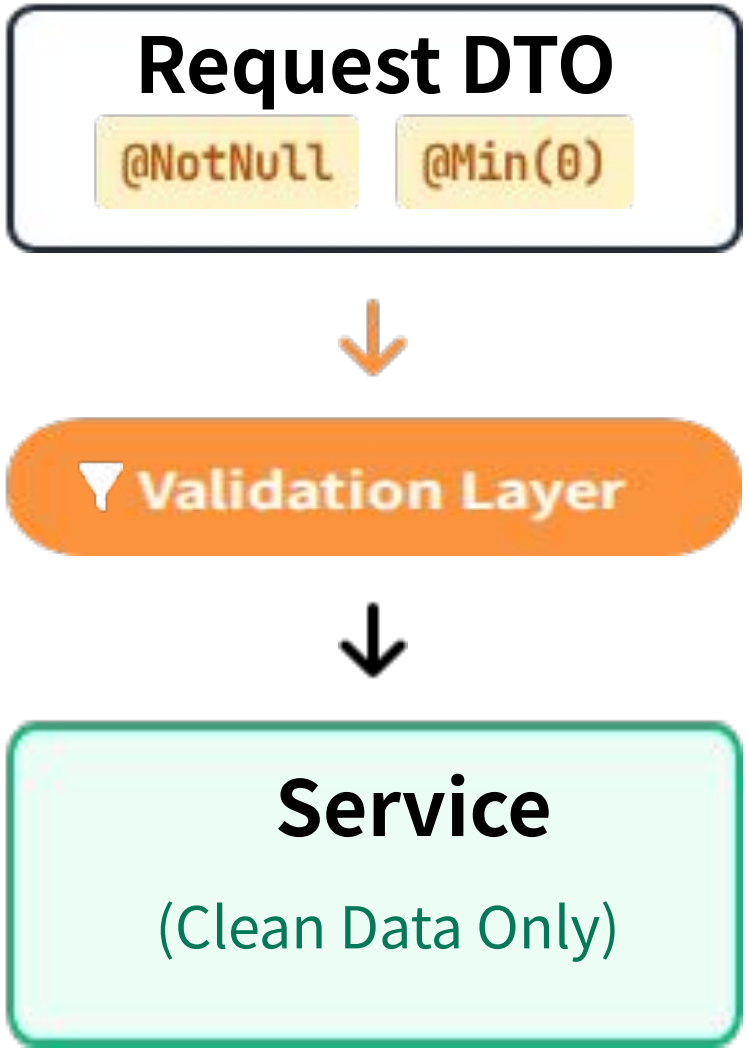
```
public void addInfo(AtmRequest req)
{
    // ❌ 驗證邏輯混雜在業務中
    if (req.getAccount() == null) {
        throw new Exception("帳號空");
    }
    if (req.getPassword().length() <
    8) {
        throw new Exception("密碼太短");
    }
    // ... 更多 if-else ...
    repo.save(req); // ✅ 真正的業務
}
```

解決方案：Bean Validation

宣告式驗證 (Declarative Validation)

將規則定義在 DTO 上，而非 Service 程式碼中。

- **核心思想：**在「進貨單」(Request DTO) 上直接貼標籤。
- **JSR-380 標準：**Java 官方定義的 Bean Validation 規範。
- **自動攔截：**Controller 收到資料前自動檢查，不合格直接擋在門外。



環境建置 - 加入依賴

工欲善其事，必先利其器

Spring Boot 2.3 版本後，Validation 模組已被移出核心，必須手動引入。

> ⚠注意
> 修改 build.gradle 後，記得點擊 Refresh (Reload) 專案。

```
// build.gradle
dependencies {
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
// 👇 加入這一行
implementation 'org.springframework.boot:spring-boot-starter-validation'
}
```

常用驗證註解一覽

這是我們打造 ATM 系統最常用的防護武器：

註解	適用型別	ATM 應用場景與說明
@NotNull	Any	欄位不能為 null。
@NotBlank	String	不能為 null 且至少包含一個非空白字元。 *常用於：帳號、密碼
@Length / @Size	String, Collection	限制字串長度或集合大小。 *常用於：帳號長度限制 3~8 碼
@Min / @Max	Number	數值範圍限制。 *常用於：存款金額（@Min(0)）
@Pattern	String	正規表達式（Regex）自定義驗證。 *常用於：密碼複雜度檢查

實作 Step 1：改造 DTO (帳號與密碼)

定義規則

- **帳號**：必填，長度 3~8 碼。
- **密碼**：必填，需符合 Regex 規則 (英數混合 8~16 碼)。
- **message**：自定義錯誤訊息，讓使用者看懂。

```
public class AtmRequest {
    @NotBlank(message = "帳號不能為空")
    @Length(min = 3, max = 8, message = "長度需
    3~8 碼")
    private String account;
    @NotBlank(message = "密碼不能為空")
    // 👇 Regex: 8-16位英數混合
    @Pattern(regexp = "^[a-zA-Z0-9_]{8,16}$",
    message = "格式不符")
    private String password;
}
```

實作 Step 2：改造 DTO (金額)

防止邏輯漏洞

ATM 最怕邏輯錯誤，例如存入「負數」金額導致餘額異常。

-500 → Bad Input

Blocked!

```java
public class AtmRequest {
    // ... 帳號密碼省略 ...
    @Min(value = 0, message = "金額不能為負數")
    private Integer amount;
}
```

進階技巧：巢狀驗證 (Nested Validation)

連鎖檢查機制

如果 DTO 裡面還包著另一個物件 (例如聯絡資訊)，預設是不會檢查內層的。

🔑關鍵鑰匙：@Valid
在外層屬性加上 @Valid，告訴 Spring 進入內層繼續檢查。

```
class AtmRequest {
    @Valid // 🔑 沒加這行，內層無效
    private ContactInfo contact;
}
```

Controller 整合：啟用 @Valid

插上電源，讓規則生效

只在 DTO 貼標籤是不夠的，必須在 Controller 門口放置警衛。

- **位置**：Controller 方法參數前。
- **動作**：加上 @Valid (或 @Validated)。
- **效果**：觸發 Spring MVC 驗證攔截器。

```
@RestController
public class AtmController {
    @PostMapping("/add")
    public AtmResponse add(
    // 👇 重點在這！
    @Valid @RequestBody
    AtmRequest request ) {
        return
    atmService.addInfo(request);
    }
}
```

觀察預設錯誤：看不懂的 400 Bad Request

使用者體驗災難

雖然成功擋住了錯誤資料，但 Spring Boot 預設回傳的錯誤格式包含了太多技術細節。

- 包含了 Trace Stack。
- 包含了 Timestamp。
- 前端難以解析 (Parse)。

HTTP 400 Bad Request

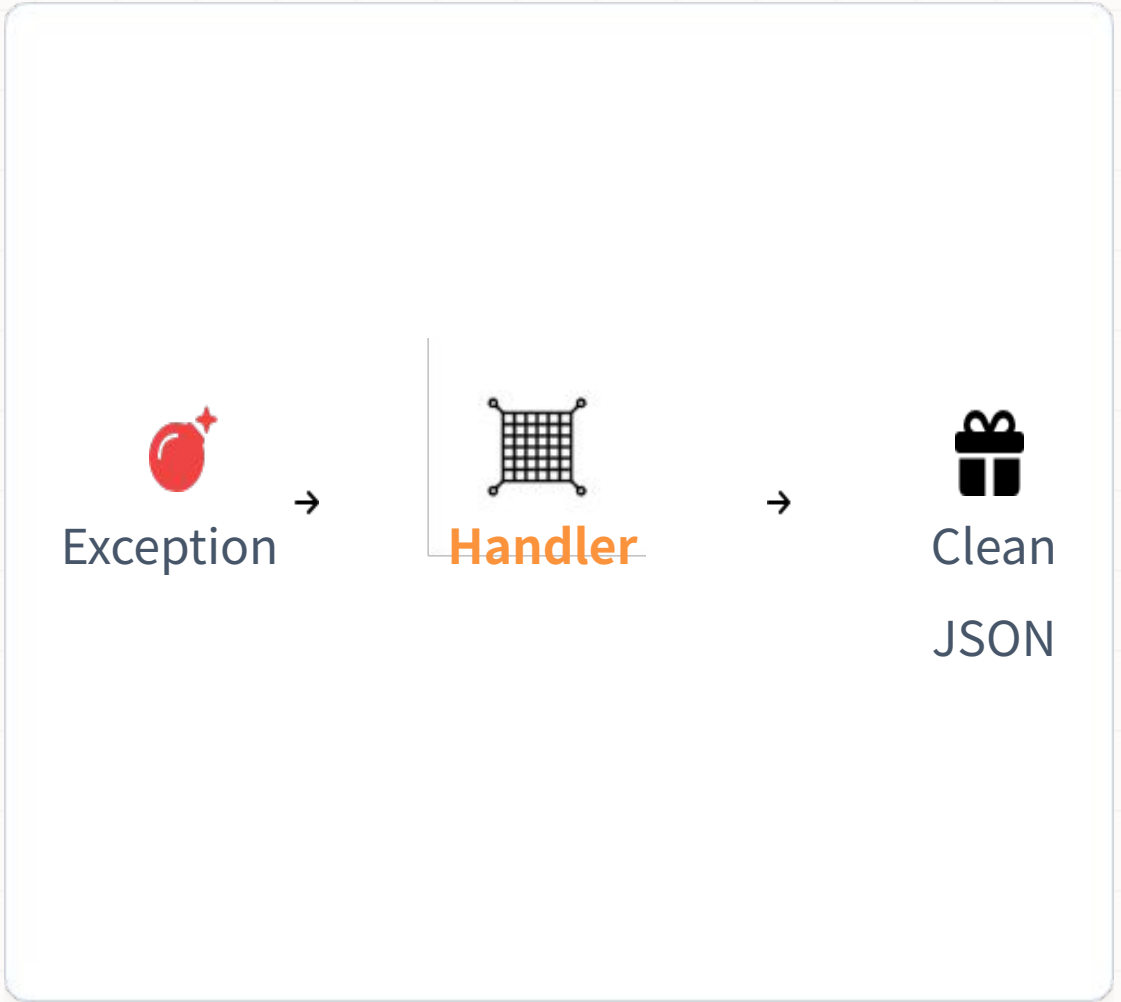
```
{
  "timestamp":
"2023-10-27T10:00:00.000+00:00",
  "status": 400,
  "error": "Bad Request",
  "trace": "org.springframework.web.bind...",
  "message": "Validation failed for object...",
  "path": "/add"
}
```

解決方案：全域異常處理

建立「防爆小組」

我們不讓 Controller 直接拋出 Exception，而是由專門的類別接手處理。

- **@RestControllerAdvice**：宣告這是一個全域攔截器。
- **@ExceptionHandler**：指定要捕捉 MethodArgumentNotValidException



實作：撰寫錯誤攔截邏輯

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, Object>> handle(
MethodArgumentNotValidException e ) {
        Map<String, Object> res = new HashMap<>();
        res.put("code", 400); // 👇 關鍵：抓出 DTO 裡寫的 message
        res.put("message",
    e.getBindingResult().getAllErrors().get(0).getDefaultMessage());
        return ResponseEntity.badRequest().body(res);
    }
}
```

精準抓取我們在 DTO 設定的 message="帳號不能為空"。

最終成果：乾淨的 JSON 回應

使用者體驗提升

現在錯誤訊息變成了我們自己定義的格式，前端開發者可以輕鬆處理。

- 結構簡單。
- 訊息明確。
- 無多餘雜訊。

Postman 400 Bad Request


```
{
    "code": 400,
    "message": "密碼格式不符"
 }
```

挑戰情境：新增 vs 存款 (分組驗證)

同一個 DTO，不同的標準

AtmRequest 同時用於「新增帳號」與「存款」，但驗證需求不同。

- **新增帳號 (Create)**：金額可以是 null (預設 0)。
- **存款交易 (Transaction)**：金額必須大於 0。

使用 **Groups (分組)** 標籤介面來區隔。

CreateGroup
Skip Amount Check

TransGroup
Check @Min(0)

實作：配置 @Validated 分組

```
// 1.1 建立不同的Group

public interface CreateGroup {}
```

```
// 1.2 建立不同的Group

public interface TransactionGroup {}
```

```
// 2. DTO 設定

class AtmRequest {
@NotBlank(groups =
CreateGroup.class)
private String account;
@Min(value=0, groups =
TransactionGroup.class)
private Integer amount; }
```

```
// 3. Controller 設定

@PostMapping("/deposit")
public AtmResponse deposit(
// 👇 指定要檢查 TransactionGroup
@Validated(TransactionGroup.class)
@RequestBody AtmRequest request )
{ ... }
```

在 Controller 呼叫時，指定這次要檢查哪一組，靈活適應需求。