참고강의

<https://www.youtube.com/watch?v=XBG6CUtVCIg&list=PL93mKxaRDidG_OIfRQ4nztPQ13y74lCYg&index=1>

블로그

<https://getinthere.tistory.com/category/%EC%8A%A4%ED%94%84%EB%A7%81%EB%B6%80%ED%8A%B8%20with%20JPA%20%EA%B0%9C%EB%85%90>

<스프링이란?>

- 스프링은 프레임워크다.

틀 안에서 동작한다. 틀에서 벗어나지 마라! 니 맘대로 만들지말고 틀에 맞춰서 개발해라! 그렇게 하면 너도 나처럼 좋은 프로그램을 만들 수 있을거야! --> 프레임워크

- 스프링은 오픈소스이다.

오픈소스란 소스코드가 공개되어 있는 것! 스프링이 어떻게 만들어졌는지 코드를 볼 수 있다! 그말은 내부를 뜯어 고칠 수도 있다는 것!

공부를 너무 많이해서 내부 깊숙한 것까지 알게되어 불편한 게 생겼을 때 직접 고친 후 contribute도 가능하다!

- 스프링은 IoC 컨테이너를 가진다.

IoC 컨테이너? Inversion of Control – 제어의 역전(주도권이 스프링에게 있다!)

class -> 설계도

object -> 실체화가 가능한 것

instance -> 실체화 된 것

ex) 가구 : object가 아니다! 추상적이기 때문.(abstract class)

의자, 침대 : object이다! 실체화가 가능하기 때문 -> 만들어지는 순간 intance가 된다.

내가 오브젝트를 **직접 new**를 해서 heap이라는 공간에 올리게 되면,

오브젝트를 내(메소드 내부)가 관리한다. 그러나 이걸 다른 메소드에서 사용할 때 많은 다시 new하던가 하는 복잡한 로직이 있기 때문에 Spring에서 내가 만든 class들을 스캔해서 직접 heap메모리에서 관리해주어 사용하기 편리하게 해준다. 내가 원하는 모든 class의 메소드에서 사용할 수 있다. --> 제어의 역전

- 스프링은 DI를 지원한다.

Dependency Injection 의존성 주입. 스프링이 스캔한 나의 class들을 다른 곳에서 사용하는 것을 di라고 한다. IoC와 DI에 의해 프로그래밍이 편해진다!

- 스프링은 엄청나게 많은 필터를 가지고 있다.

검열해주는 것.

스프링 자체 필터 사용 or 스프링의 많은 필터 중 사용되지 않았던 필터를 사용 or 필터 직접 생성하여 사용할 수 있다.

스프링 컨테이너의 필터 -> 인터셉터(AOP)

- 스프링은 엄청나게 많은 어노테이션을 가지고 있다. (리플렉션, 컴파일체킹)

// (주석) - 컴파일러가 무시함.

어노테이션(주석 + 힌트) : 컴파일러가 무시하지 않음.

Animal이란 객체에 run()이라는 메소드가 있다고 가정.

Dog가 Animal을 상속하고 run()을 재정의 하고 싶을 때, 윗 줄에 @override (어노테이션)을 사용하면 컴파일체킹할 때 좋음.

또한 스프링은 어노테이션 객체를 생성하여 사용함.

@Component -> 클래스를 메모리에 로딩

@Autowired -> 로딩된 객체를 해당 변수에 집어넣어

@Component

Class A {

}

-> 스프링이 class를 스캔해서 heap 메모리에 로딩함. 어노테이션으로 약속했기 때문에!

클래스 A를 클래스 B에서 사용하고 싶은 상황

어노테이션을 사용하면 편하게 불러올 수 있다.

Class B {

@Autowired

A a;

}

이때 B 클래스 내부에 어떤애(메서드,필드,어노테이션)가 있는지 분석하고 동작시키는 걸 **리플렉션**이라고 한다. 리플렉션은 **런타임시 발생**한다.

- 스프링은 MessageConverter를 가지고 있다. 기본값은 현재 Json이다.

자바 오브젝트와 파이썬 오브젝트가 서로 통신(request or response)하거나 주고받을 때 언어가 다르므로 힘들다. 그래서 그 중간에 Json오브젝트로 바꾸어 서로 주고받는다. 둘다 중간언어로 이해하기 쉬움.

자바 프로그램 -> JSON (request) -> 파이썬 프로그램

자바 프로그램 <- JSON (request) <- 파이썬 프로그램

중간언어(MessageConverter) : xml -> Json

- 스프링은 BufferedReader와 BufferedWriter를 쉽게 사용할 수 있다.

통신(영어권)

한문자 -> 8bit = 2^8 = 256. 256가지의 문자 전송가능!

8bit(1byte)씩 끊어 읽으면 영어권에서는 한 문자씩 받을 수 있다.

한국은 2byte, 중국은 3byte. 나라마다 다르다.

전세계에서 인터넷으로 커뮤니케이션하는 시대에, 자기나라 만의 방식으로 글을 인코딩하게 되면 전세계적인 데이터 통신은 불가능! 따라서 유니코드에서 utf-8 이라는 캐릭터 인코딩을 제공함. 3byte 통신.

자바에서 데이터를 통신할 때 Byte Stream을 통해서 InputStreamReader(바이트를 문자로 바꿔서 전달)로 여러개의 문자를 배열형태로 받아왔는데, 배열은 길이가 정해져있으므로 많이 설정하면 낭비가 되고, 적게 설정하면 문자가 버려지는 문제발생.

따라서 BufferedReader사용. 가변길이의 문자를 받을 수 있다.

JSP에선 데이터를 받을 때 request.getReader() 함수를 사용하면 BufferedReader일을 해줌.

데이터를 쓸 때는 PrintWriter 함수를 사용하면 BufferedWriter 일을 해줌. 문자열로 가변길이의 데이터를 쓸 수 있다.

어노테이션으로 사용가능

@ResponseBody -> BufferedWriter

@RequestBody -> BufferedReader

- 스프링은 계속 발전중이다.

<JPA란?>

- JPA는 Java Persistence API 이다.

Persistence(영속성) : 데이터를 생성한 프로그램의 실행이 종료되더라도 사라지지 않는 데이터의 특성을 의미. 영속성은 파일 시스템, 관계형 데이터베이스 혹은 객체 데이터베이스 등을 활용하여 구현한다.

API : Application Programming Interface

인터페이스 : 상하관계가 존재하는 약속

프로토콜 : 관계가 동등한 약속

수많은 프로토콜로 만들어진 ‘인터넷'

JPA

Java 자바 프로그래밍을 할 때

Persistence 영구적으로 데이터를 저장하기 위해 필요한

Application Programming Interface 인터페이스

- JPA는 ORM 기술이다.

-> Object-Relational Mapping

ORM -> 나의 하인!

자바는 클래스를 통해서 데이터베이스에 있는 테이블을 모델링해야 한다.

Class Team {

int id;

String name; → DB세상에 있는 데이터를 자바세상에 모델링한다!

String year;

}

**위 클래스를 실행하면 아래의 데이터베이스 테이블을 자동으로 만들어주는 것을 ORM**이라고 한다.

Team 테이블

id int

Name varchar

year varchar

- JPA는 반복적인 CRUD 작업을 생략하게 해준다.

Select, Delete, Update, Insert 등 굉장히 반복적으로 일어나는 일들.

위의 일들을 할 때 자바프로그램에서 DB에 커넥션을 요청함. -> DB가 요청자의 신분을 확인하고 세션열어줌. -> 자바에서 Connection을 가짐. -> 자바가 DB에 쿼리 전송 -> DB에서 Data를 자바프로그램에 응답 -> but, Data type이 서로 달라 자바프로그램은 해당 데이터를 자바 object로 변경해야함 -> 단순한 반복 로직..노가다..굉장히 귀찮음!

이런 일을 줄일 수 있는 것이 JPA!

1. 전송된 쿼리에 대한 응답이 있을 때, 데이터를 받고 받은 데이터를 자바 오브젝트로 바꾸고 연결된 세션을 끊고, 연결된 커넥션을 끊는 이런 모든 과정을 함수 하나로 가능하게 해줌!!

- JPA는 영속성 컨텍스트를 가지고 있다.

영속성 : 데이터를 영구적으로 저장하게 해주는 속성 -> 자바에서는 DB에 저장.

컨텍스트 : Context. 모든 정보를 가지고 있는 것.

길동 -> 영숙 : 난 영숙이 너의 모든 컨텍스트를 가지고 있어. == 영숙이의 모든것들을 알고있다!!

영숙 -> 길동 : 변태.

영속성 컨텍스트는 자바가 DB에 데이터를 저장하고, 가져오는 일련의 모든 과정에 중간에서 관여한다. 자바에서 DB로 바로 가는 게 아니라, 영속성 컨텍스트를 거쳐서 데이터가 처리된다. 자바, 영속성컨텍스트, DB는 서로 동기화 되어있다.

- JPA는 ORM을 통해 DB와 OOP의 불일치성을 해결하기 위한 방법론을 제공한다. (DB는 객체저장 불가능)

DB에선 객체저장이 불가능하므로 foreign key를 사용하지만, 자바에선 객체사용이 가능하므로, CRUD를 할때 JPA가 ORM을 통해 자동으로 객체를 매핑해준다.

OOP(Object-Oriented Programming) - 객체 지향 프로그래밍

- JPA는 OOP의 관점에서 모델링을 할 수 있게 해준다. (상속, 콤포지션, 연관관계)

Class Car extends EntityDate{

int id; (Primary key)

String name;

String color;

Engine engine;

}

Class Engine {

int id;

int power;

}

Class EntityDate extends EntityDate {

TimeStamp createDate;

TimeStamp updateDate;

}

Car 클래스에서 Engine을 사용하고 싶을때, 상속을 해서는 안된다. 부모 자식 관계는 아니기 때문에. 콤포지션(결합)을 써야한다.

시간과 같은 클래스에 공통으로 사용되는 객체가 필요하면, 새로운 객체를 만들어 상속하면 된다.

클래스를 통해 DB를 자동생성하므로 아래와 같이 생성됨.

Car

| id | name | color | engineId | createDate | updateDate |
| --- | --- | --- | --- | --- | --- |
| 1 | bmw | white | 2 |  |  |
| 2 | sonata | black | 1 |  |  |
|  |  |  |  |  |  |

Engine

| id | power | createDate | updateDate |
| --- | --- | --- | --- |
| 1 | 2000 |  |  |
| 2 | 4000 |  |  |

- 방언 처리가 용이하여 Migration 하기 좋음. 유지보수에도 좋음.

JPA는 DB에 붙어있는 추상화 객체를 사용해 mysql 뿐만 아니라, dialect(오라클, 마리아DB, MSsql, mysql, postsql) 등 다양한 방언들 또한 추상화 객체가 수시로 변경되어 처리하기 용이하다.

원래는 방언들 사이에 코드가 다르기 때문에 처리하기 불편하고 힘들다. but JPA는 쉽게 가능!

<스프링 부트 동작원리>

1. 내장 톰켓을 가진다.

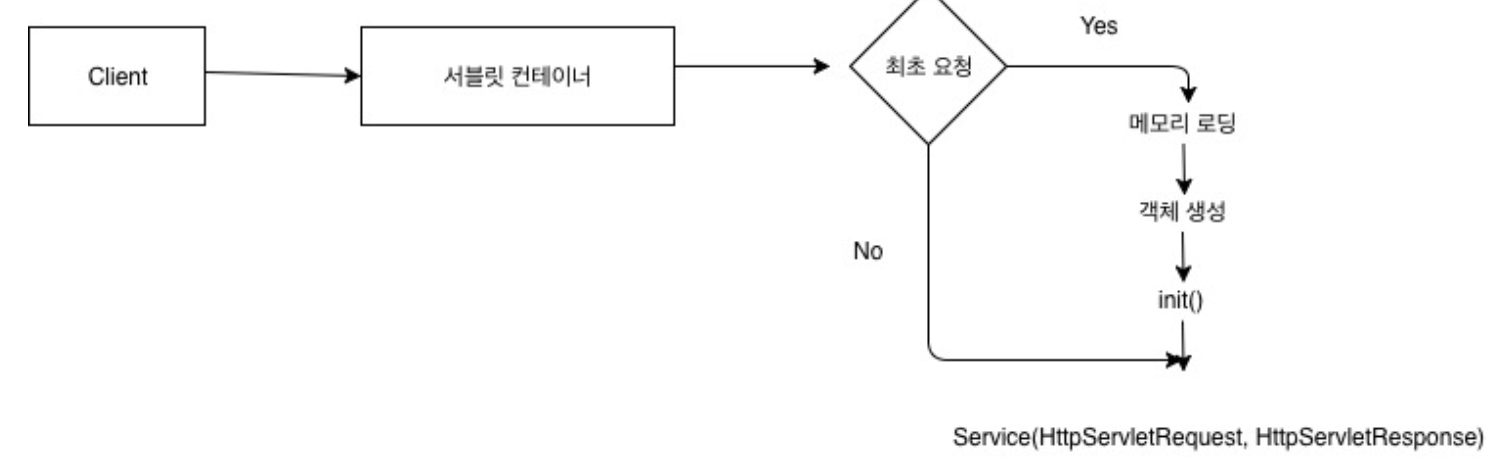
톰캣을 따로 설치할 필요 없이 바로 실행가능하다.

Socket : 운영체제가 가지고 있는 것!

아파치 : 요청한 파일을 응답해주는 것

톰캣 : 요청한 파일 중 자바 코드가 요청되면 그것을 컴파일하여 html로 돌려준다.

2. 서블릿 컨테이너(톰캣)



요청 -> 무조건 동작하는 게 아님.

정적인 파일(html,css,png 등) 요청하면 아파치가 응답. 톰캣 x

동적인 파일(자바) 요청하면 톰캣이 응답.

URL - Uniform Resource Locator : 위치 http://naver.com/a.png

URI - Uniform Resource Identifier : 식별자 http://naver.com/picture/a

스프링은 url 접근방식을 막아놈. uri로 요청해야함. 특정한 파일 요청을 할 수 없다.

-> 요청시에는 무조건 자바를 거쳐야한다. -> 톰캣이 수행한다.

자바관련된 것을 요청할 때 마다 서블릿 컨테이너에서 일어나는 일 :

1. 스레드 생성
2. 서블릿 객체 생성

스레드가 요청마다 만들어진다. 서블릿 객체는 하나로 사용한다. 최대 스레드 개수를 넘어가면 다음 요청들은 대기한다. 응답을 해주면 그 스레드와 서블릿객체는 대기중인 요청에 사용된다.

-> 속도가 빠름! 다중 스레드를 사용하고 지속적으로 재사용하기 때문.

3. web.xml

web.xml : 웹 어플리케이션의 배포 설명자로, 각 어플리케이션의 환경을 설정하는 역할을 한다. 서버가 처음 로딩될 때 읽어들이고, 해당 환경설정에 대해 tomcat에 적용하여 서버를 시작한다.

* ServletContext의 초기 파라미터 설정

- 하나의 웹어플에 적용할 수 있는 공통 변수라고 생각하면 된다. (웹 어플리케이션당 하나)

- 웹 어플당 하나라는 말은 웹 어플이 하나의 JVM에서 돌아갈 때 하는 말이다.

분산 웹 어플 환경이라면 JVM 마다 context가 다를수 있으므로 주의 요망

- web.xml 파일의 <context-param> 부분의 name 과 value 를 이용하여 셋팅한다.

- ex) getServletContext().getInitParameter(String);

출처: <https://ceomk.tistory.com/69> [The way of the future]

* Session의 유효시간 설정

사용자의 session 유효기간을 설정한다.

* Servlet/JSP에 대한 정의 및 매핑

정의 : Servlet/JSP의 위치나 정보가 정의되어있음.

매핑 : 요청한 자원, 데이터의 위치를 알려주고 이동할 수 있게 도와줌.

* Mime Type 매핑

Mime Type : 들고온 데이터의 type (text, image, video, audio, application 등)

Mime은 "Multipurpose Internet Mail Extensions"의 약자로 파일 변환을 위한 포맷이다. 이름에서도 알 수 있듯이 MIME은 이메일에 첨부된 파일을 텍스트 문자 형태로 변환해서 이메일과 함께 전송하기 위해 개발된 포맷이다. MIME 이름의 Internet Mail Extension에서 본래의 의도를 알 수 있다.

Mime Type에 맞게 매핑을 해준다. Type이 다른 곳으로 가면 에러!

* Welcome File list

아무 이유없이 들어온 애들을 보내는 곳. 관리자가 설정하기 나름이다.

* Error Pages 처리 및 설정

이상한 주소가 들어올때 Error Pages를 보냄

* 리스너/필터 설정

리스너 : 바쁜 문지기(web.xml)를 대신하여 관리자가 원하는 애를 빠르게 식별하여 처리하도록 도와줌.

필터 : 들어올 자격이 있는지 검증

* 보안

말 그래도 보안.

여기에서 Servlet/JSP 매핑시(web.xml에 직접 매핑 or @WebServlet 어노테이션 사용)에 모든 클래스에 매핑을 적용시키기에는 코드가 너무 복잡해지기 때문에 FrontController 패턴을 이용.

4. FrontController 패턴

최초 앞단에서 request 요청을 받아서 필요한 클래스에 넘겨준다. 왜? web.xml에 다 정의, 매핑하기가 너무 힘들기 때문.

요청이 URI 혹은 자바파일인 경우 바로 자원으로 이동못하고, 톰캣으로 이동

-> 톰캣에서 request(요청한 사람의 정보. 어떤 데이터를 들고왔는지, 요청하는지)와 response(응답해야할 데이터) 객체 자동 생성

-> 객체로 만들면 자바에서 request.함수or변수를 쉽게 사용가능

-> web.xml에서 일을 하지만 바쁘기 때문에, but 미리 선택한 특정주소가 들어오면 FrontController가 그 주소를 낚아 챔.

-> 낚아 채면 실제 자원에 접근하기 위해 내부에서 자원으로 request요청을 함. (외부에선 자원 접근 불가, 내부에서는 가능)

-> 이때 새로운 요청이 생기기 때문에 request와 response가 새롭게 new될 수 있다. 그래서 아래의 RequestDispatcher가 필요하다.

5. RequestDispatcher

필요한 클래스 요청이 도달했을 때 FrontController에 도착한 request와 response를 그대로 유지시켜준다. 웹페이지에서 페이지를 이동했을 때, 데이터를 가지고 이동하게 해준다.

6. DispatchServlet

FrontController 패턴을 직접짜거나 RequestDispatcher를 직접 구현할 필요가 없다. 왜냐면 스프링에는 DispatchServlet이 있기 때문! JSP에선 직접 짜야됨!

스프링의 DispatchServlet = FrontController + RequestDispatcher

DispatchServlet이 자동생성되어질 때 수 많은 객체가 생성(IoC)된다. 보통 필터들이다. 해당 필터들은 내가 직접 등록할 수 도 있고 기본적으로 필요한 필터들은 자동 등록 된다.

궁극적으로 하는 일 : 주소 분배

주소 분배를 하려면 각각의 클래스들이 메모리에 저장되어 있어야한다. 그래서 주소 분배를 하기 컴포넌트 스캔을 통해 Src 소스폴더 내의 자바파일에서 객체를 자동으로 메모리에 등록한다. IoC. 이때 자바파일에 어노테이션을 보고 메모리에 띄울지를 결정함. 보통 @Controller, RestController, Configuration, Repository, Service, Component가 붙으면 띄움.

7. 스프링 컨테이너

DispatchServlet에 의해 생성되는 수 많은 객체들은 어디에서 관리될까?

**첫째, ApplicationContext.**

수 많은 객체들이 ApplicationContext에 등록된다. 이것을 IoC(제어의 역전)라고 한다. 개발자가 직접 new를 통해 객체를 생성하게 된다면 해당 객체를 가리키는 레퍼런스 변수를 관리하기 어렵다. 그래서 스프링이 직접 해당 객체를 관리한다. 이때 우리는 주소를 몰라도 된다. 왜냐하면 필요할 때 DI하면 되기 때문이다. DI를 의존성 주입이라고 한다. 필요한 곳에서 ApplicationContext에 접근하여 필요한 객체를 가져올 수 있다. ApplicationContext는 싱글톤으로 관리되기 때문에 어디에서 접근하든 동일한 객체라는 것을 보장해준다.

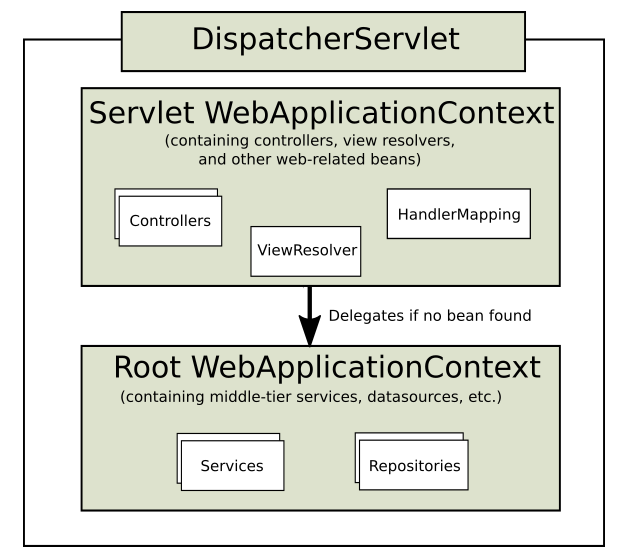
ApplicationContext의 종류에는 두가지가 있는데 root-applicationContext와 servlet-applicationContext 이다.

servlet-applicationContext는 ViewResolver, Interceptor, MultipartResolver 객체를 생성하고 **웹과 관련된 어노테이션** Controller, RestController를 스캔한다.

해당 파일은 DispatchServlet에 의해 실행된다.

root-applicationContext는 해당 어노테이션을 제외한 어노테이션 Service, Repository등을 스캔하고 DB관련 객체를 생성한다. 스캔이란 메모리에 로딩한다는 뜻.

해당파일은 ContextLoaderListener에 의해 실행된다. ContextLoaderListener를 실행해주는 녀석은 web.xml이기 때문에 root-applicationContext는 servlet-applicationContext보다 먼저 로드 된다. 당연히 servlet-applicationContext에서는 root-applicationContext가 로드한 객체를 참조할 수 있지만 그 반대는 불가능하다. 생성 시점이 root -> servlet 이기 때문이다.



**둘째, Bean Factory**

필요한 객체를 Bean Factory에 등록할 수 도 있다. 여기에 등록하면 초기에 메모리에 로드되지 않고 필요할 때 getBean()이라는 메소드를 통하여 호출하여 메모리에 로드할 수 있다. 지금은 메소드 위에 @Bean 쓰기만 하면 됨. 이것 또한 IoC이다. 그리고 필요할 때 DI하여 사용할 수 있다. ApplicationContext와 다른 점은 Bean Factory에 로드되는 객체들은 미리 로드되지 않고 요청자가 필요로할 때 호출하여 로드하기 때문에 lazy-loading이 된다는 점이다.

8. 요청 주소에 따른 적절한 컨트롤로 요청(Handler Mapping)

GET요청 => <http://localhost:8080/post/1>(uri) (post/1 : 식별자)

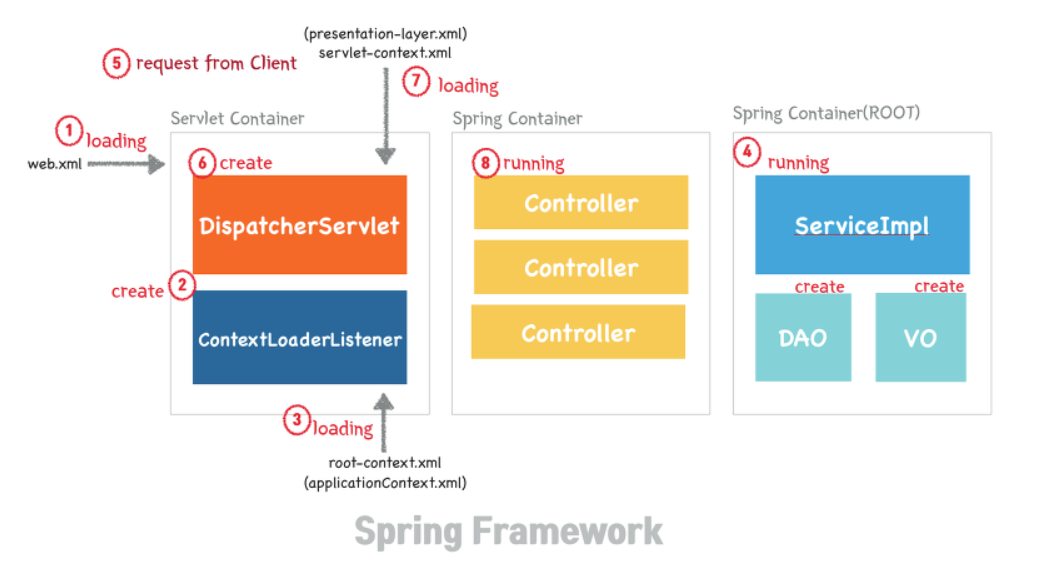
해당 주소 요청이 오면 적절한 컨트롤러의 함수를 찾아서 실행한다.

9. 응답

html파일을 응답할지 Data를 응답할지 결정해야 하는데 html 파일을 응답하게 되면 ViewResolver가 관여하게 된다.

하지만 Data를 응답하게 되면 MessageConverter가 작동하게 되는데 메시지를 컨버팅할 때 기본전략은 json이다.

요청 처리 로직



1. 톰캣 실행 시. web.xml이 자신이 해야할 일을 쫙 읽는다. 근데 할 일이 너무 많다…누가 분배해줄 순 없나..?라는 생각을 가지고 있음.
2. ContextLoaderListener create.
3. applicationContext가 읽어짐. -> root-context.xml도 같이 읽어짐. 보통 DB와 관련된 어떤 객체들을 컴포넌트 스캔을 해서 메모리에 올린다.
4. Spring Container(root) 내부에 DB관련된 애들을 메모리에 로드한다.
5. 사용자에게 request 받음.
6. DispatcherServlet이 FrontController 패턴으로 web.xml을 일을 분배해서 해줌.
7. servlet-context.xml을 읽고 웹과 관련된 애들을 메모리에 띄운다.
8. 이제 주소 분배를 하고 응답을 html파일로 할지 Data로 할지 결정한다.