LẬP TRÌNH JAVA SPRING BOOT

Bài 7: Exception, Interceptor, Filters, Spring Boot Tests, Actuator

- Kết thúc bài học này bạn có khả năng
  - Xử lý Exception phát sinh trong ứng dụng Spring Boot
  - Hiểu về Interceptor, Filters
  - Testing Trong Spring Boot, JUnit 5
  - Giám sát ứng dụng với Spring Boot Actuator

- Cách bắt exception phổ biến là dùng try catch

```
try {
    // Do something
} catch (Exception e) {
    // Xử lý lỗi
}
```

Nếu dùng try catch mọi nơi có exception, thì việc quản lý và bảo trì sẽ phức tạp

Giải Pháp

- Cũng giống như Logging, chúng ta sẽ ứng dụng AOP và kết hợp với **@ControllerAdvice** và **@ExceptionHandler** để giải quyết vấn đề trên
- Giải pháp trên giúp code dễ hiểu, code gọn và gom lại một chỗ dễ quản lý và bảo trì

- Spring Boot sử dụng hai annotation **@ControllerAdvice** và **@ExceptionHandler** bên trong để thực hiện bắt mọi exception xuất hiện trong ứng dụng

```
@RestControllerAdvice
public class GlobalExceptionHandler {

   @ExceptionHandler({ NullPointerException.class, ClassNotFoundException.class })
   // Có thể bắt nhiều loại exception
   public ResponseEntity<String> handleExceptionA(Exception e) {
      return ResponseEntity.status(432).body(e.getMessage());
   }

   // Nên bắt cả Exception.class
   @ExceptionHandler(Exception.class)
   public ResponseEntity<String> handleUnwantedException(Exception e) {
      // Log lỗi ra và ẩn đi message thực sự
      e.printStackTrace();  // Thực tế người ta dùng logger
      return ResponseEntity.status(500).body("Usa");
   }
}
```

- Nếu có exception ở bất cứ đâu, sẽ được chuyển tới method có **@ExceptionHandler** tương ứng
- Nếu không tìm thấy method tương ứng thì chuyển lên exception class cha (do đó, nên có một **@ExceptionHandler** để bắt **Exception** class, dành cho các exception còn lại)
- Các method này viết tương ứng với method của Controller, nhưng thay vì trả data về thì chúng ta trả về message lỗi
- **Lưu ý**:
  - Ta có **@RestControler** = **@Controller** + **@ResponseBody**
  - Nên có thể hiểu: **@RestControllerAdvice** = **@ControllerAdvice** + **@ResponseBody**
- **@RestControllerAdvice** hiểu đơn giản là chú thích tiện lợi hơn khi bắt Exception cho Restful API

I. Tạo các exception class tùy chỉnh

- Mặc dù Spring Boot có các class exception có sẵn như **IllegalAccessException**,... chúng ta vẫn nên tạo cac class exception của **riêng mình**. Điều này giúp ta phân biệt giữa
  - **System exception**: Do hệ thống, các framework, thư viện ném ra
  - **Custom exception**: Do code mình viết ném ra. Ví dụ user không tồn tại, request quá nhiều,...

```
@Getter @AllArgsConstructor
public class AppException extends RuntimeException {
    private int code;
    private String message;
}
```

- Và thực hiện ném Exception đó trong ngữ cảnh thích hợp

```
public class UserService {
    public User getUser(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) throw new AppException(404, "User not found");
        ...
    }
}
```

II. Che giấu lỗi hệ thống

Nguyên tắc quan trọng khi xử lý exception phía backend là

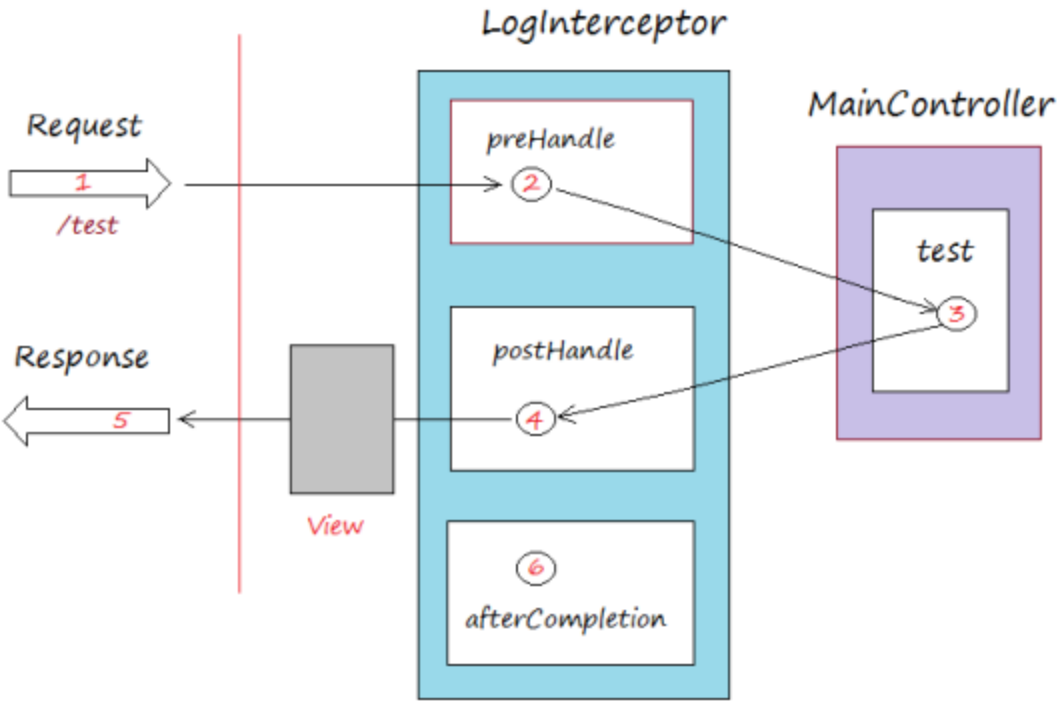
- Không bao giờ được trả về **chi tiết** lỗi (error details) cho client
- Do đó, dễ thấy được lợi ích của việc chia hai loại exception ở phần trên
  - **Với system exception**: cần **ẩn** message đi, **không** return về cho client (mà return về một message khác)
  - **Với custom exception**: do viết code ném ra, nên message có thể return về cho client được
- Ngoài ra, với system exception cần thực hiện log ra đâu đó, để biết và sửa lỗi. Ví dụ log ra console, hoặc ra file, để nhanh chóng phát hiện và xử lý

Hình dung về Interceptor?

- Khi bạn tới công ty và muốn gặp sếp của công ty đó. Bạn cần phải đi qua các **chốt chặn** (Interceptor), các chốt chặn ở đây có thể là người bảo vệ cổng, nhân viên lễ tân,..
- Trong Spring, khi một request được gửi đến controller, trước khi request được xử lý bởi Controller, nó phải vượt qua các **Interceptor** (0 hoặc nhiều)
- **Spring Interceptor** là một khái niệm khá giống với **Servlet Filter**
- **Spring Interceptor** chỉ áp dụng đối với các request đang được gửi đến một Controller

- Bạn có thể sử dụng Interceptor trong Spring Boot để làm một số việc như
  - Trước khi gửi request tới Controller
  - Trước khi gửi response tới Client
  - Ghi lại Log
- Lớp Interceptor của bạn cần phải thực hiện interface **HandlerInterceptor** hoặc mở rộng từ lớp **HandlerInterceptorAdapter**
- Để làm việc với Interceptor, bạn cần tạo class **@Component** và được implement từ interface **HandlerInterceptor**
- Sau đây là ba method bạn nên biết khi làm việc với Interceptor
  - **preHandle**() method: Thực hiện các hoạt động trước khi gửi request tới Controller. Method này trả về true rồi trả response cho Client
  - **postHandle**() method: method này sử dụng để thực hiện các hoạt động trước khi gửi request tới Client
  - **afterCompletion**() method: method này được sử dụng để thực hiện các hoạt động sau khi hoàn thành việc gửi request và response

- Luồng đi khi có Interceptor



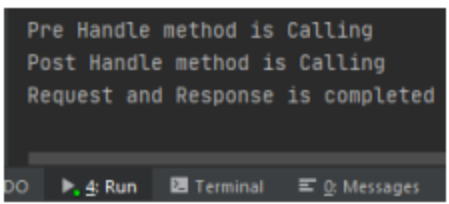
- Quan sát đoạn code dưới đây để hiểu rõ hơn

```
@Component
public class ProductServiceInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle( HttpServletRequest request
            , HttpServletResponse response, Object handler) throws Exception {
        System.out.println("Pre Handle method is Calling");
        return true;
    }

    @Override
    public void postHandle( HttpServletRequest request,
    HttpServletResponse response, Object handler
            , ModelAndView modelAndView) throws Exception {
        System.out.println("Post Handle method is Calling");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler
            , Exception exception) throws Exception {
        System.out.println("Request and Response is completed");
    }
}
```

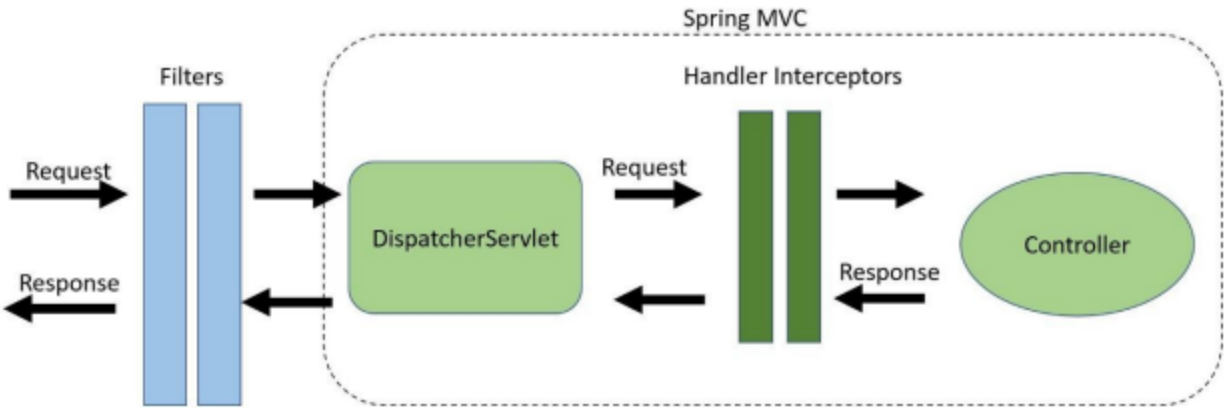
- Bạn sẽ phải đăng ký Interceptor này với **InterceptorRegistry** bằng cách sử dụng **WebMvcConfigurerAdapter** như hình dưới đây

```
@Component
public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter {
    @Autowired
    ProductServiceInterceptor productServiceInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry)    {
        // Áp dụng cho mọi URL
        registry.addInterceptor(productServiceInterceptor);

        // Áp dụng cho URL cụ thể ví dụ: /admin/*
        registry.addInterceptor(productServiceInterceptor)
                .addPathPatterns("/admin/*");
    }
}
```

- Output



- Một **Filter** là một đối tượng được sử dụng để chặn các yêu cầu HTTP và phản hồi của ứng dụng của bạn
- Bằng cách sử dụng filter, chúng ta có thể thực hiện hai hoạt động tại hai trường hợp sau
  - Trước khi gửi request tới controller
  - Trước khi gửi response tới client



- Để sử dụng filter tạo lớp triển khai **Servlet Filter** với chú thích **@Component**

Code Ví Dụ Servlet Filter

- Ví dụ sau cho thấy mã để đọc máy chủ từ xa và địa chỉ từ xa từ đối tượng **ServletRequest** trước khi gửi yêu cầu đến bộ điều khiển
- Trong phương thức **doFilter**(), chúng tôi đã thêm các câu lệnh **System.out.println** để print ra **remote host** và **remote address**

```
@Component
public class SimpleFilter implements Filter {
    @Override
    public void destroy() {}

    @Override
    public void doFilter(ServletRequest request, ServletResponse response
            , FilterChain filterchain) throws IOException, ServletException {
        System.out.println("Remote Host:"+request.getRemoteHost());
        System.out.println("Remote Address:"+request.getRemoteAddr());
        filterchain.doFilter(request, response);
}
    @Override
    public void init(FilterConfig filterconfig) throws ServletException {}
}
```

- Output

```
Remote Host:0:0:0:0:0:0:0:1
Remote Address:0:0:0:0:0:0:0:1
```

- Viết Test là một phần quan trọng trong việc xây dựng tất cả ứng dụng chứ không riêng gì **Spring Boot**

Sơ lược về JUnit

- **JUnit** là một Java testing framework được sử dụng phổ biến trong các dự án Java
- Cung cấp các annotation để định nghĩa các phương thức kiểm thử
- Cung cấp các **Assertion** để kiểm tra kết quả mong đợi
- Test case JUnit có thể được chạy **tự động**
- JUnit cho thấy kết quả test một cách **trực quan**: pass (không có lỗi) là màu xanh và fail (có lỗi) là màu đỏ
- JUnit 5 là phiên bản mới nhất, gồm có 3 module từ 3 sub-project
  - JUnit Platform: layer nền tảng cho phép những Testing Framework có thể chạy trên máy ảo JVM
  - JUnit Jupiter: là phiên bản JUnit 5 và được chạy trên JUnit Platform
  - JUnit Vintage: kế thừa từ TestEngine và cho phép chạy các phiên bản cũ hơn của JUnit

Một số khái niệm cần biết trong Unit Test

- **Unit Test case**: là 1 chuỗi code để đảm bảo rằng đoạn code được kiểm thử làm việc như mong đợi. Mỗi function sẽ có nhiều test case, ứng với mỗi trường hợp function chạy
- **Setup**: Đây là hàm được chạy trước khi chạy các test case, thường dùng để chuẩn bị dữ liệu để chạy test
- **Teardown**: Đây là hàm được chạy sau khi các test case chạy xong, thường dùng để xóa dữ liệu, giải phóng bộ nhớ.
- **Assert**: Mỗi test case sẽ có một hoặc nhiều câu lệnh Assert, để kiểm tra tính đúng đắn của hàm
- **Mock**: là một đối tượng ảo, mô phỏng các tính chất và hành vi giống hệt như đối tượng thực được truyền vào bên trong
- **Test Suite**: Test suite là một tập các test case và nó cũng có thể bao gồm nhiều test suite khác, test suite chính là tổ hợp các test

Vấn đề Test + Spring

- Spring Boot phải tìm kiếm các Bean và đưa vào Context quản lý. Sau tất cả các bước config và khởi tạo thì chúng ta sử dụng **@Autowired** để lấy đối tượng ra sử dụng
- Vấn đề đầu tiên bạn nghĩ tới khi viết Test sẽ là làm sao @Autowired bean vào class Test được và làm sao cho **JUnit** hiểu **@Autowired** là gì

Giải pháp

- Với **JUnit** có version nhỏ hơn 5 dùng **@RunWith**(**SpringRunner**.class), sẽ giúp chúng ta tích hợp Spring + Junit. Trong mọi class Test chúng ta sẽ thêm lên trên Class Test đó
- Với **JUnit 5** sử dụng **@ExtendWith**(**SpringExtension**.class)

```
@RunWith(SpringRunner.class)
public class TodoServiceTest {
        ...
}
```

- Khi bạn chạy **TodoServiceTest** nó sẽ tạo ra một **Context** riêng để chứa **bean** trong đó, từ đó có thể **@Autowired** trong nội hàm Class này
- Vấn đề tiếp theo là làm sao đưa **Bean** vào trong **Context**
- Có 2 cách

1. **@SpringBootTest**
2. **@TestConfiguration**

- **@SpringBootTest** sẽ đi tìm kiếm class có gắn **@SpringBootApplication** và từ đó đi tìm toàn bộ **Bean** và nạp vào **Context**
- **Lưu ý**: Nên sử dụng trong trường hợp muốn Integration Tests, vì nó sẽ tạo toàn bộ Bean, giống khi chạy SpringApplication.run(App.class, args);, tốn nhiều thời gian và có nhiều Bean không cần thiết

- Với **JUnit 5** chỉ cần **@SpringBootTest** sẽ bao gồm cả **@ExtendWith**(MockitoExtension.class)
- Ví dụ

```
@SpringBootTest
@DisplayName("Main App Tests")
class MybatisApplicationTests {

  @Autowired
  private BookService bookService;

  @Test
  @DisplayName("Test add book")
  public void whenApplicationStarts_thenHibernateCreatesInitialRecords() {
    Book save = bookService.addBook(new Book("test"));
    assertThat(save).isNotNull();
  }
}
```

- Trong API JUnit 5 cung cấp các static method mà chúng ta có thể sử dụng để viết các assertion
- Cần phải sử dụng class org.junit.jupiter.api.Assertions
- Các method như
  - assertEquals(): dùng để xác minh giá trị mong đợi và giá trị thực tế bằng nhau
  - assertNotEquals(): dùng để xác minh giá trị mong đợi và giá trị thực tế không bằng nhau
  - assertNull(): khẳng định rằng một object là null
  - assertNotnull(): khẳng định rằng object là not null

```
@Test void assertEqualsExample() {
     //Test will pass
     assertEquals(calculator.sum(2, 2), 5);

     //Test will failed
     assertNotEquals(calculator.sum(1,1),2);
}
```

```
@Test void assertNull_assertNotNull(){
    String nullString = null;
    String notNullString = "Techmaster";

    //Test will pass
    assertNull(nullString);
    assertNotNull(notNullString);
}
```

- *Truy cập vào để xem thêm nhiều assertion*
- *Source*: https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

- Giới thiệu AssertJ là một thư viện để đơn giản hóa việc viết các assertions. Nó cũng cải thiện khả năng đọc các câu lệnh assertion
- AssertJ có fluent interface cho assertions, giúp bạn dễ dàng viết code. Method cơ sở cho các AssertJ là assertThat
- Để viết assertions, cần bắt đầu bằng cách chuyển object của mình tới phương thức assertThat() và thực hiện theo các assertions thực tế
- Ví dụ

```
@Test
@DisplayName("TestAssertJ")
void sampleAssertion(){
   assertThat("This is my sample Test").isNotNull()
      .startsWith("This")
      .contains("sample")
      .endsWith("Test");
}
```

- *Truy cập vào để xem thêm AssertJ*
- *Source*: https://assertj.github.io/doc/

- Thêm các dependency trong file pom.xml như sau

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <!-- exclude junit 4 -->
  <exclusions>
    <exclusion>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<!-- junit 5 -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-api</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <scope>test</scope>
</dependency>
```

- Unit testing ở tầng DAO / Repository
- Ví dụ kết hợp giữa **@DataJpaTest** và h2-database (Memory database)

```
@DataJpaTest // load các Bean có liên quan tới tầng Repository
class BookRepositoryTest {
    @Autowired
    private BookRepository bookRepository;

    @AfterEach // sau mỗi testcase sẽ thực hiện
    void tearDown() {
        bookRepository.deleteAll();
    }

    @Test // testcase
    void getListBook() {
        Book afterSaved = bookRepository.findById(1l).get();
        assertThat(afterSaved).isNotNull(); // kết quả mong đợi
    }
}
```

- Unit testing ở tầng service
- Ví dụ

```
@ExtendWith(MockitoExtension.class) // sử dụng Mockito vào junit test
class BookServiceImplTest {
    @Mock // tạo giả một bean dùng khi là interface
    private BookRepository bookRepository;
    @InjectMocks // tạo giả một bean dùng khi là class
    private BookServiceImpl bookService;
    @Test // testcase
    void list() {
        bookService.list();
        verify(bookRepository).findAll();
    }
    @Test //test case
    void canAddStudent() {
        // given
        Book book = new Book(
                "Clean Code"
        );
        // when
        bookService.addBook(book);
        // then
        ArgumentCaptor<Book> bookArgumentCaptor =
                ArgumentCaptor.forClass(Book.class);

        verify(bookRepository)
                .save(bookArgumentCaptor.capture());
        Book capturedBook = bookArgumentCaptor.getValue();
        assertThat(capturedBook).isEqualTo(book);
    }
}
```

- Unit testing ở tầng controller
- Ví dụ

```
@ExtendWith(SpringExtension.class) // cần tạo context vì trong controller có bean BookService
@WebMvcTest(BookController.class) // Cung cấp lớp Controller cho @WebMvcTest
class BookControllerTest {
   /**
    * Đối tượng MockMvc do Spring cung cấp
    * Có tác dụng giả lập request, thay thế việc khởi động Server
    */
   @Autowired
   private MockMvc mockMvc;

   @MockBean // tạo bean giả
        private BookService bookService;

   @Test
   void getTheBook() throws Exception {
      this.mockMvc.perform(get("/book/")).andDo(print()).andExpect(status().isOk());
   }
}
```

- Khi muốn test tích hợp hệ thống gồm nhiều bước hoặc logic hệ thống
- Ví dụ

```
@SpringBootTest
@DisplayName("Main App Tests")
class MybatisApplicationTests {

  @Autowired
  private BookRepository bookRepository;

  @Autowired
  private BookService bookService;

  @Test
  @DisplayName("Test add book")
  public void whenApplicationStarts_thenHibernateCreatesInitialRecords() {
    Book save = bookService.addBook(new Book("test"));
    assertThat(save).isNotNull();
  }

  @Test
  @DisplayName("Test get all list")
  void getlist() {
    List<Book> books = bookService.list();
    assertEquals(4, books.size());
  }
}
```

- Spring hỗ trợ mock với annotation **@MockBean**, chúng ta có thể mock lấy ra một Bean "giả"

```
@RunWith(SpringRunner.class)
public class TodoServiceTest2 {

    /**
    * Đối tượng TodoRepository sẽ được mock, chứ không phải bean trong context
    */

    @MockBean
    TodoRepository todoRepository;
}
```

- Demo với **@MockBean**

```
@Service
public class TodoService {

    @Autowired
    private TodoRepository todoRepository;

    public int countTodo(){
       return todoRepository.findAll().size();
    }

    public Todo getTodo(int id){
       return todoRepository.findById(id);
    }

    public List<Todo> getAll(){
       return todoRepository.findAll();
    }

}
```

```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class TodoServiceTest {

    /** * Đối tượng TodoRepository sẽ được mock, chứ không phải bean trong context */
    @MockBean
    TodoRepository todoRepository;

    @Autowired
    private TodoService todoService;

    @Before
    public void setUp() {
       Mockito.when(todoRepository.findAll())
         .thenReturn(IntStream.range(0, 10)
          .mapToObj(i -> new Todo(i, "title-" + i, "detail-" + i))
           .collect(Collectors.toList())); // giả lập trả ra một List<Todo> 10 phần tử
    }

    @Test
    public void testCount() {
      Assert.assertEquals(10, todoService.countTodo());
    }
}
```

- **@WebMvcTest** là annotation Spring Boot hỗ trợ test Controller mà không cần khởi động Tomcat Server
- Bây giờ, khi muốn test một Controller nào đó, chúng ta làm như sau

```
@RunWith(SpringRunner.class)
// Bạn cần cung cấp lớp Controller cho @WebMvcTest
@WebMvcTest(TodoRestController.class)
public class TodoRestControllerTest {

    /**
    * Đối tượng MockMvc do Spring cung cấp
    * Có tác dụng giả lập request
    * Thay thế việc khởi động Server
    */

    @Autowired
    private MockMvc mvc;

}
```

- Demo với **@WebMvcTest**
- Tạo Controller

```
@RestController
@RequestMapping("/api/v1")
public class TodoRestController {

     @Autowired
     TodoService todoService;

     @GetMapping("/todo")
     public List<Todo> findAll(){
        return todoService.getAll();
     }

}
```

```java
@RunWith(SpringRunner.class)
// Bạn cần cung cấp lớp Controller cho @WebMvcTest
@WebMvcTest(TodoRestController.class)
public class TodoRestControllerTest {
    /**
     * Đối tượng MockMvc do Spring cung cấp * Có tác dụng giả lập request
     */
    @Autowired
    private MockMvc mvc;

    @MockBean
    private TodoService todoService;

    @Test
    public void testFindAll() throws Exception {
       // Tạo ra một List<Todo> 10 phần tử
       List<Todo> allTodos = IntStream.range(0, 10)
                                .mapToObj(i -> new Todo(i, "title-" + i, "detail-" + i))
                                .collect(Collectors.toList());

       // giả lập todoService trả về List mong muốn
       given(todoService.getAll()).willReturn(allTodos);
       mvc.perform(get("/api/v1/todo").contentType(MediaType.APPLICATION_JSON)) // GET REQUEST
          .andExpect(status().isOk()) // Mong muốn Server trả về status 200
          .andExpect(jsonPath("$", hasSize(10))) // Hi vọng server trả về List độ dài 10
          .andExpect(jsonPath("$[0].id", is(0))) // Hi vọng phần tử trả về đầu tiên có id = 0
          .andExpect(jsonPath("$[0].title", is("title-0")))// phần tử trả về đầu tiên có title "title0"
          .andExpect(jsonPath("$[0].detail", is("detail-0")));
    }
}
```

- **Spring Boot Actuator** được xây dựng để thu thập, giám sát các thông tin về ứng dụng
- Để giám sát ứng dụng bạn cần truy cập vào các **endpoint** (Điểm cuối) được xây dựng sẵn của Spring Boot Actuator, đồng thời bạn cũng có thể tạo ra các endpoint của riêng của bạn nếu muốn
- Tất cả mọi thứ bạn cần làm là khai báo các thư viện Spring Boot Actuator (pom.xml) và một vài thông tin cấu hình trong tập tin **application.properties**

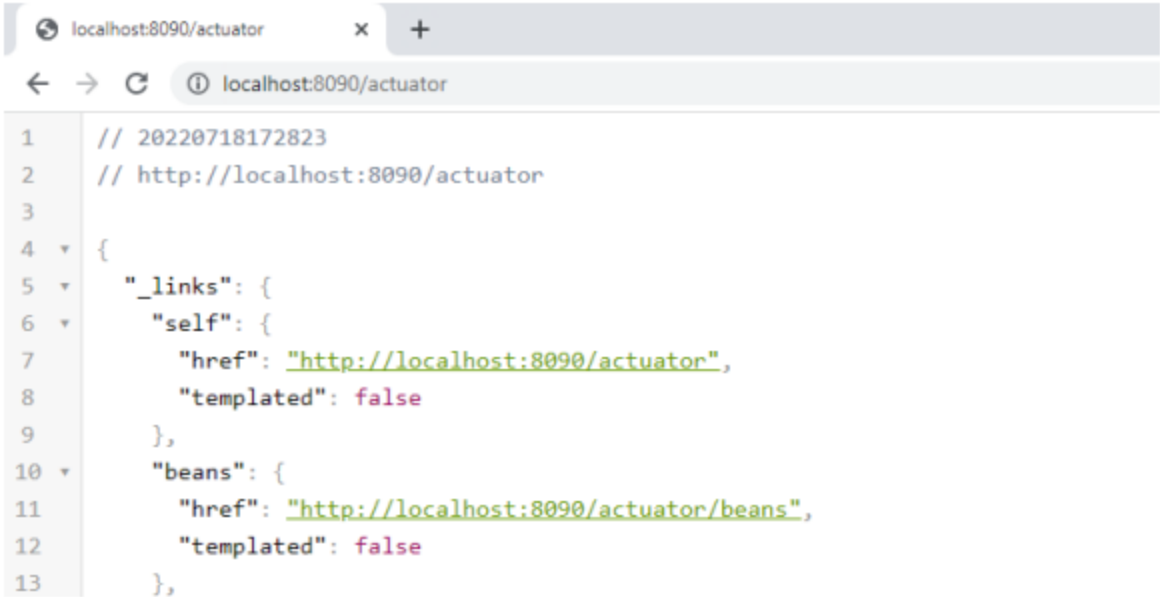
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- Thêm một vài đoạn cấu hình vào tập tin **application.properties**

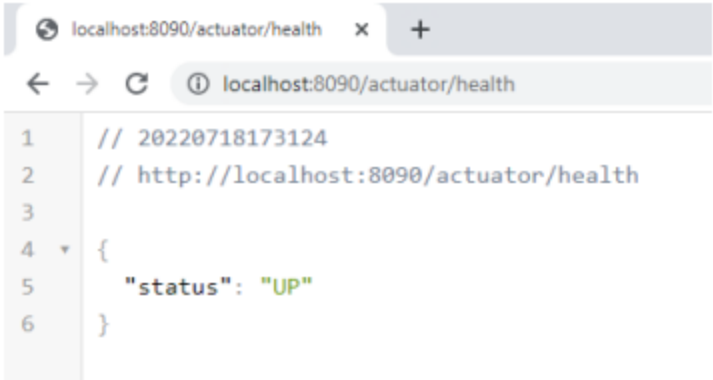
```
server.port=8080
management.server.port=8090
management.endpoints.web.exposure.include=*
management.endpoint.shutdown.enabled=true
```

- **server.port=8080**
  - Ứng dụng của bạn sẽ chạy trên cổng 8080 (port 8080), đây là cổng mặc định và bạn có thể thay đổi nó nếu muốn
- **management.server.port=8090**
  - Các endpoint liên quan tới việc giám sát của Spring Boot Actuator sẽ được truy cập theo cổng khác với cổng 8080 ở trên, mục đích của việc này là tránh nhầm lẫn và tăng cường bảo mật. Tuy nhiên điều này không bắt buộc.
- **management.endpoints.web.exposure.include=***
  - Mặc định không phải tất cả các Endpoint của Spring Boot Actuator đều được kích hoạt. Sử dụng dấu * để kích hoạt hết tất cả các Endpoint này
- **management.endpoint.shutdown.enabled=true**
  - **shutdown** là một **Endpoint** của Spring Boot Actuator, nó cho phép bạn tắt (shutdown) ứng dụng một các an toàn mà không cần sử dụng các lệnh như "Kill process", "end task" của hệ điều hành

- **/actuator** là một endpoint (Điểm cuối), nó cung cấp danh sách các endpoint khác của Spring Boot Actuator mà bạn có thể truy cập vào được



- **/actuator/health** cung cấp cho bạn thông tin về sức khỏe của ứng dụng. Trạng thái còn sống (UP) hoặc đã chết (DOWN), và các thông tin khác về ổ cứng như kích thước ổ cứng, dung lượng đã sử dụng và dung lượng chưa sử dụng



- **/actuator/info** cung cấp một thông tin tùy biến của bạn. Mặc định nó là một thông tin rỗng. Vì vậy bạn cần phải tạo một Spring BEAN để cung cấp thông tin này
- **/actuator/shutdown** endpoint giúp bạn tắt (shutdown) ứng dụng. Gọi nó với phương thức POST, nếu thành công sẽ được một text "Shutting down, bye..."

- Spring Boot sử dụng hai annotation **@ControllerAdvice** và **@ExceptionHandler** bên trong để thực hiện bắt mọi exception xuất hiện trong ứng dụng
- **System exception**: Do hệ thống, các framework, thư viện ném ra
- **Custom exception**: Do code mình viết ném ra
- Spring **Interceptor** chỉ áp dụng đối với các request đang được gửi đến một Controller
- Ba method bạn nên biết khi làm việc với Interceptor: **preHandle**(), **postHandle**(), **afterCompletion**()
- **Filter** là một đối tượng được sử dụng để chặn các yêu cầu HTTP và phản hồi của ứng dụng của bạn
- Spring Boot đã thiết kế ra lớp SpringRunner, sẽ giúp chúng ta tích hợp **Spring + Junit**
- Spring hỗ trợ mock với annotation **@MockBean**, chúng ta có thể mock lấy ra một Bean "giả"
- **@WebMvcTest** là annotation Spring Boot hỗ trợ test Controller mà không cần khởi động Tomcat Server
- Spring Boot Actuator đươc xây dựng để thu thập, giám sát các thông tin về ứng dụng, truy cập vào /actuator/health, /actuator/info, /actuator/shutdown...

Cảm Ơn Bạn Đã Chăm Chỉ!