I,

```@Controller``` đánh dấu một class thành component, và chuyên xử lý các request gọi đến

- ```@GetMapping``` đánh dấu một phương thức sẽ hứng (handle), xử lý (process) các yêu cầu (request) kiểu [GET]

II,

 @GetMapping(value="/json", produces=MediaType.APPLICATION\_JSON\_VALUE)

```MediaType.APPLICATION\_JSON\_VALUE``` thực chất là chuỗi ```application/json```

  public static final String APPLICATION\_JSON\_VALUE = "application/json";

  ```

Phương thức trả về chỉ nhận kiểu String, nếu muốn trả về một mảng, một đối tượng chúng ta cần chuyển đổi (convert) chúng sang JSON string. Thư viện [thư viện jackson-databind](https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind) sẽ làm nhiệm vụ chuyển đổi.

Tạo class Car ở model/Car.java. Đoạn này hãy sử dụng SourceAction để sinh getter, setter và constructor

III,

**application.properties và application.yml:file cấu hình**

giá trị của application.yml có thể ghi đè lên giá trị của application.properties

IV,

Bản chất của View Template Engine là cơ chế ghép dữ liệu (data) với khuôn mẫu (template).

@GetMapping(value = "/")

  public String getHome(Model model) {

    model.addAttribute(APP\_NAME, appName);

    return "home";

}

```

Giờ phía trên phương thức getHome không còn ```@ResponseBody``` nữa. Phương thức này không trả về dữ liệu ngay, mà nó đổ dữ liệu vào đối tượng ```Model model``` rồi trả về Thymeleaf view template là ```home.html```

```<img th:attr```

th:attr cho phép ghép nhiều thuộc tính src, alt trong một expression

**BEAN**

I,

[Application Context]là một giao diện tập trung quản lý và cấu hình các ```@Component``` và ```@Bean```.

@Component là một annotation đánh dấu một kiểu Class sẽ được khởi tạo một đối tượng, đối tượng này sẽ được quản lý trong [Application Context]

Trong lúc khởi động ```@SpringBootApplication``` sẽ quét (component scan) tất cả các ```@Component``` và ```@Bean```, nạp chúng, và sẵn sàng dùng để xử lý request gọi đến.

```@Autowired``` để tự động nạp đối tượng kiểu Zip vào ApplicationContext

`@Component``` này là singleton (duy nhất trong vòng đời ứng dụng).

**\*\*Singleton object\*\*** đối tượng duy nhất

**\*\*Vài đặc điểm của Autowired\*\***

1. Nó phải được đặt trên khai báo instant variable của class, hoặc ở constructor của class chứ không ở local variable của method

2. Tại thời điểm đối tượng của class X, trong ví dụ này là ```HomeController```, khởi tạo (construct), thì instant variable có ```@Autowired``` trong X sẽ được gán hay trỏ đến đối tượng singleton được khởi tạo trước đó và quản lý trong Application Context

II,

ApplicationContext là một singleton object(đối tượng duy nhất) quan trọng trong SpringBoot.

 Truy xuất đến ApplicationContext:

1. Sử dụng ```@Autowired``` gắn vào một biến kiểu ```ApplicationContext```

2. Sử dụng phương thức ```void setApplicationContext(ApplicationContext applicationContext)``` của ```interface ApplicationContextAware```

Hãy dòng giống hệt nhau vậy kết luận ```applicationContext1``` và ```applicationContext2``` đều trỏ về một đối tượng      ```ApplicationContext``` duy nhất.

III,

Khi đã nạp vào Application Context, Spring Boot không còn phân biệt đâu là Component và đâu là Bean. Chúng được gọi chung là Bean !

```@Bean``` đánh dấu cho phương thức public trả về đối tượng cần nạp vào ApplicationContext. Tên phương thức sẽ là tên của Bean. Phương thức này phải thuộc một class được đánh đấu là ```@Configuration```

IV,

Dependency: Thành phần phụ thuộc

Injection: Tiêm

Dependency Injection = Tiêm thành phần phụ thuộc vào đối tượng ~ Cấu hình thành phần

Thực tế ```@Bean``` linh hoạt hơn ```@Component``` ở chỗ: ```@Component``` đánh dấu class, còn ```@Bean``` đánh dấu phương thức trả về đối tượng. Phương thức thì có thể thêm logic tuỳ chỉnh theo ý đồ lập trình viên.

 @Bean(name = "electricEngine")

public Engine teslaEngine() {

    return new Engine("Tesla Electric Engine");

  }

Tham số ```name``` trong @Bean dùng để đặt lại tên cho Bean khác với tên mặc định của phương thức.

- ```@Autowired```: tự động tìm đến Bean có tên phù hợp để gán vào biến mà nó đánh dấu.

- ```@Qualifier```: nếu có nhiều Bean khác tên nhưng trả về đối tượng cùng kiểu, thì cần định rõ tên.

@Autowired

@Qualifier("manualsteering")

**\*\*Khi đã dùng ```@Autowired``` cho một thuộc tính, thì bạn không khai báo tham số ở contructor để khởi tạo thuộc tính này nữa.\*\***

Có nghĩa là ```@Autowired``` đã giúp tự động inject (tiêm, cấu hình) cho thuộc tính ```steering``` khi đối tượng ```Car``` được khởi tạo (construct).

tuỳ biến loại động cơ (engine), mà không phải sửa code. Vậy chúng ta sẽ đọc giá trị từ key ```engineType``` trong application.properties nhờ cú pháp sau đây:

```java

@Value("${engineType}")

private String engineType;

Dựa vào biến engineType được gán giá trị từ application.properties, bạn có thể lấy ra loại động cơ phù hợp từ Application Context thông quá lệnh ```context.getBean("nameOfBeanHere");```

Chú ý: ```ApplicationContext.getBean("typeEngineHere")``` sẽ lấy một đối tượng Bean được đăng ký, quản lý trong ApplicationContext. Đối tượng này thường là duy nhất trong vòng đời ứng dụng SpringBoot. Nếu bạn sử dụng phương pháp thông thường

```engine = new Engine("typeEngineHere")``` mỗi lần chạy, sẽ tạo ra một đối tượng động cơ mới.

**Bổ xung @Primary vào public Car funcar()**

```java

@Primary

@Bean

public Car funcar() {

    return new Car((Engine) context.getBean("hybridEngine"));

}

```

V,

**# 03 phương pháp Dependency Injection khác nhau**

- Cách 1: Constructor Based Injection

- Cách 2: Setter Based Injection

- Cách 3: Field Based Injection (hay dùng nhất vì dễ)

Trong [CarConfig.java], có phương thức ```public Car car()``` trả về Bean.

ở dòng cuối cùng ```return new Car(engine);```: đối tượng engine được truyền vào constructor của Car

return new Car(engine); //Construction based Dependency Injection

**\*\*Chú ý:\*\***

Với những class được annotated bởi ```@Component```, ```@Controller```, ```@Service```, ```@Respository``` , chúng ta phải sử dụng phương pháp inject qua constructor dùng ```@Autowired```

chassis.setDoor(door()); //Setter based Dependency Injection

đoạn lệnh dùng đối tượng Bean trả về từ phương thức ```door()``` truyền vào Setter method của chassis

**## Cách 3: Field Based Injection**

DI cho thuộc tính, chúng ta sử dụng ```@Autowired``` đánh dấu cho thuộc tính.

public class Car {

  @Autowired

  @Qualifier("manualsteering")

  private Steering steering; //Field based Dependency Injection

  ...

}

```

VI,

Phải tạo mới đối tượng mỗi lần dùng đến và vẫn đăng ký đối tượng đó vào ApplicationContext. Dùng đến 2 annotation đó là ```@Scope("prototype")``` và ```@Lazy```

Annotation ```@Scope("prototype")``` định hướng cách tạo đối tượng Bean luôn là bản mẫu, chứ không phải là đối tượng duy nhất. Nó ngược với ```@Scope("singleton")```

```@Lazy``` ở đây có nghĩa là thuộc tính này khi nào dùng đến hãy khởi tạo.

 Bộ đôi ```@Scope("prototype")``` và ```@Lazy``` sẽ tạo mới đối tượng Bean mỗi khi đối tượng này được dùng / truy xuất.

VII,

```@Controller```. Khái niệm Controller phổ biến trong mọi web framework vì chúng đều chung một design patter là MVC (Model - View - Controller)

- Controller làm nhiệm vụ hứng các lời gọi đến (request)

- View làm nhiệm vụ tạo ra giao diện hiển thị. Chúng ta đã học Thymeleaf Template Engine

- Model là dữ liệu sẽ kết hợp với View Template để tạo thành giao diện

Trong Controller, chúng ta tạo nên model

// Dựng model để trả về cho View Template đây

    model.addAttribute(BOOKS, bookCollection);

```@Service``` dùng để đánh dấu cho loại component sẽ được Controller sử dụng để đọc / ghi dữ liệu.

Phần Service đó có thể được gọi bởi nhiều Controller khác nhau.

 Service component thường là Singleton (duy nhất)

```@Service``` bản chất của nó cũng là một ```@Component```. Service cũng cấp phương thức xử lý nghiệp vụ cho Controller.

MappingIterator<Car> mi = oReader.readValues(reader); //Iterator đọc từng dòng trong file

\*\*\*Class:

Singleton: khởi tạo ngay

Prototype: dùng đến sẽ tạo instance mới

@Lazy

@Bean

(Dùng đến mới khởi tạo)

Class khởi tạo, khởi tạo tất cả các dependencies trong đó

Dùng Prototype:

@Lazy

Application context

Proxy(trung gian, không cho hai phần biết nhau)

Object Factory

@Getter

@Setter

Tác dụng cho toàn bộ

@NoArgs

@AllArgs

@Data

**REST API**

I，

@RestController  //Đánh dấu đây là REST API

@RequestMapping(value = "/api/") // Đường dẫn gốc cho controller này là /api/

@GetMapping("/books/{bookId}")

II，

**# OpenAPI để tạo văn bản chú thích cho REST API**

@Operation(summary = "Get a book by id")

  public ResponseEntity<Book> findBookById(

    @Parameter(description = "id of book to be searched")

- ```@Operation(summary = "Get a book by id")``` : Mô tả tên phương thức, sẽ được OpenAPI trích xuất

- ```@Parameter(description = "id of book to be searched")```: Mô tả tên tham số sẽ được OpenAPI trích xuất

Để front end developer dễ hình dùng hơn, cần phải mô tả ràng buộc (buộc phải có, độ dài trường...) và dữ liệu mẫu.

Hãy sử dụng các annotation:

- ```@NotBlank```: không được rỗng

-``@Size(min = 5, max = 400, message = "Tên sách từ 4 đến 400 ký tự")```: số lượng ký tự min và max. Tham số message dùng để báo lỗi khi dữ liệu phạm quy.

@NotNull(message = "Name cannot be null")

@Schema(description = "Unique ID of Contact.", example = "1", required = true)

**Chỉ định OpenAPI bỏ qua một phương thức REST API**

Trong annotation ```@Operation``` hãy thêm tham số ```hidden = true``` như đoạn code dưới.

@Operation(summary = "Beta API method, đang kiểm thử", hidden = true)

### **Gom nhóm cụm các API cùng chức năng**

 nên gom những phương thức REST thuộc cùng một domain lại

Trong ví dụ, có 2 models [Book.java](https://github.com/TechMaster/SpringBootBasic/blob/main/rest/02OpenAPI/bookstore/src/main/java/vn/techmaster/bookstore/model/Book.java) và [Contact.java](https://github.com/TechMaster/SpringBootBasic/blob/main/rest/02OpenAPI/bookstore/src/main/java/vn/techmaster/bookstore/model/Contact.java), tương ứng có 2 REST Controller [BookAPI.java](https://github.com/TechMaster/SpringBootBasic/blob/main/rest/02OpenAPI/bookstore/src/main/java/vn/techmaster/bookstore/controller/BookAPI.java) và [ContactAPI.java](https://github.com/TechMaster/SpringBootBasic/blob/main/rest/02OpenAPI/bookstore/src/main/java/vn/techmaster/bookstore/controller/ContactAPI.java).

Trong file [OpenAPIConfig.java](https://github.com/TechMaster/SpringBootBasic/blob/main/rest/02OpenAPI/bookstore/src/main/java/vn/techmaster/bookstore/config/OpenAPIConfig.java), tôi bổ xung thêm 2 bean:

public GroupedOpenApi bookOpenApi() {…}

@Bean

public GroupedOpenApi contactOpenApi() {…}

**CRUD**

I, @Repository： Đánh dấu một Class Là tầng Repository, phục vụ truy xuất dữ liệu.

@RequestMapping("/book") : Đường dẫn /book sẽ là đường dẫn gốc chung cho các phương thức bên trong BookController

II,

public void readCSV(String csvFile) {

    try {

      File file = ResourceUtils.getFile("classpath:static/" + csvFile);

      CsvMapper mapper = new CsvMapper(); // Dùng để ánh xạ cột trong CSV với từng trường trong POJO

      CsvSchema schema = CsvSchema.emptySchema().withHeader().withColumnSeparator('|'); // Dòng đầu tiên sử dụng làm Header

      ObjectReader oReader = mapper.readerFor(Book.class).with(schema); // Cấu hình bộ đọc CSV phù hợp với kiểu

      Reader reader = new FileReader(file);

      MappingIterator<Book> mi = oReader.readValues(reader); // Iterator đọc từng dòng trong file

      while (mi.hasNext()) {

        Book book = mi.next();

        this.add(book);

      }

    } catch (IOException e) {

      System.out.println(e);

    }

  }

 public void add(Book book) {

    //Cơ chế tự tăng

    int id;

    if (collections.isEmpty()) {

      id = 1;

    } else {

      Book lastBook = collections.get(collections.size() - 1);

      id = lastBook.getId() + 1;

    }

    book.setId(id);

    collections.add(book);

  }

III,

Các biến được sử dụng thông qua Optional Class sẽ tránh được lỗi NullPointerException.

public Optional<Book> get(int id) {

  return collections.stream().filter(u -> u.getId() == id).findFirst();

}

```

@GetMapping(value = "/{id}")

public String getByID(@PathVariable("id") int id, Model model) {

  Optional<Book> book = bookDao.get(id);

  if (book.isPresent()) {

    model.addAttribute("book", book.get());

  }

  return "book";

}

```BindingResult result``` là kết quả của quá trình chuyển dữ liệu trong Body của POST request sang đối tượng Book:

Lỗi khi chuyển đổi sẽ được trả về qua phương thức ```BindingResult.hasErrors()```

"redirect:/book"; // Chuyển về đường dẫn /book

**File Upload**

1. @ModelAttribute is an annotation that binds a method parameter or method return value to a named model attribute and then exposes it to a web view.
2. To upload our file, we can build a simple form in which we use an HTML input tag with type='file'.
3. To store the uploaded file we can use a MultipartFile variable. We can retrieve this variable from the request parameter inside our controller's method:

@RequestMapping(value = "/uploadFile", method = RequestMethod.POST)

**public** String **submit**(@RequestParam("file") MultipartFile file, ModelMap modelMap) { modelMap.addAttribute("file", file);

**return** "fileUploadView"; }

MultipartFile class provides access to details about the uploaded file

1. To upload multiple files in a single request, we simply put multiple input file fields inside the form:

<form:form method="POST" action="/spring-mvc-java/uploadMultiFile" enctype="multipart/form-data">

<table>

<tr>

<td>Select a file to upload</td>

<td><input type="file" name="files" /></td>

</tr>

<tr>

<td>Select a file to upload</td>

<td><input type="file" name="files" /></td>

</tr>

<tr>

Each input field has the same name so that it can be accessed as an array of MultipartFile

**public** String **submit**(@RequestParam("files") MultipartFile[] files, ModelMap modelMap) { modelMap.addAttribute("files", files);

**return** "fileUploadView"; }

1. If we want to control the maximum file upload size, we can edit our application.properties

spring.servlet.multipart.max-file-size=128KB

spring.servlet.multipart.max-request-size=128KB

We can also control whether file uploading is enabled, and the location for file upload:

spring.servlet.multipart.enabled=true

spring.servlet.multipart.location=${java.io.tmpdir}

1. multipart/form-data: encoding HTML form types for a POST request, used for non-acsii data and binary files
2. The enctype attribute specifies the multipart/form-data encoding type

  <form action="/doUpload" method="post" enctype="multipart/form-data">

This lets the browser know how to encode the form

1. InputStream , represents an ordered stream of bytes. Useful when reading data from a file, or received over the network.
2. the var keyword means the type for the local variable will be inferred by the compiler, so you don't need to declare that

**JPA**

1. @JoinColumn: tạo khóa ngoại liên kết với bảng

Eg: @JoinColumn(name=”person\_id”) ở bảng Person

Tạo một khóa ngoại person\_id ở bảng Person liên kết với bảng Address

1. @OneToMany:

Entity phía One cần có thuộc tính kiểu List, Set hoặc HashMap để lưu các Entity phía Many + 2 phương thức thêm và bớt Entity phía Many

Eg: Một post có nhiều comment, comment xuất hiện dưới dạng ArrayList

public void addComment(Comment comment) {

comments.add(comment);

comment.setPost(this); //Mỗi comment lại gán chính

}

public void removeComment(Comment comment) {

comments.remove(comment);

comment.setPost(null);

}

@ManyToOne

Eg: Chỉ có một thực thể là post

**3**.Cascade Operation là Entity phía One có những hành động gì, thì áp dụng hành động tương tự ở Entity phía Many.

Có 5 loại hành động Entity thực hiện với Persistence Context. Loại All là tổ hợp tất cả.

1. Persist: lưu entity mới tạo nhưng chưa lưu (đang ở trạng thái transient) vào Persistence Context. Chỉ thêm một động tác flush, thì các thay đổi thực sự được ghi ổn định vào CSDL.
2. Merge: sát nhập một entity đã tách ra (không được quản lý giám sát thay đổi bởi Persistence Context nữa)
3. Remove: xoá entity ra khỏi Persistence Context.
4. Refresh: đọc lại dữ liệu từ CSDL vào Entity trong Persistence Context.
5. Detach: tách entity ra khỏi Persistence Context. Tạm thời không quản lý thay đổi nữa, nhưng không xoá.
6. All (tổng hợp tất cả các loại hoạt động trên)

**4**. Hibernate: quản lí các enity và lưu vào Database(trung gian)

Eg: Muốn lưu entity phải lưu vào persistence context

1. orphanRemoval = true khi một Entity phía Many không được tham chiếu bởi một Entity phía One, thì nó sẽ bị xoá.
2. @BeforeEach: chạy trước @Test