01

Spring Boot

01 Spring Boot

- Java 기반의 애플리케이션 프레임워크인 Spring Framework의 확장판
  - 자동 설정(Auto-Configuration)
    - 애플리케이션의 설정을 자동으로 해주어 개발자는 최소한의 설정
  - 내장 서버 제공 : Tomcat, Jetty, Undertow
  - 간편한 의존성 관리: 다양한 starter 패키지를 제공
    - spring-boot-starter-...

02. REST API

API (Application Programming Interface)

- 응용프로그램에서 다른 응용프로그램을 제어할 수 있도록 만든 Interface
- API를 사용하면 내부 구현 로직을 알지 못해도 정의되어 있는 기능을 쉽게 사용할 수 있음

- 인터페이스(Interface) : 어떤 장치간 정보를 교환하기 위한 수단
- Ex) 마우스, 키보드, 터치패드

02. REST API

REST (Representational State Transfer)

- 자원의 이름으로 구분하여 해당 자원의 상태를 교환하는 것
- REST는 서버와 클라이언트의 통신방식 중의 하나임
- HTTP URI(Uniform Resource Indentifier)를 통해 자원을 명시
- HTTP method(GET, POST, PUT, DELETE)를 통해 자원을 교환

02. REST API의 특징

- **Server – Client 구조**
  - 자원이 있는 쪽이 Server, 요청하는 쪽이 Client
  - Client와 Server가 독립적으로 분리되어 있어야 함
- **Stateless**
  - 요청 간에 Client정보가 Server에 저장되지 않음
  - Server는 각각의 요청을 완전히 별개의 것으로 처리
- **Cacheable**
  - HTTP 프로토콜을 그대로 사용하기 때문에 HTTP의 특징인 캐쉬 기능을 적용, 대량의 요청을 효율적으로 처리하기위해 캐쉬를 사용
  - Client 동일한 요청(GET)을 반복할 때 서버에 부담을 줄이고, 응답 시간을 단축하기 위해 응답 데이터를 캐시
- **REST 구조를 구현한 웹서비스를 RESTful 하다고 함**

03. REST API 설계 규칙

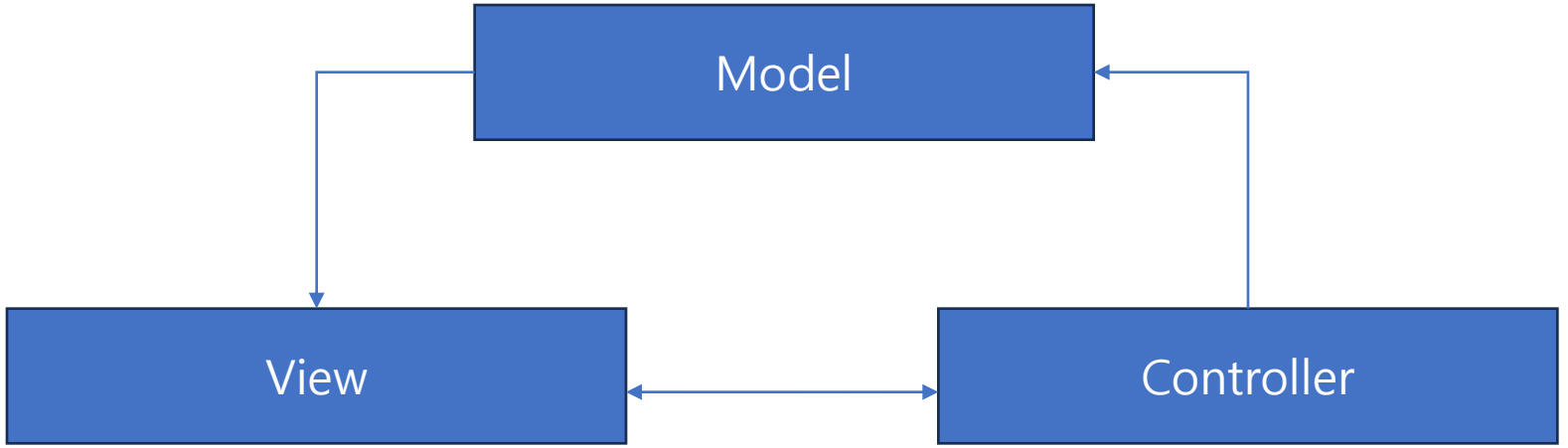
- URI 통해 자원을 표현해야 함
  - http://www.example.com/item/112
    - item: Resource
    - 112: Resource ID
- 자원의 조작은 HTTP Method(get, post, put, delete)를 통해 표현해야 함
  - URI에 행위가 들어가면 안됨(동사:X, 명사:O)
  - Header를 통해 CRUD (get, post, put, delete) 를 표현해야 함
  - Request Packet에 포함되어 있는 Header:
    - HTTP Method(get, post, put, delete)
    - URI
    - HTTP 버전 정보
    - ...

03. REST API 설계 규칙

- URI에는 소문자를 사용
- Resource의 이름이나 URI가 길어질 경우 하이픈(-)을 통해 가독성을 높임
- 언더바(_)는 사용하지 않음
- 파일 확장자는 표현하지 않음

04. MVC 패턴(Model, View, Controller)

- 디자인 패턴의 하나인 MVC패턴은 Application을 구성할 때 구성요소를 세 가지의 역할로 구분한 패턴을 의미
- 사용자 인터페이스로부터 비즈니스 로직을 분리하여 서로 영향이 없이 쉽게 고칠수 있는 설계가 가능



04. MVC 패턴(Model, View, Controller)

Controller

- Model과 View사이에 다리 역할
- 사용자로부터의 입력에 대한 응답으로 Model및 View를 업데이트 하는 로직을 포함
- 사용자의 모든 요청은 Controller를 통해 진행
- Controller로 들어온 요청은 어떻게 처리할지 결정하여 Model로 전달

- 예) 쇼핑몰에서 쇼핑을 검색하면 그 키워드를 Controller가 전달받아 Model과 View에 적절하게 입력을 처리하여 전달

04. MVC 패턴(Model, View, Controller)

Model

- 데이터를 처리하는 영역
- 데이터 베이스와 연동하는(DAO)와 데이터의 구조를 표현하는 Do 로 구성됨
- 검색을 위한 키워드가 넘어오면 데이터베이스에서 관련된 상품의 데이터를 받아 View에 전달

04. MVC 패턴(Model, View, Controller)

View

- 데이터를 보여주는 화면 영역 자체
- 사용자 인터페이스(UI)요소들이 여기에 포함되며 데이터를 각 요소에 배치함
- View에서는 별도의 데이터를 보관하지 않음
- HTML 파일의 태그요소들이 View에 해당함

02

Annotation

01. Annotation

- 클래스, 메서드, 필드 등에 추가적인 메타데이터를 제공하여 Spring이 객체를 관리하고 필요한 동작을 수행할 수 있도록 돕는 역할

```
@Controller
@RequiredArgsConstructor
public class ProductController {
    private final ProductRepository productRepository;


    @GetMapping(value="/list")
    String list(Model model){
    …
   …..
    }
}
```

01. @SpringBootApplication

- Spring Boot 애플리케이션의 진입점을 정의
- 다음 세 가지 Annotaion 포함한 Annotaion
  - @EnableAutoConfiguration
    - Spring Boot가 애플리케이션의 설정을 자동으로 구성
    - 프로젝트에 데이터베이스 라이브러리가 있으면 데이터베이스 설정을 자동으로 추가
  - @ComponentScan
    - @Component, @Service, @Repository, @Controller 등으로 정의된 빈(Bean)을 자동으로 검색하여 Spring 컨텍스트에 등록
    - 일반적으로 @SpringBootApplication이 선언된 클래스의 패키지와 하위 패키지를 스캔
  - @Configuration:
    - Spring의 설정 클래스로 사용됨을 나타내고, 빈을 정의할 수 있음

✓ Spring IoC(Inversion of Control) Container

- Spring 프레임워크의 핵심 구성 요소
- 애플리케이션의 객체를 생성하고 관리
- 객체 간의 의존성 주입(DI)을 처리하는 역할
- IoC(제어의 역전)
  - 객체의 생성과 의존성 관리 책임을 애플리케이션 코드가 아닌 컨테이너(Spring IoC Container)가 담당하게 하는 것
  - 필요한 객체를 직접 생성하는 것이 아니라, IoC 컨테이너가 객체를 생성하여 의존성을 주입해 줌으로써 객체 간의 결합도를 낮추고 유지보수를 쉽게 함
- 빈(Bean)
  - Spring IoC 컨테이너에서 관리되는 객체를 의미
  - 개발자가 직접 인스턴스를 생성하지 않고, 컨테이너가 생성과 관리를 담당

02. @Controller, @RestController

- REST API 엔드 포인트를 구성하는 컨트롤러
- @Contrlloer
  - View Template을 반환(HTML파일)
  - Json형태의 데이터를 반환하려면 @ResponseBody와 함께 사용
- @RestController
  - @Contrlloer 와 @ResponseBody를 합한 것
  - Json형태의 데이터를 반환
  - REST API 에서 주로 사용

03. @GetMapping

@GetMapping

- GET Method로 들어오는 URI와 메소드를 연결
- @GetMapping(value="/list")
- @GetMapping(value="/list/{var}")
  - 경로와 파라미터 연결
  - 메소드의 매개변수에 연결

```
@GetMapping(value="/hello")
public String getHello() {
    return "Hello World";
}
@GetMapping(value="/variable/{var}")
public String getVariable(@PathVariable String var){
    return var;
}
@GetMapping(value="/varable1/{var}")
public String getVariable1(@PathVariable("var") String v){
    return v;
}
```

04. @PostMapping

@PostMapping

- POST Method로 들어오는 URI와 메소드를 연결
- Client의 from 안의 요소에 name속성에 설정해준 값과 연결된 메소드의 매개변수 이름이 같아야 함

```
@PostMapping("/new-product")
String writePost(@RequestParam String title,
                 @RequestParam Integer price) {
    this.productService.save(title, price);
    return "redirect:/list";
}
```

05. @PutMapping/@DeleteMapping

- @PutMapping : 수정
- @DeleteMapping : 삭제

```
@DeleteMapping(value="/product/{id}")
public ResponseEntity<String> deleteProduct(@PathVariable("id") Integer id){
    Integer pid=this.productService.deleteProduct(id);
    return ResponseEntity.status(200).body(String.valueOf(pid));
}

@PutMapping(value="/product")
public ResponseEntity<ProductDTO> updateProduct(@RequestBody ProductDTO
productDTO){
    ProductDTO product=this.productService.updateProduct(productDTO);
    return ResponseEntity.status(200).body(product);
}
```

06. @RequestParam

@RequestParam

- URI의 Query 문자열 및 폼데이터를 메소드의 매개변수로 받음

```
@GetMapping(value="request1")
public String getRequest1(@RequestParam String name,
                          @RequestParam String age){
    return "이름 : "+name + "  나이 : "+ age;
}

@GetMapping(value="request2")
public String getRequest2(@RequestParam Map<String,String> map){
    StringBuilder sb = new StringBuilder();
    map.entrySet().forEach(item->{
        sb.append(item.getKey()+" : "+ item.getValue() + "¥n");
    });
    return sb.toString();
}

@GetMapping(value="request3")
public String getRequest3(MemberDTO memberDTO){
    return memberDTO.toString();
}
```

03

데이터베이스 연결

01. 템플릿 엔진

■ 데이터를 템플릿에 전달하여 렌더링해주는 엔진

- build.gradle에 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

- 사용할 메소드에 매개변수로 Model 변수 추가
- model.addAttribute("데이터 변수명", "데이터");
- html 파일의 데이터가 렌더링될 요소에 th:text="${변수명}"
  - model.addAttribute("product", "바지");
  - <h1 th:text="${product}"></h1>

02. MySql 연결

- 데이터베이스 탭에서 Data Source-MySql
- 계정정보 입력 후 연결
- build.gradle에 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
runtimeOnly 'com.mysql:mysql-connector-j'
```

- application.properties 에 연결 정보 입력

```
spring.datasource.url=jdbc:mysql://127.0.0.1/shop
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.properties.hibernate.show_sql=true
spring.jpa.hibernate.ddl-auto=update
```

02. MySql 연결

Entity Class 작성

- @Entity : 해당 Class가 Table로 등록됨
- @Table(name="producttbl") : Class 이름과 실제 Table이름이 다를때 사용
- @Id : 해당 멤버변수를 Table의 PK로 지정
- @GeneratedValue(strategy = GenerationType.IDENTITY)
  - Mysql의 auto_increment 로 지정한 것과 같은 효과
- @Column : 제약조건 추가
  - @Column(nullable=false), @Column(unique = true)...

```
@Entity
@Table(name="producttbl")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long id;
    public String title;
    public Integer price;
}
```

02. MySql 연결

Repository Interface 작성

JpaRepository<> Interface 상속

- JPA를 기반으로 한 CRUD작업 및 데이터베이스 조작 기능을 사용 가능
  - save(S entity), findById(ID id), findAll(),.. 다양한 메소드 사용 가능

JpaRepository<Item, Long> :

- Item : Table 자료형 (Table Class)
- Long : PK의 자료형

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemRepository extends JpaRepository<Product, Long> {

}
```

02. MySql 연결

Controller 에서 사용

- private final ProductRepository productRepository
  - Repository 등록
- @RequiredArgsConstructor
  - 모든 final 필드와 @NonNull이 붙은 필드에 대해 생성자를 자동으로 생성 (의존성 주입을 위한 생성자 생성)

```
@Controller
@RequiredArgsConstructor
public class ProductController {
    private final ProductRepository productRepository;

@GetMapping(value="/product-list")
    public String list(Model model) {
        List<Item> result=this.productRepository.findAll();
        model.addAttribute("products", result);
        return "list.html";
    }
}
```

02. MySql 연결

@ModelAttribute

- Form에서 보낸 데이터를 자동으로 테이블 레코드 객체로 만들어 주고 전달

```
@PostMapping("/new-product")
String writePost(@ModelAttribute Product product) {
    this.productRepository.save(product);
    return "redirect:/product-list ";
}
```

02. MySql 연결

Optional<T> 타입

- null 안전성을 제공하기 위한 자료형
- 값이 있을 수도 있고 없을 수도 있다.
- 두 가지 상태를 가질 수 있다.
  - 값이 있는 상태 (present)
  - 값이 없는 상태 (empty)
- NullPointerException 방지

```
@GetMapping(value="/edit-form/{id}")
public String edit(@PathVariable("id") Long id, Model model) {
    Optional<Item> result=this.productRepository.findById(id);
    if(result.isPresent()) {
        model.addAttribute("product", result.get());
    }
    return "edit_form.html";
}
```