

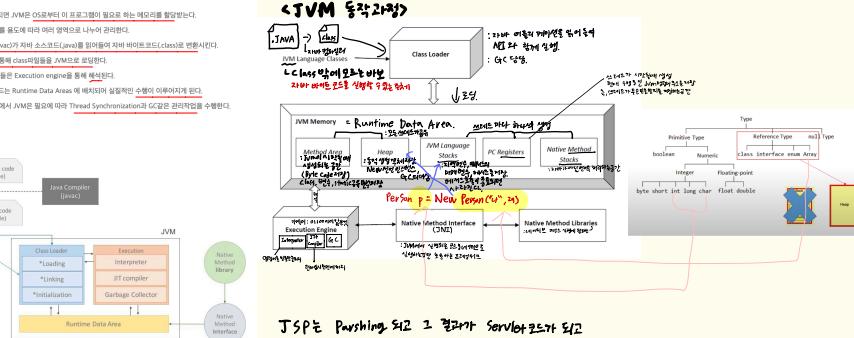


Spring

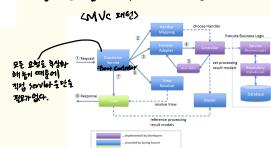
자바 프로그램 실행 과정

- 프로그램이 실행되면 JVM은 OS로부터 이 프로그램이 필요로 하는 메모리를 할당받는다.
 - JVM은 이 메모리를 풀에서 라는 영역으로 나누어 관리한다.
 - 자바 디파일러(Java)가 자바 소스코드(.java)를 읽어들여 자바 바이트코드(.class)로 변환시킨다.
 - Class Loader를 통해 class 디파일(.class)을 JVM으로 전송한다.
 - 포함된 class 파일들은 Execution engine(Java Virtual Machine)을 통해 해석된다.
 - 해석된 바이트코드는 Runtime Data Area에 메모리에 실질적인 수행이 이루어지게 된다.
- 이러한 실행 과정 속에서 JVM은 멀티 코어 Thread Synchronization과 GC(Heap 관리)작업을 수행한다.

VM>>



JSP는 Parsing 되고 그 결과가 Servlet 코드가 되고
동적으로 커버링되어 class 가 된다.



스프링과 웹은 관계가 없고 Spring MVC라는 확장기 용법하고 관계가 있는 것.

2. Servlet API

Servlet의 개념

별기기로써 자체적인 처리기능이 있는 허브 역할을 한다.

- Server 내부에埋藏된 Java Programs
- 개인화 처리기능

Servlet Program의 기본적인 동작 과정



Web Server는 HTTP 요청을 Web Container Container에 전달한다.

- 1. http://www.google.com/index.html에 접속하는 경우
- 2. 웹 브라우저가 웹 서버에게 “index.html”을 찾고자 하는 요청

Web Container는 해당 요청에 맞는 Servlet을 선택하고 실행하는 역할을 한다.

- 1. “index.html”에 맞는 Servlet을 선택하고 실행하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 1. View(HTML)를 처리하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 2. Business Logic(Java 코드)를 처리하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 3. View(HTML)와 Business Logic(Java 코드)를 통합하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 4. View(HTML)를 전달하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 5. Business Logic(Java 코드)를 처리하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 6. View(HTML)를 전달하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 7. Business Logic(Java 코드)를 처리하는 역할

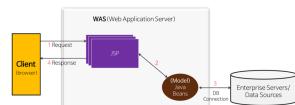
선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

- 8. View(HTML)를 전달하는 역할

선택된 Servlet은 View(HTML)와 Business Logic(Java 코드)를 처리하는 역할을 한다.

Servlet과 JSP의 관계

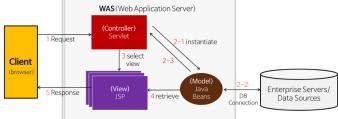
JSP만 이용하는 모델



- JSP가 사용자의 요청을 받아 Java Bean(DTO, DAO)을 호출하여 적절한 동적인 페이지를 생성한다.
- JSP로 작성된 프로그램은 내부적으로 WAS로 파일로 변환
- JSP 테그를 분해하고 추출하여 다시 순수한 HTML 폼 페이지로 변환
- 클라이언트로 흘림
- 특징
 - 개발 속도가 빠르다.
 - 배우기 쉽다.
 - 프레젠테이션 로직(View)과 비즈니스 로직(Controller)이 혼재한다.
 - JSP 코드가 복잡해져 유지 보수가 어려워진다.

JSP와 Servlet을 모두 이용하는 모델 (MVC Architecture)

MVC Architecture



- JSP와 Servlet을 모두 사용하여 프레젠테이션 로직(View)과 비즈니스 로직(Controller)을 분리한다.
- View(보이지는 부분)는 HTML이 중심이 되는 JSP를 사용
- Controller(다른 자바 클래스에 데이터를 넘겨주는 부분)는 Java 코드가 중심이 되는 Servlet을 사용
- Model은 Java Beans, DTO, DAO를 통해 MySQL과 같은 Data Storage에 접근
- 구체적인 MVC 패턴은 MVC-Architecture 참고



Shutdown [2 가지 종류] **Stopped** : 정지된 것 (복구 가능)
Terminated : 완전 삭제된 것 (복구 불가능)

== 연산자

비교 대상이 Primitive type이면 \Rightarrow 가 값 비교

// Reference type이면 \Rightarrow 주소 비교

ex) `String str1 = "4"` \Rightarrow 이전식의 선언을
`String str2 = "4"` 자바로 선언했다고 한다.

결과 == 은 true

`String str3 = new String("4")`
`String str4 = new String("4")`

결과 == 은 false

`str3, 4는 서로 다른 메모리 주소이다.`
↳ 동적 할당으로



equals()는 Default는 Primitive type 검사

Reference type이면 \Rightarrow 주소 비교

`str3.equals(str4)`

결과는 true

주소는 달라도 값이 같으므로

but, 만약 Person이 age와 name을 가진 객체라면
 \Rightarrow false

```

@Controller
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @PostMapping(value = "/info")
    public @ResponseBody User info(@RequestBody User user){
        return userService.retrieveUserInfo(user);
    }

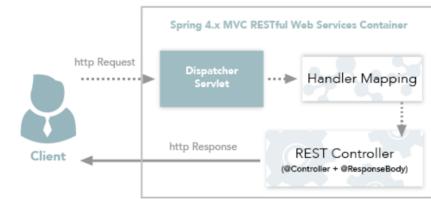
    @GetMapping(value = "/infoView")
    public String infoView(Model model, @RequestParam(value = "userName", required = true) String userName){
        User user = userService.retrieveUserInfo(userName);
        model.addAttribute("user", user);
        return "/user/userInfoView";
    }
}

```

위 예제의 info는 User라는 데이터를 반환하고자 하고 있고, User를 json으로 반환하기 위해 `@ResponseBody`라는 어노테이션을 붙여주고 있습니다. infoView 함수에서는 View를 전달해주고 있기 때문에 String을 반환값으로 설정해주었습니다. (물론 이렇게 데이터를 반환하는 RestController와 View를 반환하는 Controller를 분리하여 작성하는 것이 좋습니다.)

[RestController] - Json 데이터는 꽉!

`@RestController`은 Spring MVC Controller에 `@ResponseBody`가 추가된 것입니다. 당연하게도 RestController의 주용도는 Json 형태로 객체 데이터를 반환하는 것입니다. 개인적으로는 VueJS + Spring boot 프로젝트를 진행하며 Spring boot를 API 서버로 활용할 때 또는 Android 앱 개발을 하면서 데이터를 반환할 때 사용하였습니다.



1. Client는 URI 형식으로 웹 서비스에 요청을 보낸다.
2. Mapping되는 Handler와 그 Type을 찾는 DispatcherServlet이 요청을 인터셉트한다.
3. RestController는 해당 요청을 처리하고 데이터를 반환한다.

즉, back단과 Front단의 분리

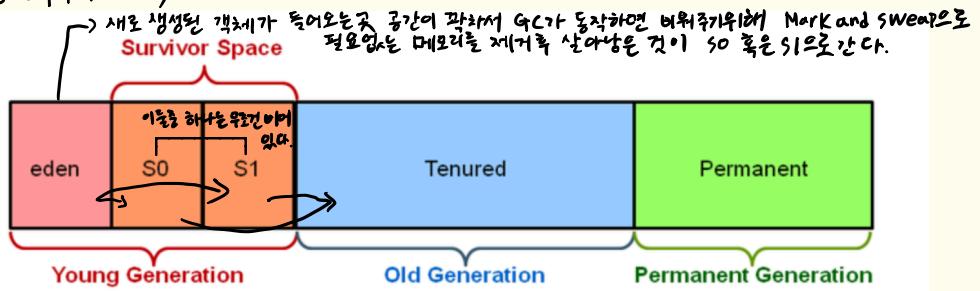
부록

※ `@JsonManagedReference` → 양방향 매핑에서 순환 참조를 방지하기 위해 (정방향 참조할 변수에 추가하여 직역화에 포함)

`@OneToMany(mappedBy = "funding", cascade = CascadeType.ALL, fetch = FetchType.LAZY)`
`List<FundingFile> fundingFiles = new ArrayList<>();`

영속성 전이 : 한쪽이 바뀌면 연관되어 있는 곳도 바뀐다.

JVM Memory 중 Heap 영역



young - 비교적 젊은 reference가 살아있는 곳

old - 특정 횟수 이상을 살아남은 reference가 살아있는 곳

permanent - Method Area의 메타정보가 기록된 곳

eden - young 영역중에서도 특히 방금 막 생성된 녀석들이 있는 곳

survivor - 영역이 두개 존재하는데 eden에서 생존된 녀석들이 당분간 생존해 있는 곳

이렇게 영역을 나누는 이유는 Full GC를 막기 위해서다.

Full GC는 모든 heap 영역을 뭉땅 뒤져서 생존해 있는 녀석들을 모조리 죽여버리는 역할을 한다.

당연히 시간도 오래걸린다. 그래서 GC 역시 세단계로 나눠져 있다.

Minor GC - young 영역만 뒤져서 다 죽인다

Major GC - old 영역까지 뒤져서 다 죽인다.

Full GC - permanent 영역까지 뒤져서 다 죽인다.

일반적으로 GC가 작동하는 알고리즘에 대해서 설명하도록 하겠다.

일단 기본적으로 MarkSweep을 하는건 동일하다.