*****절대경로: 경로의 처음, 즉 최상위 디렉토리부터 마지막까지 완전히 적힌 경로

*****상대경로: 현재 위치한 곳을 기준 (현재 디렉토리)를 기준으로 작성한 경로

***pageContext.request.contextPath**: pageContext 객체는 JSP에서 내장 되어 있는 객체인데 pageContext.request.contextPath를 사용함으로 써 상대 경로, 즉 내 위치에 따라 변하던 것을 알아서 위치를 찾아서 맞춰줌

*서버: 데이터 요구하면 데이터 보내주는 프로그램 (그냥 요청 처리해주는 기계)

*****동기**(Synchronous),** 비동기**(Asynchronous)**:

동기 (동시에 일어나는):
-> 요청과 그 결과가 동시에 일어난다는 약속임
-> 요청을 하면 시간이 얼마나 걸리던지 요청한 자리에 결과가 주어져야 함
-> 순서에 맞춰 진행되는 장점이 있지만, 여러 가지 요청을 동시에 처리할 수 없음

비동기 (동시에 일어나지 않는):
-> 요청과 결과가 동시에 일어나지 않을 거라는 약속임
-> 하나의 요청에 따른 응답을 즉시 처리하지 않아도, 그 대기 시간동안 또 다른 요청에 대해 처리 가능한 방식임
-> 여러 개의 요청을 동시에 처리할 수 있는 장점이 있지만 동기 방식보다 속도가 떨어질 수 있음

***AJAX (Asynchronous Javascript and XML)**: 비동기적으로 서버와 데이터 주고받는 자바스크립트 기술 (새로고침없이 서버에게 데이터 요청하는 자바스크립트 코드)

-> 페이지 이동 없음
-> 서버 처리를 기다리지 않고 비동기 요청이 가능함
-> 수신하는 데이터 양을 줄일 수 있음
-> XML, Plain, Text, JSON 등 다양한 포맷의 데이터를 주고 받을 수 있지만 일반적으로 사용이 편리한 JSON 형태로 주고 받을 수 있음

*__Annotation__: 자바에서 특별한 기능을 수행하도록 하는 기술임

***Mapping**: Key -> Value 데이터를 짝지어 저장하는 데이터 구조임

***Migration**: 다른 운영 환경으로 이동하는 것 (DB 이동, 데이터 이동)

*****가상 환경: 같은 프로그램이라도 버전별로 다른 프로그램이랑 호환성 문제가 생길 수 있음. 동시에 다른 버전 프로그램을 설치 할 수 없기 때문에 가상환경을 따로 만들어 실행시킴

-> 다른 운영체제를 사용해야하는 경우(Window에서 Mac, Mac에서 Linux 사용해야 하는경우)
-> 바이러스를 회피하기 위해 독립된 작업공간이 필요한 경우

*__@Qualifier__: 특정한 객체를 찾기 위한 이름을 지정함 (사용하면 어떤 빈을 주입 해야 하는지 문제를 없앨 수 있음)

*__@RequestBody, @RespondBody__: 웹에서 페이지 전환없이 이루어지는 동작들은 대부분 비동기 통신으로 이루어짐. 비동기 통신 위해서 클라이언트가 서버에 요청 메시지의 본문(Body)에 데이터 담아서 보내서 서버도 클라이언트에 응답을 하기 위해서는 메시지의 본문에 데이터를 담아서 보내야 하는데 이를 각각 @RequestBody @ResponseBody로 부름 그리고 Body에 담기는 데이터 형식은 JSON(JavaScript Object Notation)임

-> @RequestBody는 Body 안의 데이터(JSON 객체)를 자바 객체(VO)로 변환해 주는 Annotation

->@RespondBody는 자바 객체(VO)를 데이터(JSON 객체)로 바꿔 Body 안에 넣어주는 Annotation임

*__Mybatis__: 자바에서 데이터베이스를 쉽게 다룰 수 있도록 도와주는 오픈 소스 ORM (Object-Relational Mapping) 프레임워크임

->SQL 쿼리를 직접 작성할 수 있으므로 매우 유연함. 그리고 동적 쿼리를 작성 할 수 있음
->SQL 쿼리문 과 언어 코드를 분리하기 때문에 코드가 간결 및 유지보수에 용이함
->Mybatis는 캐시 기능을 제공하여 데이터베이스 연산 속도를 높일 수 있음

*__@Mapper__: @Mapper을 사용하면 Mybatis에서 인식할 수 있도록 도와줌. 이 인터페이스가 xml 파일과 연결되어 DB에 쿼리를 날리는 역할을 수행함

*모듈: 코드들이 작성되어 있는 하나의 파이썬 파일을 의미함

*패키지: 모듈의 집합

*라이브러리**:** 프로그램 개발을 쉽게 하기 위한 도구. 정상제어 (개발자가 모듈의 기능을 직접 호출함) 하는 모듈. 예를 들어 무언가를 자를 때 ‘도구’인 ‘가위’를 사용해서 ‘내가’ 직접 컨트롤함

*프레임워크**:** 프로그램 개발을 쉽게 하기 위한 도구. 제어 역전(모듈이 개발자가 만들 기능을 호출하는 것)이 발생하는 모듈. 다른 곳으로 이동할 때 ‘도구’인 비행기를 타고 이동하지만 ‘비행기’가 직접 컨트롤하고 나는 가만히 앉아 있어야 함

*__SessionStorage__: 데이터 저장의 목적이 아닌 휘발성 데이터를 저장하기 위한 목적으로는 SessionStorage, LocalStorage를 사용함. SessionStorage.setItem(key, value)f로 사용함

*클래스: 제품의 설계도
-> a class is like an object constructor, or a ‘blueprint’ for creating objects

*객체: 설계도로 만든 제품, 속성과 기능을 가진 프로그램 단위
-> an object is created from a class

*속성: 클래스안의 변수

*메서드: 클래스안의 함수

*생성자: 객체를 만들 때 실행되는 함수

->생성자는 필수 정보(파라미터)를 받고, 메모리를 할당해서 객체를 생성하는 책임을 가짐. 반면에 초기화는 이렇게 생성된 값들을 활용해서 어떠한 동작을 수행함
-> A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

*인스턴스: 메모리에 살아 있는 개체

*초기화: 처음으로 값을 지정함

*상속**(extends)**: 자식 클래스가 부모 클래스의 메서드 등을 상속받아 사용하며 자식 클래스에서 추가 및 확장을 할 수 있는 것을 말함

-> 재사용성이 이루어짐
-> 중복성의 최소화가 이루어짐

*구현**(implements)**: 부모 인터페이스를 자식 클래스에서 재정의하여 구현함. 상속과 달리 반드시 부모 클래스의 메서드를 재정의 하여 구현해야 함

->상속은 일반 클래스, abstract 클래스를 기반으로 구현하며, 구현은 인터페이스를 기반으로 구현함

***Client**: 둘 이상의 컴퓨터들이 통신할 때 호스트에 종속되어 통신하는 컴퓨터를 뜻함

***Web Server**: 클라이언트가 브라우저 주소창에 url을 입력하여 어떤 페이지를 요청하면 (HTTP 요청), http 요청을 받아 들여 HTML 문서와 같은 정적인 컨텐츠를 사용자에게 전달해줌 (HTTP 응답)

-> HTTP 기반으로 동작함
-> 정적 리소스 및 기타 부가기능 제공함
-> 정적(파일) HTML, CSS, JS, 이미지, 영상 주고 받음
-> 예) NGINX, APACHE

***Web Application Server**: Web Server와 동일하게 HTTP 기반으로 동작함. Web Server가 할 수 있는 기능 대부분이 처리 가능하며, 비즈니스 로직(서버사이드 코드)을 처리할 수 있어 사용자에게 동적인 컨텐츠를 전달 할 수 있음. 주로 데이터베이스 서버와 같이 수행함

-> HTTP 기반으로 동작함
-> 웹 서버 기능 포함 + (정적 리소스 제공 가능함)
-> 프로그램 코드를 실행해서 애플리케이션 로직 수행함

- 동적 HTML, HTTP API (JSON)
- 서블릿, JSP, 스프링 MVC

-> 톰캣(Tomcat), Jetty, Undertow
-> 웹 서버는 정적 리소스(파일), WAS는 애플리케이션 로직 담당함
-> 사실은 둘의 용어 경계가 모호함

- 웹 서버도 프로그램을 실행하는 기능을 포함하기도 함
- 웹 애플리케이션 서버도 웹 서버의 기능을 제공함

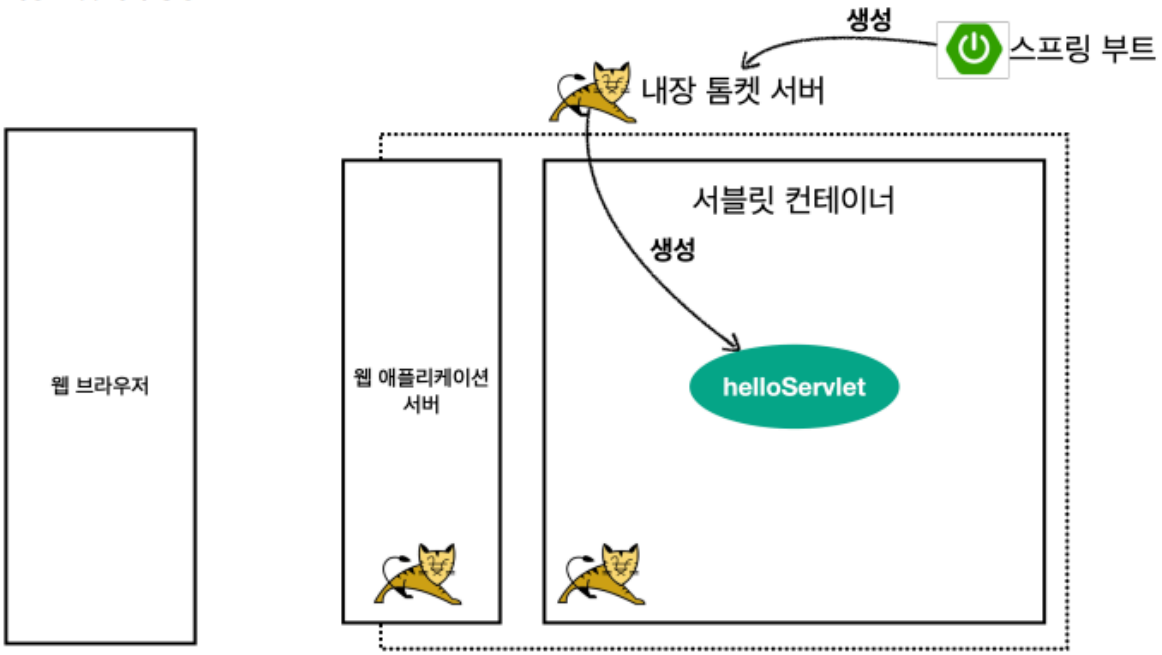
-> 자바는 서블릿 컨테이너 기능을 제공하면 WAS (서블릿 없이 자바코드를 실행하는 서버 프레임워크도 있음)
-> WAS는 애플리케이션 코드를 실행하는데 더 특화되었음

***Servlet**: 자바 웹 애플리케이션 서버의 구성 요소 중 동적인 처리를 프로그램 하는 역할임.

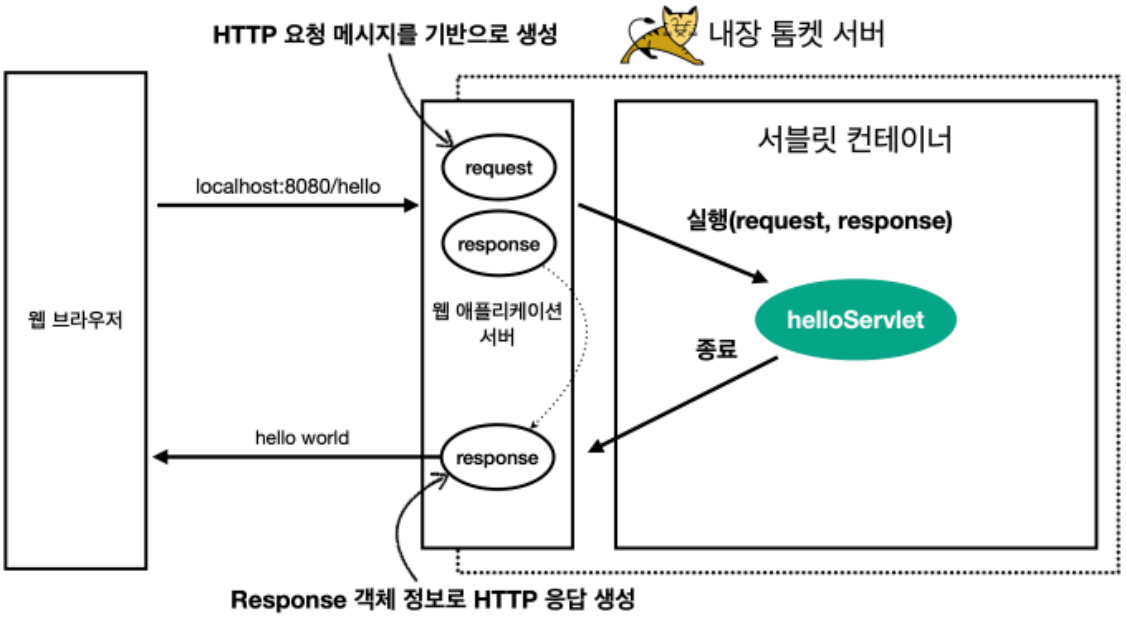
-> Dynamic Web Project로 웹을 만들 때 페이지에 대한 요청과 응답을 간단한 메소드 호출로 해결해 줄 수 있는 자바 라이브러리임. 웹페이지에 대한 클라이언트의 요청에 대한 응답을 메소드 호출로 간편하게 해줄 수 있음

-> 서블릿은 톰캣 같은 웹 애플리케이션 서버를 직접 설치하고 그 위에 서블릿 코드를 클래스 파일을 빌드해서 올린 다음 톰캣 서버를 실행하면 됨. 하지만 이 과정은 매우 번거로움
스프링 부트는 톰캣 서버를 내장하고 있으므로 톰캣 서버 설치 없이 편리하게 서블릿 코드를 실행할 수 있음

내장 톰캣 서버 생성



웹 애플리케이션 서버의 요청 응답 구조



특징
-> urlPatterns의 URL이 호출되면 서블릿 코드가 실행 됨
-> HTTP 요청 정보를 편리하게 사용할 수 있는 HttpServletRequest
-> HTTP 응답 정보를 편리하게 제공할 수 있는 HttpServletResponse
-> 개발자는 HTTP 스펙을 매우 편리하게 사용함

HTTP 요청시
-> WAS는 Request, Response 객체를 새로 만들어서 서블릿 객체 호출함
-> 개발자는 Request 객체에서 HTTP 요청 정보를 편리하게 꺼내서 사용함
-> 개발자는 Response 객체에 HTTP 응답 정보를 편리하게 입력함
-> WAS는 Response 객체에 담겨있는 내용으로 HTTP 응답 정보를 생성함

-> a small, server-resident program that typically runs automatically in response to user input
-> WAS에 동작하는 JAVA 클래스임
-> HttpServlet 클래스를 상속하여 구현함
-> HTML은 JSP에서 구현하고(정적인 것) 복잡한 프로그래밍은 서블릿으로 구현해야 함(동적인 것). Servlet은 해당 프로젝트에서 MVC 패턴의 Controller를 담당함

*서블릿 컨테이너 **(Servlet Container)**:

-> 톰캣처럼 서블릿을 지원하는 WAS를 서블릿 컨테이너라고 함
-> 서블릿 컨테이너는 서블릿 객체를 생성, 초기화, 호출, 종료하는 생명주기 관리함
-> 서블릿 객체는 싱글톤으로 관리
- 고객의 요청이 올 때 마다 계속 객체를 생성하는 것은 비효율적임
- 최초 로딩 시점에 서블릿 객체를 미리 만들어주고 재활용함
- 모든 고객 요청은 동일한 서블릿 객체 인스턴스에 접근함
- 공유 변수 사용 주의해야 함
- 서블릿 컨테이너 종료시 함께 종료함

-> JSP도 서블릿으로 변환 되어서 사용함
-> 동시 요청을 위한 멀티 쓰레드 처리 지원함

-> 컨테이너는 인스턴스의 생명주기를 관리하며, 생성된 인스턴스에게 추가적인 기능을 제공함. 예를 들어 Servlet을 실행해주는 WAS는 Servlet 컨테이너를 가지고 있음. WAS는 웹 브라우저로부터 서블릿 URL에 해당하는 요청을 받으면 서블릿을 메모리에 올린 후 실행함. 개발자가 서블릿 클래스를 작성했지만 실제로 메모리에 올리고 실행하는 것은 WAS가 가지고 있는 Servlet 컨테이너임. Servlet 컨테이너는 동일한 서블릿에 해당하는 요청을 받으면, 또 메모리에 올리지 않고 기존에 메모리에 올라간 서블릿을 실행하여 그 결과를 웹 브라우저에 전달함

*프로그램**(Programme) vs** 프로세스**(Process)**: 우리가 프로그램을 실행하려고 하면 실행을 위해 운영체제에서 메모리 공간을 할당받아오게 되며 그 공간에 프로그램이 올려져 실행되게 됨. 즉 프로세스는 실행중인 프로그램(CPU 스케줄링의 대상이 되는 작업), 혹은 그 작업이라고 함

-> 간단히 말해 프로그램은 저장장치에 저장되어있는 정적인 상태이고 프로세스는 실행을 위해 메모리에 올라와있는 동적인 상태임
-> 프로그램이 메모리에 올라가면 프로세스가 되는 인스턴스화가 일어나고 이후 운영체제의 CPU 스케줄러에 따라 CPU가 프로세스를 실행함

*****쓰레드**(Thread)**: a thread is a component of a process.

-> 프로세스 내에서 실행 가능한 가장 작은 단위, 프로세스는 여러 스레드를 가질 수 있음

멀티 쓰레딩: 프로세스 내 작업을 여러 개의 스레드, 멀티쓰레드로 처리하는 기법이며 쓰레드끼리 서로 자원을 공유하기 때문에 효율성이 높음. 예를 들어 웹 요청을 처리할 때 새 프로세스를 생성하는 대신 스레드를 사용하는 웹 서버의 경우 훨씬 적은 리소스를 소비하며 한 스레드가 중단(blocked) 되어도 다른 스레드는 실행(running) 상태일 수 있기 때문에 중단되지 않은 빠른 처리가 가능함. 또한 동시성에도 큰 장점이 있음 하지만 한 쓰레드에 문제가 생기면 다른 쓰레드에도 영향을 끼쳐 쓰레드로 이루어져 있는 프로세스에 영향을 끼칠 수 있음

WAS의 멀티 쓰레드 지원

-> 멀티 쓰레드에 대한 부분은 WAS가 처리함
-> 개발자가 멀티 쓰레드 관련 코드를 신경쓰지 않아도 됨
-> 개발자는 마치 싱글 쓰레드 프로그래밍을 하듯이 편리하게 소스 코드를 개발함
-> 멀티 쓰레드 환경이므로 싱글톤 객체(서블릿, 스프링 빈)는 주의해서 사용함

***HTTP(Hypertext Transfer Protocol)**: 클라이언트와 서버가 인터넷상에서 데이터를 주고받기 위한 프로토콜임

-> 연결을 끊어버리기 때문에 클라이언트의 이전 상황을 알 수 없음. 이러한 상태를 무상태라고 함. 이러한 특징 때문에 정보를 유지하기 위해서 Cookie와 같은 기술이 등장함

-> 요청 메소드: GET, POST, PUT, DELETE 등이 있음

GET: 정보를 요청하기 위해서 사용함 (Select)
-> 리소스를 조회함
-> 서버에 전달하고 싶은 데이터 query(쿼리 파라미터, 쿼리 스트링)를 통해서 전달함
-> 메시지 바디를 사용해서 데이터를 전달할 수 있지만 지원하지 않는 곳이 많아서 권장하지 않음

POST: 정보를 밀어넣기 위해서 사용함 (Insert)
-> 요청한 데이터 처리함
-> 메시지 바디를 통해서 서버로 요청 데이터 전달함
-> 서버는 요청 데이터를 처리함 (메시지 바디를 통해 들어온 데이터를 처리하는 모든 기능을 수행함)
-> 주로 전달된 데이터로 신규 리소스 등록, 프로세스 처리에 사용함
-> Insert라는 의미보다는 그 리소스 상황에 따라 요청에 포함된 표현을 처리하도록 함

PUT: 정보를 업데이트하기 위해서 사용함 (없으면 생성함) (Update)
-> 리소스가 있으면 대체하고 없으면 생성함 (덮어씀)
-> 클라이언트가 리소스 위치를 알고 URI 지정함 (/members/100)

PATCH: 정보 부분을 변경을 하기 위해 사용함 (Update)
-> 리소스 부분 변경

DELETE: 정보를 삭제하기 위해서 사용함 (Delete)
-> 리소스 제거

HTTP 메서드 속성

1) 안전한 메서드 (Safe Methods)

-> 호출해도 리소스를 변경하지 않음 (서버에 데이터를 변경하지 않는 요청 메서드, 안전한 메서드)
GET

-> 서버에 데이터를 변경하는 요청 메서드 (안전하지 않은 메서드)
POST
PUT
DELETE
PATCH

2) 멱등성 (Idempotent)

-> 연산을 여러 번 적용하더라도 결과가 달라지지 않는 성질
-> 멱등은 외부 요인으로 중간에 리소스가 변경되는 것까지 고려하지 않음

3) 캐시가능 (Cacheable)

캐시: 서버 지연을 줄이기 위해 웹 페이지, 이미지, 기타 유형의 웹 멀티미디어 등의 웹 문서들을 임시 저장하기 위한 정보기술임

-> 응답 결과 리소스를 캐시해서 사용해도 되는가?

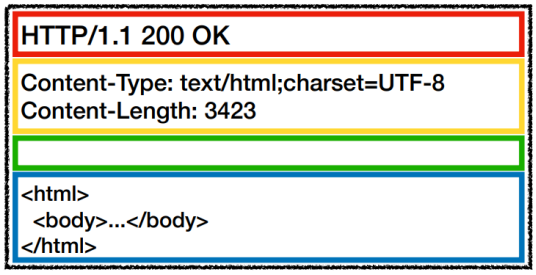
HTTP 메세지에 모든 것을 전송함

- HTML, TEXT
- IMAGE, 음성, 영상, 파일
- JSON, XML (API)
- 거의 모든 형태의 데이터 전송 가능함
- 서버 간에 데이터를 주고 받을 때도 대부분 HTTP 사용함

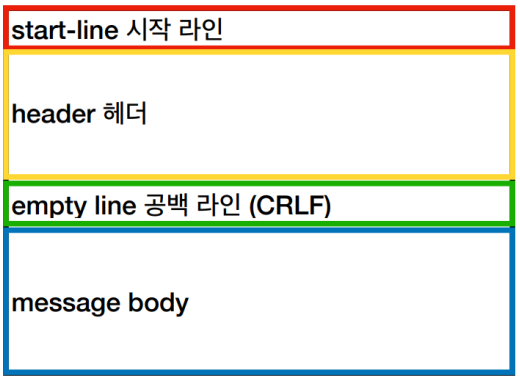
HTTP 메시지 구조

예) HTTP 요청 메시지

요청 메시지도 body 본문을 가질 수 있음



예) HTTP 응답 메시지

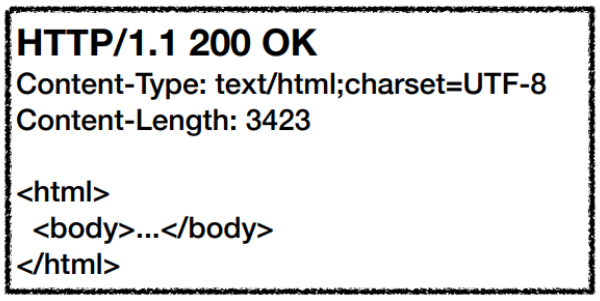


HTTP 메시지 구조

start-line = request-line 아니면 status-line

request-line = method SP(공백) request-target SP(공백) HTTP-version CRLF(엔터)

-> HTTP 메서드 (GET: 조회)
-> 요청 대상 (/search?q=hello&hi=ko)
-> HTTP Version

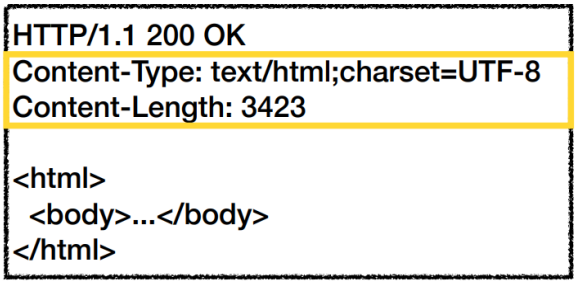


status-line = HTTP-version SP status-code SP reason-phrase CRLF

-> HTTP 버전
-> HTTP 상태 코드: 요청 성공, 실패를 나타냄
200: 성공
400: 클라이언트 요청 오류
500: 서버 내부 오류
-> 이유 문구: 사람이 이해할 수 있는 짧은 상태 코드 설명 글

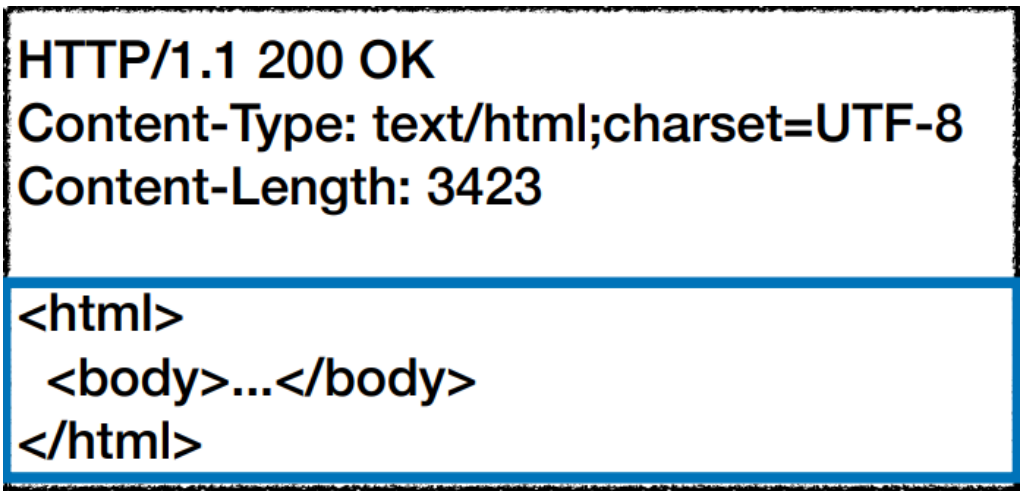
HTTP Header

-> header-field = field-name ":" OWS field-value OWS (OWS: 띄어쓰기 허용함)
-> field-name은 대소문자 구분 없음



-> HTTP 전송에 필요한 모든 부가정보가 포함 됨
-> 필요시 임의의 헤더 추가 가능함 (예: helloworld: hihi)

HTTP Message Body



-> 실제 전송할 데이터임
-> HTML 문서, 이미지, 영상, JSON 등등 byte로 표현할 수 있는 모든 데이터 전송 가능함
-> 요청 파라미터와 다르게 HTTP 메시지 바디를 통해 데이터가 직접 넘어오는 경우는 @RequestParam, @ModelAttribute를 사용할 수 없음 (물론 HTML Form 형식으로 전달되는 경우는 요청 파라미터로 인정됨)

***HTTP API** 설계

1) 문서(Document)
-> 단일 개념(파일 하나, 객체 인스턴스, 데이터베이스 row)
-> 예) members/100, /files/star.jpg

2) 컬렉션(Collection) - POST 기반 등록
-> 서버가 관리하는 리소스 디렉토리
-> 서버가 리소스의 URI를 생성하고 관리함
-> 클라이언트는 등록될 리소스의 URI를 모름
- 회원등록/members -> POST
- POST/members

-> 서버가 새로 등록된 리소스 URI를 생성해줌
- HTTP/1.1 201 Created
  Location: /members/100

-> 여기서 컬렉션은 /members
-> 예) /members

3) 스토어(Store) - PUT 기반 등록
-> 클라이언트가 관리하는 자원 저장소
-> 클라이언트가 직접 리소스의 URI를 지정함
-> 클라이언트가 리소스의 URI를 알고 있어야 함
- 파일 등록 /files/{filename} -> PUT
- PUT/files/star.jpg

-> 여기서 스토어는 /files
-> 예) /files

4) 컨트롤러(Controller), 컨트롤 URI
-> HTML FORM은 GET, POST만 지원하므로 제약이 있음
-> 이런 제약을 해결하기 위해 동사로 된 리소스 경로 사용함
-> POST의 /new, /edit, /delete가 컨트롤 URI임
-> HTTP 메서드로 해결하기 애매한 경우 사용함 (HTTP API 포함)
-> 문서, 컬렉션, 스토어로 해결하기 어려운 추가 프로세스 실행함
-> 예) /members/{id}/delete

***HTTP** 상태코드: 클라이언트가 보낸 요청의 처리 상태를 응답에서 알려주는 기능

-> 1xx(Informational): 요청이 수신되어 처리 중
-> 2xx(Successful): 요청 정상 처리함
-> 3xx(Redirection): 요청을 완료하면 추가 행동이 필요함
-> 4xx(Client Error): 클라이언트 오류, 잘못된 문법 등으로 서버가 요청을 처리할 수 없음
-> 5xx(Server Error): 서버 오류, 서버가 정상 요청을 처리하지 못 함

***모델(Model)**: the internal representations of information

-> Model 객체는 Controller에서 생성된 데이터를 담아 View로 전달할 때 사용하는 객체임. Servlet의 request.setAttribute() 와 비슷한 역할을 함. addAttribute(“key”, “value”) 메서드를 이용해 view에 전달할 데이터를 key, value 형식으로 전달할 수 있음

***URL(Uniform Resource Locator)**: 인터넷 상의 자원의 위치. 특정 웹 서버의 특정 파일에 접근하기 위한 경로 혹은 주소

***web.xml**: Web Application의 설정을 위한 deployment descriptor(배포 설명자)임. 즉 Web Application의 설정파일임.

-> Tomcat과 같은 서블릿 컨테이너에 웹 어플리케이션을 배포하는 방법을 설명함. 즉 Web Application을 실행시킬 때 함께 올라가야할 설정(설명)등을 정의해놓음.
->DispatcherServlet, ContextLoaderListener, Filter 설정함

***Scope**: 프로그램에서 변수들은 사용 가능한 범위를 가지는데 이 변수 범위를 Scope라고 함

Page Scope: 페이지 내에서 지역변수처럼 사용함
-> JSP에서 pageContext라는 내장 객체를 사용함

Request Scope: HTTP 요청을 WAS가 받아 웹 브라우저에 응답할 때까지 변수값을 유지함
-> HTTPServletRequest 객체를 사용함
-> JSP에서는 request 내장 변수를 사용하고 Servlet에서는 HTTPServletRequest 객체를 사용함
-> forward 시 값을 유지하고자 함

Session Scope: 웹 브라우저별로 변수를 관리하고자 할 경우 사용함
-> JSP에서는 session 내장 변수를 사용하고 Servlet에서는 HttpServletRequest의 getSession() 메소드를 이용해 session 객체를 얻음

Application Scope: 웹 어플리케이션이 시작되고 종료될 때까지 변수를 사용함
-> JSP에서는 application 내장 객체를 이용함. Servlet에서는 getServletContext() 메소드를 이용하여 application 객체를 이용함

scope의 공통함수
-> setAttribute(String key, Object Value): key, value 형식으로 값을 할당함
-> getAttribute(String key): key 값으로 value 값을 리턴 받음

***쿠키(Cookie) vs 세션(Session) vs 토큰(Token):**

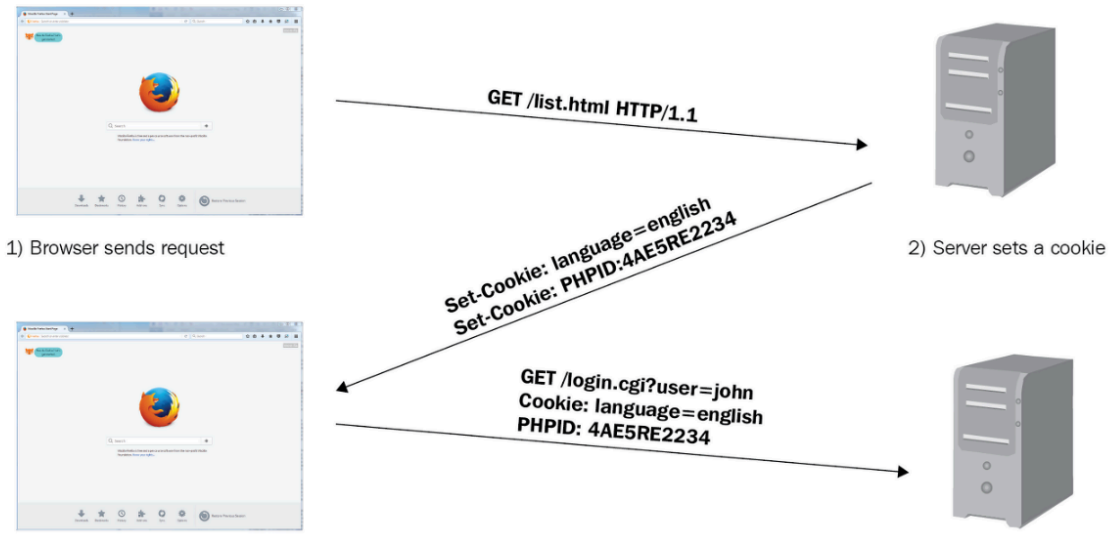
서버가 클라이언트 인증을 확인하는 방식은 쿠키, 세션, 토큰 3가지 방법이 있음

****쿠키(Cookie): Key-Value 형식의 문자열임. 클라이언트가 어떠한 웹사이트를 방문할 경우, 그 사이트가 사용하고 있는 서버를 통해 클라이언트의 브라우저에 설치되는 작은 기록 정보 파일임. 각 사용자마자 브라우저에 정보를 저장하니 고유 정보 식별이 가능함

▼ Request Headers

:authority: www.geeksforgeeks.org
:method: GET
:path: /http-headers-cookie/
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.9
cache-control: max-age=0
cookie: G_ENABLED_IDPS=google; _ga=GA1.2.236891924.1569526010; __gads=ID=f8deb276b85d6f74:T=1569559579:S=ALNI_Ma9kGxkZVOd23UT2B6UIi0-iHksPw; __utmz=245605906.1569597787.4.4.utmcsr=google|utmccn=(organic)|utmcmd=organic|utmctr=(not%20provided); __utma=245605906.236891924.1569526010.1569592113.1569597786.4; geeksforgeeks_consent_status=dismiss; gfguserName=Sabya_Samadder%2FeyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczpcL1wvd3d3LmdlZWtzZm9yZ2Vla3Mub3JnXC8iLCJpYXQiOjE1NzIzNDM3NjAsImV4cCI6MTU3NDkzNTc2MCwiaGFuZGxlIjoiU2FieWFfU2FtYWRkZXIiLCJ1dWlkIjoiYjA2NzA0Njc3MDMzMmY4Y2EyMDcxMDM4OWJjMWVkNjEifQ.f5Bky9sW46uX53XGJupbTHQPhSvjgr9_MCV5whZFzkBSCEZR_n4w-5Imj5eSa7TXuFS1r6NI1_VRnuz7Au0P_H-u6SEATsOVSkhssEMx0L6oj5NDQw1jk3ZAreK7dk_xyRLgnHsTJws40GbLi9__Yirrp9q2BNGztaMVTtXsqrW9knMAsOVKNmhEGM7bh9fpsEYh3PqKRRag4WI1dGRaZZ6Y-orBA91Srj9oyzqYOOFK3zmXd9pHKW7b_ffH5sheGW2EM7uwtjoMiGA7oc6RuG0G8sdpPPYL6Ktfkai2g_oHPRahoRsZ_UUQT3jNY91bHt75gxi2PMWw6KOUsNcujA; wordpress_test_cookie=WP+Cookie+check; AKA_A2=A
referer: https://www.google.com/
sec-fetch-mode: navigate
sec-fetch-site: cross-site
sec-fetch-user: ?1
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/77.0.3865.120 Safari/537.36

***Cookie 인증 방식



1) 클라이언트가 서버에 요청을 보냄

2) 서버는 클라이언트의 요청에 대한 응답을 작성할 때, 클라이언트 측에 저장하고 싶은 정보를 응답 헤더의 Set-Cookie에 담음

3) 이후 클라이언트는 요청을 보낼 때마다, 매번 저장된 쿠키를 요청 헤더의 Cookie에 담아 보낸다. 서버는 쿠키에 담긴 정보를 바탕으로 해당 요청의 클라이언트가 누군지 식별하거나 정보를 바탕으로 추천 광고를 띄움

***쿠키 단점

1) 가장 큰 단점은 보안에 취약함 (요청 시 쿠키의 값을 그대로 보내기 때문에 유출 및 조작 당할 위험이 존재함)

2) 쿠키에는 용량 제한이 있어 많은 정보를 담을 수 없음

3) 웹 브라우저마다 쿠키에 대한 지원 형태가 다르기 때문에 브라우저간 공유가 불가능함

4) 쿠키의 사이즈가 커질수록 네트워크에 부하가 심해짐

***쿠키와 보안 문제

1) 쿠키 값은 의미로 변경할 수 있다

- 클라이언트가 쿠키를 강제로 변경하면 다른 사용자가 된다
- 실제 웹브라우저 개발자모드 -> Application -> Cookie 변경으로 확인
- Cookie: memberId=1 -> Cookie: memberId=2 (다른 사용자들의 이름이 보임)

2) 쿠키에 보관된 정보는 훔쳐갈 수 있음

- 만약 쿠키에 개인정보나 신용카드 정보가 있다면?
- 이 정보가 웹 브라우저에도 보관되고, 네트워크 요청마다 계속 클라이언트에서 서버로 전달 됨
- 쿠키의 정보가 나의 로컬 PC에서 털릴 수도 있고 네트워크 전송 구간에서 털릴 수도 있음

3) 해커가 쿠키를 한 번 훔쳐가면 평생 사용할 수 있음

- 해커가 쿠키를 훔쳐가서 그 쿠키로 악의적인 요청을 계속 시도할 수 있음

***대안
1) 쿠키에 중요한 값을 노출하지 않고 사용자 별로 예측 불가능한 임의의 토큰(랜덤 값)을 노출하고 서버에서 토큰과 사용자 id를 매핑해서 인식함. 그리고 서버에서 토큰을 관리함

2) 토큰은 해커가 임의의 값을 넣어도 찾을 수 없도록 예상 불가능 해야 함

3) 해커가 토큰을 털어가도 시간이 지나면 사용할 수 없도록 서버에서 해당 토큰의 만료시간을 짧게(예: 30분) 유지한다. 또는 해킹이 의심되는 경우 서버에서 해당 토큰을 강제로 제거하면 됨

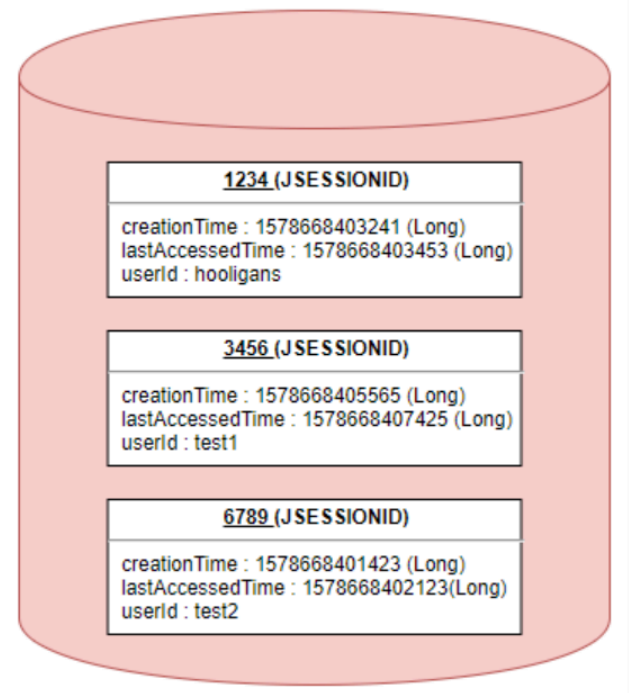
**Session 인증

이러한 쿠키의 보안적 이슈 때문에, 세션은 비밀번호 등 클라이언트의 민감한 인증 정보를 브라우저가 아닌 서버 측에 저장하고 관리함. 서버의 메모리에 저장하기도 하고 서버의 로컬 파일이나 데이터베이스에 저장하기도 함. 즉 민감한 정보는 클라이언트에 보내지 말고 서버에서 모두 관리함

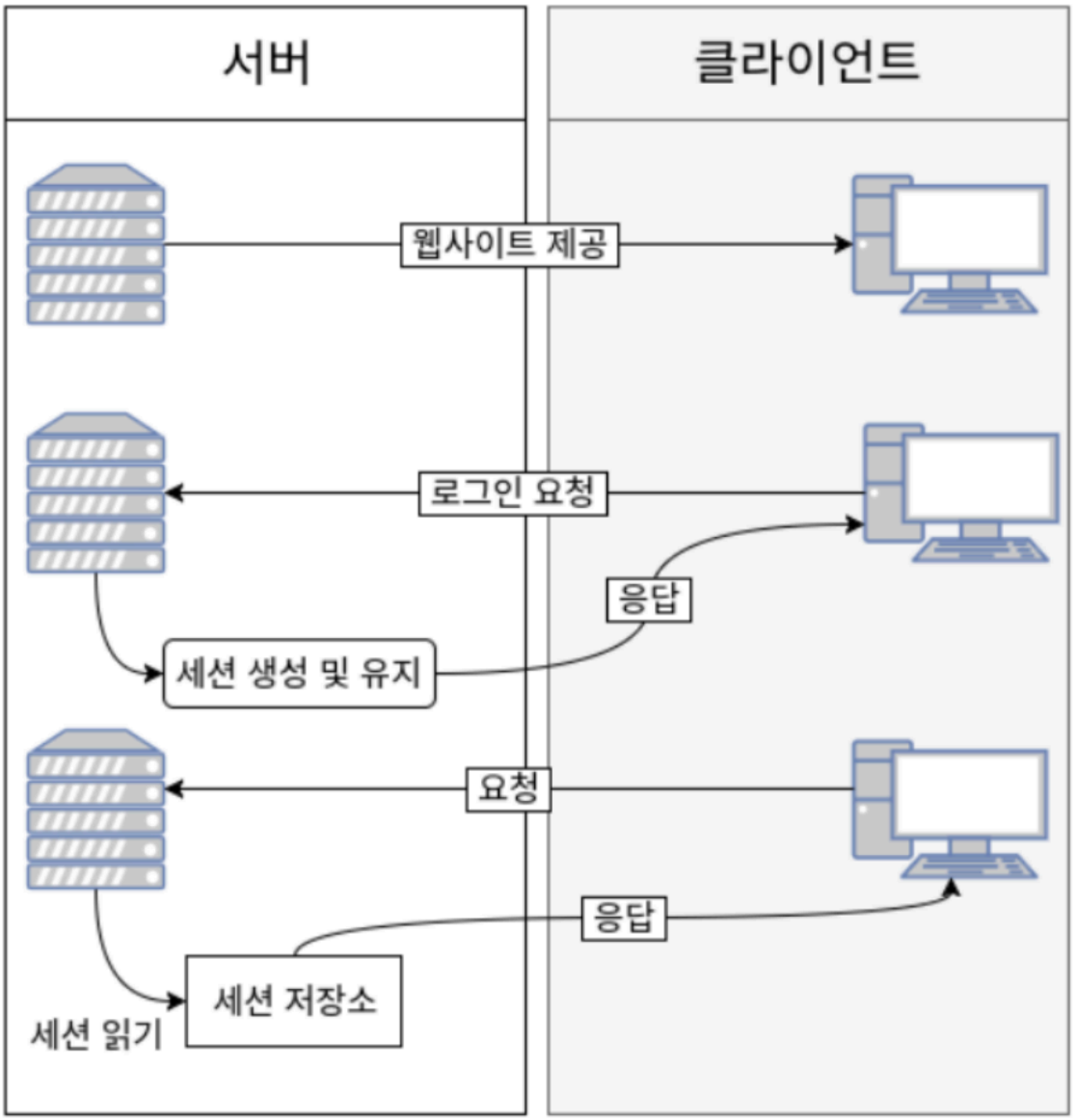
세션 객체는 어떤 형태로 이루어져 있을까 ?

세션 객체는 Key에 해당하는 SESSION ID와 이에 대응하는 Value로 구성되어 있다.

Value에는 세션 생성 시간, 마지막 접근 시간 및 User가 저장한 속성 등 이 Map 형태로 저장된다.



Session 인증 방식



1) 유저가 웹사이트에 로그인하면 세션이 서버 메모리(혹은 데이터베이스) 상에 저장됨. 이때 세션을 식별하기 위한 Session Id를 기준으로 정보를 저장함

2) 서버에서 브라우저에 쿠키에다가 Session Id를 저장함

3) 쿠키에 정보가 담겨있기 때문에 브라우저는 해당 사이트에 대한 모든 Request에 Session Id를 쿠키에 담아 전송함

4) 서버는 클라이언트가 보낸 Session Id와 서버 메모리로 관리하고 있는 Session id를 비교하여 인증을 수행함

Session 방식의 단점

1) 쿠키를 포함한 요청이 외부에 노출되더라도 Session ID 자체는 유의미한 개인정보를 담고 있지 않음. 그러나 해커가 세션 ID 자체를 탈취하여 클라이언트인척 위장할 수 있다는 한계가 존재함 (이는 서버에서 IP 특성을 통해 해결 할 수 있음)

2) 서버에서 세션 저장소를 사용하므로 요청이 많아지면 서버가 부하에 심해짐

**Token 인증

토큰 기반 인증 시스템은 클라이언트가 서버에 접속을 하면 해당 클라이언트에게 인증되었다는 의미로 '토큰'을 부여함. 이 토큰은 유일하며 토큰은 토큰을 발급받는 클라이언트는 또 다시 서버에 요청을 보낼 때 요청 헤더에 토큰을 심어서 보냄. 그러면 서버에서는 클라이언트로부터 받은 토큰을 서버에게 제공한 토큰과의 일치 여부를 체크하여 인증 과정을 처리하면 됨

기존의 세션기반 인증은 서버가 파일이나 데이터베이스에 세션정보를 가지고 있어야 하고 이를 조회하는 과정이 필요하기 때문에 많은 오버헤드가 발생함. 하지만 토큰은 세션과는 달리 서버가 아니 클라이언트에 저장되기 때문에 메모리나 스토리지 등을 통해 세션을 관리했던 서버의 부담을 덜 수 있음. 토큰 자체에 데이터가 들어있기 때문에 클라이언트에서 받아 위조되었는지 판별만 하기 되기 때문임

토큰은 앱과 서버가 통신 및 인증할 때 가장 많이 사용함. 왜냐하면 웹에는 쿠키와 세션이 있지만 앱에서는 없음

[서버 기반 vs 토큰 기반]

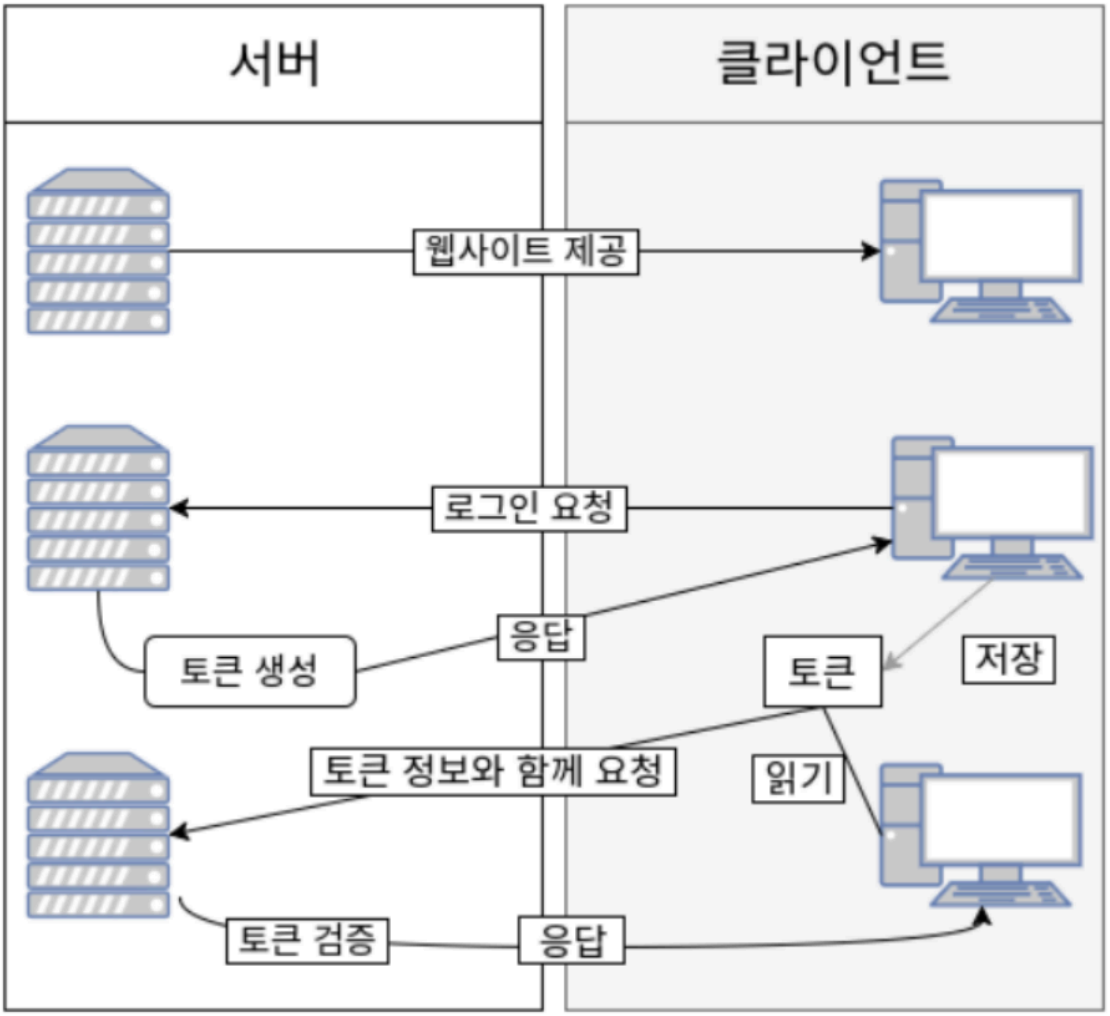
서버(세션) 기반 인증 시스템

서버의 세션을 사용해 사용자 인증을 하는 방법으로 서버측(서버 램 or 데이터베이스)에서 사용자의 인증 정보를 관리하는 것을 의미한다.
그러다 보니, 클라이언트로부터 요청을 받으면 클라이언트의 상태를 계속에서 유지해놓고 사용한다.
(Stateful) 이는 사용자가 증가함에 따라 성능의 문제를 일으킬 수 있으며 확장성이 어렵다는 단점을 지닌다.

토큰 기반 인증 시스템

이러한 단점을 극복하기 위해서 "토큰 기반 인증 시스템"이 나타났다.
인증받은 사용자에게 토큰을 발급하고, 로그인이 필요한 작업일 경우 헤더에 토큰을 함께 보내 인증받은 사용자인지 확인한다.
이는 서버 기반 인증 시스템과 달리 상태를 유지하지 않으므로 Stateless 한 특징을 가지고 있다.

Token 인증 방식



1) 사용자가 아이디와 비밀번호로 로그인을 함

2) 서버 측에서 사용자(클라이언트)에게 유일한 토큰을 발급함

3) 클라이언트는 서버 측에서 전달받은 토큰을 쿠키나 스토리지에 저장해 두고 서버에 요청을 할 때마다 해당 토큰을 HTTP 요청 헤더에 포함시켜 전달함

4) 서버는 전달 받은 토큰을 검증하고 요청에 응답함. 토큰에는 요청한 사람의 정보가 담겨있기에 서버는 DB를 조회하지 않고 누가 요청하는지 알 수 있음

Token의 단점

1) 쿠키/세션과 다르게 토큰 자체의 데이터 길이가 길어 인증 요청이 많아질수록 네트워크 부하가 심해질 수 있음

2) Payload 자체는 암호화되지 않기 때문에 유저의 중요한 정보는 담을 수 없음

3) 토큰을 탈취당하면 대처하기 어려움 (따라서 사용 기간 제한을 설정하는 식으로 극복함)

웹 프로그램은 웹 브라우저 요청 -> 서버 응답 순서로 실행되며 서버 응답이 완료되면 웹 브라우저와 서버 사이의 네트워크 연결은 끊어진다. 하지만 수 많은 브라우저가 서버에 요청할 때마다 매번 새로운 세션이 생성되는 것이 아니라 동일한 브라우저의 요청에서 서버는 동일한 세션을 사용함. 그렇다면 서버는 도대체 어떻게 웹 브라우저와 연결고리(세션)을 맺는걸까? 그 해답은 쿠키에 있음.

쿠키는 서버가 웹 브라우저에 발행하는 값임. 웹 브라우저가 서버에 어떤 요청을 하면 서버는 쿠키를 생성하여 전송하는 방식으로 응답함. 그러면 웹 브라우저는 서버에서 받는 쿠키를 저장함. 이후 서버에 다시 요청을 보낼때는 저장한 쿠키를 HTTP 헤더에 담아서 전송함. 그러면 서버는 웹 브라우저가 보낸 쿠키를 이전에 발행했던 쿠키값과 비교하여 같은 웹 브라우저에서 요청한 것인지 아닌지를 구분할 수 있음. 이때 세션은 바로 쿠키 1개당 생성되는 서버의 메모리 공간이라 할 수 있음

요청에 대한 응답을 하고 나면 클라이언트와 서버 연결이 지속되지 않음. 상태 유지를 하기 위해서 Cookie와 Session이 등장함

쿠키(Cookie): 서버가 어떤 데이터를 개인 PC에 저장한 후 다시 그 데이터를 받아 오는 기술 혹은 그 데이터 자체임

-> 서버에서 클라이언트의 브라우저로 전송되어 사용자 컴퓨터에 저장. 저장된 쿠키는 다시 해당하는 웹 페이지에 접속할 때 브라우저에서 서버에서 쿠키를 전송함
-> set-cooke: sessionId=abcde1234; expires= Sat, 26-Dec-2020 00:00:00 GMT; path=/; domain=.google.com; Secure

-> 저장된 정보를 다른 사람 또는 시스템이 볼 수 있는 단점이 있음
-> 유효시간이 지나면 사라짐
-> 팝업창을 통해 '오늘 이 창을 다시 보지 않기' 체크가 좋은 예시임
-> 쿠키 정보는 항상 서버에 전송됨

- 네트워크 트래픽 추가 유발함
- 최소한의 정보만 사용함 (Session Id, 인증 토큰)
- 서버에 전송하지 않고 웹 브라우저 내부에 데이터를 저장하고 싶으면 웹 스토리지(localStorage, sessionStorage) 참고

Set-Cookie: 서버에서 클라이언트로 쿠키 전달

Cookie: 클라이언트가 서버에서 받은 쿠키를 저장하고 HTTP 요청 시 서버로 전달함

세션(Session): 일정 시간동안 같은 사용자로부터 들어오는 일련의 요구를 하나의 상태로 보고 그 상태를 일정하게 유지시키는 기술임. 즉 방문자가 웹 서버에 접속해 있는 상태를 하나의 단위로 보고 그것을 세션이라고 함

-> 클라이언트의 웹 브라우저에 쿠키를 저장해놓고 매 요청마다 헤더에 쿠키를 넣어 전달하는 방식을 사용하는 방식으로 인증을 구현할 경우, 쿠키가 유출되거나 조작될 수 있는 등 보안상 결함이 존재함. 개인 소유가 아닌 공용 컴퓨터에서 사용할 경우, 누구나 그 사용자의 비밀번호를 확인할 수 있게 되며 HTTP로 개인 정보를 주고 받는 것은 매우 위험함. Session이란 비밀번호를 비롯한 인증 정보를 쿠키가 아닌 서버 측에서 저장하고 관리하는 방식임

브라우저: 서버에서 전송한 데이터가 클라이언트에 도착해야 할 곳은 'Browser'임. Browser에는 데이터를 해석해주는 파서와 데이터를 화면에 표현해주는 렌더링엔진이 포함되어 있음

-> A computer program used to navigate the World Wide Web, chiefly by viewing web pages and following hyperlinks

***SQL**: Structured Query Language의 약자로, 데이터베이스에서 데이터를 추출하고 조작하는 데에 사용하는 데이터 처리 언어임

-> 쉽게 말해 데이터베이스에 저장된 정보를 쉽게 찾고 정리하는데 도움을 주는 도구임

***빌드: 프로젝트를 위해 작성한 Java 코드나 여러 자원들(.xml, .jar, .properties)를 JVM이나 WAS가 인식할 수 있는 구조로 패키징 하는 과정 및 결과물이다

또 단순히 컴파일 해주는 작업 뿐만 아니라, 테스팅, 검사, 배포까지 일련의 작업들을 통틀어 빌드라고 함

***Maven**: Gradle, Maven 모두 의존관계가 있는 라이브러리를 함께 다운로드 함

-> Apache사에서 만든 build tool임. pom.xml 파일을 통해 정형화된 빌드 시스템으로 프로젝트 관리를 해주고 프로젝트의 전체적인 라이프 사이클을 관리함
-> 프로젝트마다 하나의 pom.xml 파일이 있음
-> 프로젝트의 모든 설정, 의존성 등을 설정할 수 있음

***JDBC(Java Database Connectivity)**: 자바를 이용한 데이터베이스 접속과 SQL 문장의 실행, 그리고 실행 결과로 얻어진 데이터의 핸들링을 제공하는 방법과 절차에 관한 규약

***DOM(Document Object Model)**: 문서 객체 모델을 말하며, 웹 브라우저상의 화면을 이루고 있는 요소들을 지칭함

-> 브라우저에서는 HTML 코드를 DOM이라는 객체형태의 모델로 저장하는데 저장된 정보를 DOM Tree라고 함. 결국 HTML element는 Tree 형태로 저장됨
-> 복잡한 DOM Tree를 탐색하기 위해 Javascript로 탐색 알고리즘을 구현하면 너무 힘듦 그래서 브라우저에서는 DOM 개념을 통해서 다양한 DOM API(함수 묶음 정도)를 제공하고 있음. 브라우저는 DOM Tree 찾고 조작하는 걸 쉽게 도와주는 여러 가지 메서드를 제공하고 있음

-> getElementByid(): ID 정보를 통해서 찾을 수 있음
-> Element.querySelector(): DOM을 찾는데 유용한 querySelector 메서드임. css 스타일을 결정할 때 사용하던 Selector 문법을 통해 DOM에 접근할 수 있음

***Event**: 사용자가 마우스를 조작하면 그에 대한 반응을 하는 것, 즉 특정 이벤트가 발생 되었을 때 특정 함수를 실행할 수 있는게 해주는 것이 addEventListener임. addEventListener임의 두 번째 인자는 함수임. 나중에 이벤트가 발생할 때 실행되는 함수로 Event Handler, Event Listener라고 함

***Redirect, Forward**: 둘 다 작업 중인 페이지가 전환 됨. Redirect는 페이지 전환 주체가 클라이언트이며 Forward는 페이지 전환 주체가 서버임. 클라이언트가 주체가 되어 페이지를 전환하는 방법은 접속한 URL이 아닌 다른 URL로 직접 접속하는 방법 밖에 없다. 반대로 서버가 전환 주체가 되면 URL 주소가 바뀌지 않고도 서버 내부의 동작을 통해 다른 응답을 클라이언트에 내려줄 수 있음

-> redirect는 실제 클라이언트(웹 브라우저)에 응답이 나갔다가, 클라이언트가 redirect 경로로 다시 요청함. 따라서 클라이언트가 인지할 수 있고 URL 경로도 실제로 변경 됨. 반면에 포워드는 서버 내부에서 일어나는 호출이기 때문에 클라이언트가 전혀 인지 못 함

-> dispatcher.forward(): 다른 서블릿이나 JSP로 이동할 수 있는 기능임. 서버 내부에서 다시 호출이 발생함

***Expression Language**: EL은 값을 표현을 위한 언어로 JSP 스크립트의 표현식을 대신하여 속성 값을 쉽게 출력하도록 고안된 Language임

***JSTL(Java Standard Tag Library)**: JSP 안에 자바코드와 HTML 코드가 섞여 있으면 가독성도 떨어지고 수정할 때 어려움. 이것을 수정하기 위해서 JSTL이 등장함. JSTL은 JSP 페이지에서 조건문 처리, 반복문 처리 등을 HTML 태그 형태로 쉽게 작성할 수 있도록 도와줌

***API(Application Programming Interface)**: 애플리케이션이라는 단어는 고유한 기능을 가진 모든 소프트웨어를 나타냄. 애플리케이션이 어떤 프로그램이 제공하는 기능을 사용할 수 있게 만든 매개체임. 서로 다른 소프트웨어 시스템과 통신하기 위해 따라야 하는 규칙을 정의함

***HTTP API(Hypertext Transfer Protocol Application Programming Interface)**: HTTP를 통신 방식으로 사용하는 API를 HTTP API라고 함

-> HTML이 아니라 데이터를 전달함

-> 주로 JSON 형태로 데이터 통신함
-> 다양한 시스템에서 호출함
-> 데이터만 주고 받기 때문에 UI 화면이 필요하면 클라이언트가 별도 처리함
-> UI 클라이언트 접점

- 앱 클라이언트(아이폰, 안드로이드, PC 앱) to 서버
- 앱 브라우저에서 자바스크립트를 통한 HTTP API 호출함 to 서버
- React, Vue.js 같은 웹 클라이언트

-> 서버 to 서버

- 주문 서버 -> 결제 서버
- 기업 간 데이터 통신
-

***RESTful API(Representational State Transfer Application Programming Interface):**
HTTP 요청을 보낼 때 어떤 URI에 어떤 메소드를 사용할 지 개발자들 사이에 널리 지켜지는 약속임

-> HTTP URI(Uniform Resource Identifier)를 통해 자원(Resource)를 명시하고 HTTP Method(POST, GET, PUT, DELETE)를 통해 해당 자원(URI)에 대한 CRUD Operation을 적용하는 것을 의미함
->자원은 URL로 식별해야 함
->HTTP와 URI 기반으로 자원에 접근할 수 있도록 제공하는 애플리케이션 개발 인터페이스임. 기본적으로 개발자는 HTTP 메소드와 URI만으로 인터넷에 자료를 CRUD 할 수 있음

*****서버 사이드 렌더링**(Server Side Rendering) vs** 클라이언트 사이드 렌더링**(Client Side Rendering)**:

SSR - 서버 사이드 렌더링

-> HTML 최종 결과를 서버에서 만들어서 웹 브라우저에 전달함
-> 주로 정적인 화면에 사용함
-> 관련 기술: JSP, 타임리프 (백엔드 개발자 관련)

CSR - 클라이언트 사이드 렌더링

-> HTML 결과를 자바스크립트를 사용해 웹 브라우저에서 동적으로 생성해서 사용함
-> 주로 동적인 화면에 사용함. 웹 환경을 마치 앱처럼 필요한 부분부분 변경할 수 있음
-> 예) 구글 지도, Gmail, 구글 캘린더
-> 관련 기술: React, Vue.js (웹 프론트엔드 개발자)

참고

-> React, Vue.js를 CSR + SSR 동시에 지원하는 웹 프레임워크도 있음
-> SSR을 사용하더라도 자바스크립트를 사용해서 화면 일부를 동적으로 변경 가능함

*인터넷 프로토콜**(IP, Internet Protocol)**: 인터넷에서 네트워크의 두 호스트 간에 통신할 때 호스트 주소를 지정하고 전송 계층(transport layer)의 정보를 패킷으로 분할/조립하며 패킷을 IP 주소에 따라 목적지에 전송하는 통신 규약

->출발지 IP 주소(IP Address)와 도착지 IP 주소를 정하고 데이터 전달
-> 패킷(Packet)이라는 통신 단위로 데이터 전달 (Package + Bucket 합성어임. 출발지 IP, Port, 도착지 IP, Port, 전송 데이터 HTTP 메시지 등이 담겨짐)

-> 비연결성: 패킷을 받을 대상이 없거나 서비스 불능 상태여도 패킷을 전송함
-> 비신뢰성: 중간에 패킷이 사라지면?, 패킷이 순서대로 안 오면?
-> 프로그램 구분: 같은 IP를 사용하는 서버에서 통신하는 애플리케이션이 둘 이상이라면?

*인터넷 프로토콜 스위트**(Internet Protocol Suite)**: 인터넷에서 컴퓨터들이 서로 정보를 주고받는데 쓰이는 프로토콜의 집합이며 이를 TCP/IP 4계층 모델로 설명하거나 OSI 7계층 모델로 설명하기도 함. TCP/IP 4계층은 애플리케이션 계층, 전송 계층, 인터넷 계층, 링크(네트워크 인터페이스) 계층

*전송 계층: 송신자와 수신자를 연결하는 통신 서비스를 제공하며 연결 지향 데이터 스트림 지원, 신뢰성, 흐름 제어를 제공할 수 있으며 애플리케이션과 인터넷 계층 사이의 데이터가 전달될 때 중계 역할을 함. 대표적으로 TCP와 UDP가 있음

전송 제어 프로토콜(Transmission Control Protocol): 패킷 사이의 순서를 보장하고 연결 지향 프로토콜을 사용해서 연결을 하여 신뢰성을 구축해서 수신 여부를 확인하며 '가상회선 패킷 교환 방식'을 사용함

가상회선 패킷 교환 방식: 각 패킷에서는 가상회선 식별자가 포함되며 모든 패킷을 전송하며 가상회선이 해제되고 패킷을은 전송된 '순서대로' 도착하는 방식임

-> 연결 지향 - TCP 3 way handshake (가상 연결)
-> 데이터 전달 보증함
-> 순서 보장함
-> 신뢰할 수 있는 프로토콜임
-> 현재는 대부분 TCP 사용함

사용자 데이터그램 프로토콜(User Datagram Protocol): 순서를 보장하지 않고 수신 여부를 확인하지 않으며 단순히 데이터만 주는 데이터그램 패킷 교환 방식'을 사용함

데이터그램 패킷 교환 방식: 패킷이 독립적으로 이동하며 최적의 경로를 선택하여 가는데 하나의 메시지에서 분할된 여러 패킷은 서로 다른 경로로 전송될 수 있으며 도착한 '순서가 다를 수'있는 방식을 뜻함
-> 하얀 도화지에 비유함 (기능이 거의 없음)
-> 연결지향이 아님 (TCP 3 way handshake 사용하지 않음)
-> 데이터 전달 보증하지 않음
-> 순서 보장하지 않음

-> 단순하고 빠름
-> IP와 거의 같음 + PORT + 체크섬 정도만 추가함
-> 애플리케이션에서 추가 작업 필요함

*애플리케이션 계층: FTP, HTTP, SSH, SMTP, DNS 등 응용 프로그램이 사용되는 프로토콜 계층이며 웹 서비스, 이메일 등 서비스를 실질적으로 사람들에게 제공하는 계층

*전송 계층: 송신자와 수신자를 연결하는 통신 서비스를 제공하며 연결 지향 데이터 스트림 지원, 신뢰성, 흐름 제어를 제공할 수 있으며 애플리케이션과 인터넷 계층 사이의 데이터가 전달될 때 중계 역할을 함

***Port**: 같은 IP 내에서 프로세스 구분함

-> 아파트 몇 동이 IP 주소라면 Port는 호임

***URI (Uniform Resource Identifier)**:

Uniform: 리소스 식별하는 통일된 방식임
Resource: 자원, URI로 식별할 수 있는 모든 것
Identifier: 다른 항목과 구분하는데 필요한 정보임

-> URI 안에 URL(Uniform Resource Locator), URN(Uniform Resource Name) 포함 됨

URL(Uniform Resource Locator): 리소스가 있는 위치를 지정함
URN(Uniform Resource Name): 리소스에 이름을 부여함

Protocol: 어떤 방식으로 자원에 접근할 것인가 하는 약속 규칙
-> http는 80포토, https는 443 포트를 주로 사용함. 포트는 생략 가능함
-> https는 http에 보안 추가함 (HTTP Secure)

Host(Domain): IP 주소에 이름을 부여함

Port: 한 개의 컴퓨터엔 여러 개의 서버가 존재할 수 있음

Path: 자원(파일)의 경로를 의미함, 운영체제 상의 폴더와 비슷한 개념임. 따라서 여러 개의 폴더가 생겨나듯 경로 뒤에 여러 경로가 추가 될 수 있음

Parameter(Query String): Key, Value 형태로 이루어짐. 데이터를 전달하는 데 사용함

Fragment: html 내부 북마크에 사용함 (스크롤 내릴 필요가 없음)

***Resource**: HTTP 요청 대상을 리소스라고 함. 예를 들어 'A 식당의 메뉴를 보여줘' 라는 요청이 존재하면 'www.Arestaurant.com/menu' 처럼 '~~/menu' 이렇게 작성할 수 있음. menu는

정적인 파일을 의미하는 것이 아니라 메뉴들이라는 자원을 의미함. 이렇게 식별 가능한 자원은 리소스라고 함

-> 핵심 키워드(단어)
-> 예를 들어 “회원목록을 조회해라”, “회원목록을 등록해라(/read-member-list)”, “회원목록을 수정해라(/update-member)”, “회원목록을 삭제해라(/delete-member)” 에서 회원목록 (members)이 리소스임. 동사부분은 리소스 아님

URI는 리소스만 식별함. 리소스와 해당 리소스를 대상으로 하는 행위를 분리함

리소스: 회원
행위: 조회, 등록, 삭제, 변경
행위(메서드)는 HTTP Method로 구분함

***DNS(Domain Name System)**: IP 주소에 이름을 부여함

-> 사람이 읽을 수 있는 도메인 이름을 머신이 읽을 수 있는 IP 주소로 변환함
-> IP는 수시로 바뀜. 도메인 명을 IP 주소로 변환함

*네트워크: 컴퓨터 등의 장치들이 통신 기술을 이용하여 구축하는 연결망을 지칭함

-> 노드와 링크가 서로 연결되어 있으며 리소스를 공유하는 집합을 의미함
-> 좋은 네트워크란 많은 처리량을 처리할 수 있으며 지연 시간이 짧고 장애 빈도가 적으며 좋은 보안을 갖춘 네트워크를 말함

*웹 브라우저 요청흐름**:**

1. https://www.google.com/search?q=hello&hi=ko를 브라우저에 검색한다고 가정함
2. DNS 서버에 호스트 (google.com) 조회해서 IP 및 Port 얻어옴
3. 웹 브라우저가 HTTP 요청 메시지 생성함
4. Socket 라이브러리 통해서 애플리케이션 계층에서 OS계층으로 전달함
5. TCP/IP 패킷 생성함 (전송 데이터에 아래와 같은 HTTP 메시지를 추가함)
6. 패킷 정보가 인터넷으로 흘러 들어감
7. 구글 서버에 요청 패킷 도착함
8. 구글 서버에서 TCP/IP 패킷 다 까서 버리고 HTTP 메시지 분석함
9. 구글 서버에서 응답 메시지를 만듦 (200 OK, Content-Type, Content-Length 등)
10. 클라이언트에 응답 패킷 도착함
11. 응답 메시지를 열어서 웹 브라우저에 HTML 렌더링함

*상태 유지**(Stateful) vs** 무상태**(Stateless)**:

상태 유지(Stateful): 상태 유지라 함은 클라이언트와 서버 관계에서 서버가 클라이언트의 상태를 보존함을 의미함

-> 클라이언트의 정보를 기억한다는 말은 어딘가에 정보를 저장하고 통신할 때마다 읽는다는 뜻임. 이러한 정보들은 쿠키에 저장되거나, 서버의 세션 메모리에 저장되어 상태를 유지함
-> 해당 서버가 멈추거나 여러 이유로 해당 서버가 못 쓰게 되어 다른 서버를 사용해야 할 때 문제가 발생함

무상태(Stateless): 서버가 클라이언트의 상태를 보존하지 않음을 의미함. 즉 통신에 필요한 모든 상태 정보들은 클라이언트에서 가지고 있다가 서버와 통신할 때 데이터를 실어 보내는 것이 무상태임

-> 스케일 아웃, 수평적 확대에 유리함 (대략의 트래픽 발생 시에도 서버 확장을 통해 대처를 수월하게 할 수 있음)

*비 연결성(Connection Less): TCP/IP의 경우 기본적으로 연결을 유지함. 근데 클라이언트가 요청을 보내지 않아도 연결을 유지함. 이러한 경우 서버의 자원을 소모함. HTTP는 기본적으로 연결을 유지하지 않는 모델임, 요청을 주고 받을 때만 연결을 유지하고 응답이 끝나면 TCP/IP 연결을 끊음. 최소한의 자원으로 서버를 유지함

-> TCP/IP 연결을 새로 맺어야 함 (3 way handshake 시간 추가 됨)
-> 웹 브라우저로 사이트를 요청하면 HTML 뿐만 아니라 자바스크립트, CSS 추가 이미지 등 수 많은 자원이 함께 다운로드 됨
-> 지금은 HTTP 지속 연결(Persistent Connections)로 해결함

*노드: 서버, 라우터, 스위치 등 네트워크 장치를 의미함

*링크: 유선 또는 무선을 의미함

*처리량**(throughput)**: 링크 내에서 성공적으로 전달된 데이터의 양을 말하며 보통 얼만큼의 트래픽을 처리했는지 나타남

-> 단위로는 bps(bits per second)를 씀. 초당 전송 또는 수신되는 비트 수의 의미임. 처리량은 사용자들이 많이 접속할 때마다 커지는 트래픽, 네트워크 장치 간의 대역폭, 네트워크 중간에 발생하는 에러, 장치의 하드웨어 스펙에 영향을 받음

*트래픽: 특정 시점에 링크 내에 '흐르는' 데이터의 양을 말함

*대역폭: 주어진 시간 동안 네트워크 연결을 통해 흐를 수 있는 최대 비트 수

*지연 시간: 요청이 처리되는 시간을 말함. 어떤 메세지가 두 장치 사이를 왕복하는 데 걸린 시간을 말함

-> 지연시간은 매체 타입(무선, 유선), 패킷 크기, 라우터의 패킷 처리 시간에 영향을 받음

*네트워크 토폴로지**(network topology)**: 네트워크 토폴로지는 노드와 링크가 어떻게 배치되어 있느지에 대한 방식이지자 연결 형태를 의미함

*****병목 현상: 병목 현상은 전체 시스템의 성능이나 용량이 하나의 구성 요소로 인해 제한을 받는 영상임

***LAN(Local Area Network)**: 근거리 통신망을 의미함. 같은 건물이나 캠퍼스 같은 좁은 공간에서 운영됨

***MAN(Metropolitan Area Network)**: 대도시 지역 네트워크를 나타내며 도시 같은 넓은 지역에서 운영 됨

***WAN(Wide Area Network)**: 광역 네트워크를 의미하며 국가 또는 대륙 같은 더 넓은 지역에서 운영 됨

*****네트워크 성능 분석 명령어

ping(Packet INternet Groper): 네트워크 상태를 확인하려는 대상 노드를 향해 일정 크기의 패킷을 전송하는 명령어임. 이를 통해 해당 노드의 패킷 수신 상태와 도달하기까지 시간 등을 알 수 있으며 해당 노드까지 네트워크가 잘 연결되어 있는지 확인할 수 있음. ping은 TCP/IP 프로토콜 중에 ICMP 프로토콜을 통해 동작하며 이 때문에 ICMP 프로토콜을 지원하지 않는 기기를 대상으로 실행할 수 없거나 네트워크 정책상 ICMP나 traceroute를 차단하는 대상의 경우 ping 테스팅은 불가능함

netstat: 접속되어 있는 서비스들의 네트워크 상태를 표시하는 데 바로 사용 되며 네트워크 접속, 라우팅 테이블, 네트워크 프로토콜 등 리스트를 보여줌. 주로 서비스의 포트가 열려 있는지 확인 함

nslookup: DNS에 관련된 내용을 확인하기 위해 쓰는 명령어임. 특정도메인에 매핑된 IP를 확인하기 위해 사용함

tracert: 목적지 노드까지 네트워크 경로를 확인하는데 사용하는 명령어임. 목적지 노드까지 구간들 중 어느 구간에서 응답 시간이 느려지는지 확인 할 수 있음

***Dispatcher Servlet**: HTTP 요청을 가장 먼저 받아 적합한 컨트롤러로 보내주는 프론트 컨트롤러라고 할 수 있음

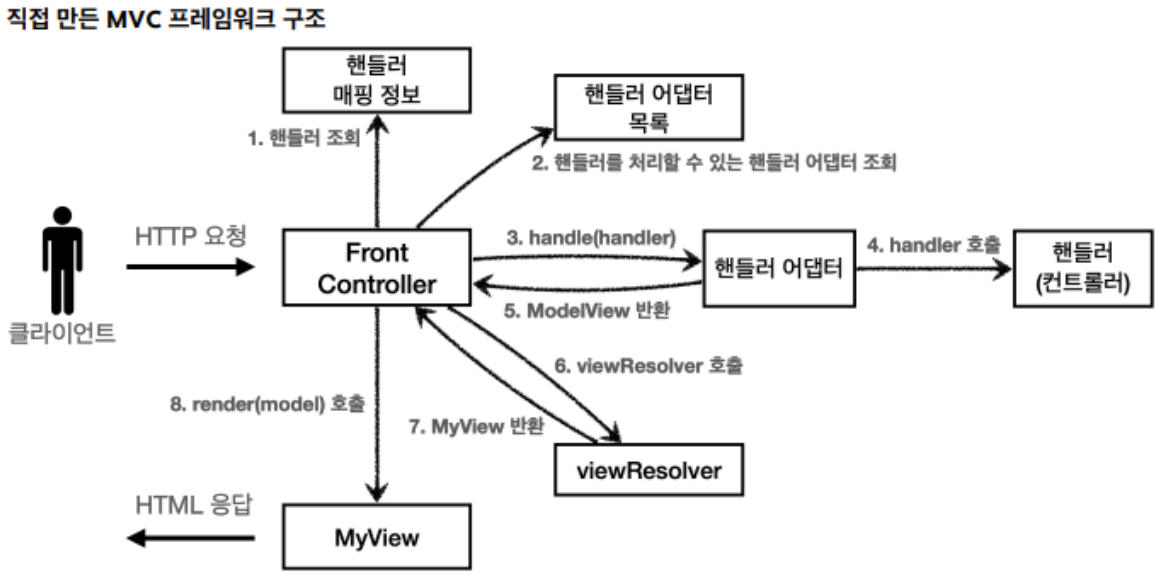
-> 프론트 컨트롤러 서블릿 하나로 클라이언트의 요청을 받음
-> 프론트 컨트롤러가 요청에 맞는 컨트롤러를 찾아서 호출함
-> 입구를 하나로, 공통 처리 가능
-> 프론트 컨트롤러를 제외한 나머지 컨트롤러는 서블릿을 사용하지 않아도 됨

-> 클라이언트로부터 어떠한 요청이들어오면 Tomcat 같은 서블릿 컨테이너가 요청을 받게 됨. 그리고 이 모든 요청을 프론트 컨트롤러인 Dispatcher-Servlet이 가장 먼저 받게 됨. 그러면 디스패처 서블릿은 공통적인 작업을 먼저 처리한 후에 해당 요청을 처리하는 컨트롤러를

찾아서 작업을 위임함

-> 클라이언트의 모든 요청을 받은 후 이를 처리할 핸드러에게 남기고 핸들러가 처리한 결과를 받아 사용자에게 응답 결과를 보여 줌

과정



-> 클라이언트의 요청을 디스패처 서블릿이 받음

-> 컨트롤러 찾음 (디스패처 서블릿은 요청을 처리할 핸들러인 컨트롤러를 찾고 해당 객체의 메소드를 호출함. 따라서 가장 먼저 어느 컨트롤러가 요청을 처리 할 수 있는지 식별해야 하는데 해당 역할을 하는 것이 바로 HandlerMapping임)

-> 클라이언트의 요청을 찾은 컨트롤러로 위임할 핸들러 어댑터를 찾아서 전달함 (이후에 컨트롤러로 요청을 위임해야 하는데 디스패처 서블릿은 컨트롤러로 요청을 직접 위임하는 게 아니라 HandlerAdapter를 통해 위임함

-> 핸들러 어댑터가 컨트롤러로 요청을 위임함

-> 비즈니스 로직을 처리함 (컨트롤러는 서비스를 호출하고 우리가 작성한 비즈니스 로직들이 진행 됨)

-> 비즈니스 로직을 처리한 후 컨트롤러가 리턴값을 반환함 (응답 데이터를 사용하는 경우에는 ResponseEntity를 반환하게 되고 응답 페이지를 보여주는 경우라면 String으로 View의 이름으로 반환할 수 있음)

-> HandlerAdapter가 반환값을 처리함 (HandlerAdapter는 컨트롤러부터 받은 응답을 응답 처리기인 ReturnValueHandler가 후 처리한 후에 디스패처 서블릿으로 돌려줌)

->서버의 응답을 클라이언트로 반환함 (응답이 데이터라면 그대로 반환되지만, 응답이 화면이라면 View의 이름에 맞는 View를 찾아서 반환해주는 ViewResolver가 적절한 화면을 내려줌)

*싱글톤 패턴**(Singleton Pattern)**: 하나의 클래스에 오직 하나의 인스턴스만 가지는 패턴임

-> 보통 데이터베이스 연결 모듈에 많이 사용함

-> 하나의 인스턴스를 만들어 놓고 해당 인스턴스를 다른 모듈들이 공유하며 사용하기 때문에 인스턴스를 생성할 때 드는 비용이 줄어드는 장점이 있음. 하지만 의존성이 높아지는 단점이 있음

***프록시 패턴**(Proxy Pattern)**: 대상 객체에 접근하기 전, 그 접근에 대한 흐름을 가로채 해당 접근을 필터링하거나 수정하는 등의 역할을 하는 계층이 있는 디자인 패턴임

***프록시 서버**(Proxy Server)**: 프록시는 “대리”의 의미임. 클라이언트가 원서버가 아닌 자신을 통해 다른 네트워크 서비스에 간접적으로 접속할 수 있게 해 주는 컴퓨터 시스템이나 응용 프로그램을 가리킴

***public:** 클래스에 정의된 함수에서 접근 가능하며 자식 클래스와 외부 클래스에서 접근 가능한 범위

***protected**: 클래스에 정의된 함수에서 접근 가능, 자식 클래스에서 접근 가능하지만 외부 클래스에서 접근 불가능한 범위

-> 같은 패키지거나 상속관계에서 사용 가능함

***private**: 클래스에 정의된 함수에서 접근 가능하지만 자식 클래스와 외부 클래스에서 접근 불가능한 범위

-> 같은 클래스 내에서만 사용 가능함

***Enum**: 정해져 있고 한정된 데이터 묶음을 열거형 타입인 Enum으로 묶어주면 보다 구조적인 프로그램이 가능함

데이터 주제에 따라 정해져 있고 한정된 값만 가지는 경우가 있음. 예를 들어 ‘요일’은 월,화,수,목,금,토,일 이렇게 구성되어 있고 계절은 봄, 여름, 가을, 겨울로 구성되어 있다

***캐시**(Cache)**: 서버 지연을 줄이기 위해 웹 페이지, 이미지, 기타 유형의 웹 멀티미디어 등의 웹 문서 들을 임시 저장하기 위한 정보기술임

-> 캐시 유효 시간이 초과해도 서버의 데이터가 갱신되지 않으면 304 Not Modified + 헤더 메타 정보만 응답함(바디 X). 클라이언트는 서버가 보낸 응답 헤더 정보로 캐시의 메타 정보를 갱신함 그리고 캐시에 저장되어 있는 데이터 재활용함. 결과적으로 네트워크가 다운로드가 발생하지만 용량이 적은 헤더 정보만 다운로드함

검증 헤더: 캐시 데이터와 서버 데이터가 같은지 검증하는 데이터

조건부 요청 헤더
-> 검증 헤더로 조건에 따른 분기 If-Modified-Since: Last-Modified 사용함 그리고 if-None-Match: ETag 사용함. 조건이 만족하면 200 OK, 만족하지 않으면 304 Not Modified

*자바 뷰 템플릿**(Java View Template)**: HTML을 편리하게 생성하는 뷰 기능임

-> 서블릿과 자바코드 만으로 HTML을 만들면 동적인 HTML 만들기가 힘듦. 그래서 자바 코드에 HTML을 넣는 게 아니라 HTML에 자바코드를 넣어 HTML이 쉬워짐. 이것이 바로 템플릿 엔진이 나온 이유임. 템플릿 엔진이 나오면 HTML 문서에서 필요한 곳만 코드를 적용해서 동적으로 변경할 수 있음

1) JSP(Java Server Pages): 속도 느림, 기능 부족

-> <%@ page contentType=”text/html;charset=UTF-8” language=’java’ %> 첫 줄은 JSP 문서라는 뜻임
-> HTML과 JSP의 차이점은 자바 소스코드의 유무임
-> HTML 문서 내부에 자바 소스 코드 작성을 가능하게 하는 Server Side Script임
-> HTML 내부에 자바소스코드를 사용하면서 동적페이지에 한 걸음 다가감. 즉 Javascript를 사용하지 않고도 페이지를 동적으로 만들어주는 역할을 함
-> JSP 파일이 실행되면 WAS는 내부적으로 JSP 파일을 서블릿 파일로 변환하고 동작하여 필요한 기능을 수행함 그리고 생성된 웹 페이지와 함께 클라이언트로 응답함
-> JSP는 해당 프로젝트에서 MVC의 V를 담당함
-> JSP와 서블릿 하는 일은 동일하지만 JSP는 HTML 내부에 Java 소스코드가 들어가는 반면 서블릿은 Java 코드 내에 HTML 코드가 있음. 이에 따라 서블릿은 복잡한 로직 구현에 적합하고 JSP는 수정이 용이하기 때문에 화면 작성 작업할 때 좋음. 각 각의 최대한의 장점을 살려 MVC에서 View는 html이나 jsp와 같이 클라이언트에게 보여주는 문서를 뜻함. Controller는 Servlet과 같이 클라이언트의 요청과 응답을 이어주는 API임. Model은 요청에 대한 로직을 수행하는 부분임

(JSP는 성능과 기능면에서 다른 템플릿 엔진과의 경쟁에서 밀림)

2) 프리마커(Freemarker), Velocity(벨로시티): 속도 문제 해결, 다양한 기능 제공함

3) 타임리프(Thymeleaf): 컨트롤러가 전달하는 데이터를 이용해 동적으로 화면을 만들어주는 역할을 하는 뷰 템플릿 엔진임

-> 네츄럴 템플릿임 (타임리프 문법을 HTML의 모양을 유지하면서 뷰 템플릿 적용 가능), 스프링 MVC와 강력한 기능 통합, 최선의 선택임(단 성능은 프리마커, 벨리시티가 더 빠름)

-> 화면을 동적으로 만들려면 템플릿 엔진을 사용해야 함. 미리 정의된 템플릿(Template)을 만들고 동적으로 HTML 페이지를 만들어서 클라이언트에 전달하는 방식임. 요청이 올 때마다 서버에서 새로운 HTML 페이지를 만들어 주기 때문에 서버 사이드 렌더링 방식임

-> A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes

***Business Logic(Domain Logic)**: 비즈니스 로직은 유저의 눈에는 보이지 않지만, 유저가 결과를 올바르게 도출 할 수 있도록 짜여진 코드 로직임

-> 사용자의 요구사항을 해결하기 위한 실질적인 코드임

-> Business Logic is the part of the program that encodes the real-world business rules that determine how data can be created, stored, and changed

***Entity vs DTO(Data Transfer Object)**:
Entity: DB 테이블에 존재하는 Column들을 필드로 가지는 객체임

-> Entity는 DB의 테이블과 1대 1로 대응되며 때문에 테이블이 가지지 않는 컬럼을 필드로 가져서는 안 됨
-> 다른 클래스를 상속 받거나 인터페이스의 구현체여서는 안 됨

DTO(Data Transfer Object): 데이터를 전달하기 위해서 사용하는 객체임

Entity는 DB 테이블과 1대 1로 대응 되는 객체이고 DTO는 계층 간 교환을 위한 순수한 데이터 객체임

왜 분리해야 하는가?

-> Entity와 DTO를 분리해야 하는 가장 근본적인 이유는 관심사가 서로 다르기 때문. 관심사의 분리 (separation of concerns, SoC)는 소프트웨어 분야의 오래된 원칙 중 하나로써 서로 다른 관심사들을 분리하여 변경 가능성을 최소화하고 유연하며 확장 가능한 클리 아키텍처를 구축하도록 도와 줌

DTO의 핵심 관심사는 이름 그대로 데이터 전달임. DTO는 데이터를 담고 다른 계층 또는 다른 컴포넌트들로 데이터를 넘겨주기 위한 자료구조임. 그러므로 어떠한 기능 및 동작도 없어야 함

반면에 Entity는 핵심 비즈니스 로직을 담는 비즈니스 도메인의 영역의 일부임. 그러므로 Entity 또는 도메인 객체는 그에 따른 비즈니스 로직이 추가 될 수 있음. Entity 또는 도메인

객체는 다른 계층이나 컴포넌트들 사이에서 전달을 위해 사용되는 객체가 아님. Entity와 DTO는 엄연히 서로 다른 관심사를 가지고 있고 그렇게 분리하는 것이 합리적임

***Static Method(Class Method) vs Instance Method**:

-> Java에서 static을 사용한다는 것은 어떠한 값이 메모리에 한 번 할당되면 프로그램이 끝날 때까지 그 메모리에 값이 유지된다는 것을 의미함. 즉 특정한 값을 공유하는 경우라면 static 사용 시 메모리의 이점을 얻을 수 있음 그래서 static 메서드는 객체 생성없이 호출이 가능한 메서드 임
->인스턴스 변수, 인스턴스 메서드는 말 그대로 인스턴스를 생성한 후 사용이 가능함

***HTTPServletRequest:**

HTTP 요청 메시지를 개발자가 직접 파싱해서 사용해도 되지만 매우 불편함. 서블릿은 개발자가 HTTP 요청 메시지를 편리하게 사용할 수 있도록 개발자 대신에 HTTP 요청 메시지를 파싱함. 그리고 그 결과를 HttpServletRequest 객체에 담아서 제공함

임시 저장소 기능

해당 HTTP 요청이 시작부터 끝날 때까지 유지되는 임시 저장소 기능

저장: request.setAttribute(name, value)
조회: request.getAttribute(name)

HttpServletRequest, HttpServletResponse를 사용할 때 HTTP 요청 메시지, HTTP 응답 메시지를 편리하게 사용하도록 도와주는 객체임

HTTP 요청 메시지를 통해 클라이언트에서 서버로 데이터를 전달하는 방법

1) GET - 쿼리 파라미터
-> /url**?username=hello&age=20**
-> 메시지 바디 없이, URL의 쿼리 파라미터에 데이터를 포함해서 전달함
-> 예) 검색, 필터, 페이지 등에서 많이 사용하는 방식임
-> 쿼리 파라미터는 URL에 다음과 같이 ?를 시작으로 보낼 수 있음. 추가 파라미터는 &로 구분 됨
-> request.getParameter()로 조회 가능함

서버에서는 HttpServletRequest가 제공하는 다음 메서드를 통해 쿼리 파라미터를 편리하게 조회함
-> String username = request.getParameter(“username”); // 단일 파라미터 조회
-> Enumeration<String> parameterNames = request.getParameterNames(); // 파라미터 이름들 조회
-> Map<String, String[]> parameterMap = request.getParameterMap(); // 파라미터를 Map으로 조회

-> String[] usernames = request.getParameterValues(“username”); // 파라미터가 중복일 때 조회

2) POST - HTML Form
-> content-type: application/x-www-form-urlencoded
-> 메시지 바디에 쿼리 파라미터 형식으로 전달함. username=hello&age=20
-> 예) 회원 가입, 상품 주문, HTML Form 사용함
-> Post HTML Form을 전송하면 웹 브라우저는 다음 형식으로 HTTP 메시지를 만든다 (웹 브라우저 개발자 모드 확인)

- 요청 URL: http://localhost:8080/request-param
- content-type: application/x-www-form-urlencoded
- message body: username=hello&age=20

application/x-www-form-urlencoded 형식은 앞서 GET에서 살펴본 쿼리 파라미터 형식과 같다. 따라서 쿼리 파라미터 조회 메서드를 그대로 사용하면 됨. 클라이언트(웹 브라우저) 입장에서는 두 방식에 차이가 있지만 서버 입장에서는 둘의 형식이 동일하므로 request.getParameter()로 편리하게 구분없이 조회할 수 있음

정리하면 request.getParameter()는 GET URL 쿼리 파라미터 형식도 지원하고 POST HTML Form 형식도 둘 다 지원함

-> content-type은 HTTP 메시지 바디의 데이터 형식을 지정함
-> GET URL 쿼리 파라미터 형식으로 클라이언트에서 서버로 데이터를 전달할 때는 HTTP 메시지 바디를 사용하지 않기 때문에 content-type이 없음
-> POST HTML Form 형식으로 데이터를 전달하면 HTTP 메시지 바디에 해당 데이터를 포함해서 보내기 때문에 바디에 포함된 데이터가 어떤 형식인지 content-type을 꼭 지정해야 함. 이렇게 폼으로 데이터를 전송하는 형식을 application/x-www-form-urlencoded라 함

3) HTML Message Body에 데이터를 직접 담아서 요청함
-> HTTP API에서 주로 사용함. JSON, XML, TEXT
-> 데이터 형식은 주로 JSON 사용함
-> POST, PUT, PATCH

HTTP 요청 데이터 - API 메시지 바디 - 단순 텍스트

가장 단순한 텍스트 메시지를 HTTP 메세지 바디에 담아서 전송할 수 있는데 HTTP 메시지 바디의 데이터를 InputStream을 사용해 직접 읽을 수 있음

(inputStream은 byte 코드를 반환함. byte 코드를 우리가 읽을 수 있는 문자(String)으로 보려면 문자표(Charset)를 지정해줘야 함. 여기서는 UTF_8 Charset을 지정해줌)

문자 전송
-> POST http://localhost:8080/request-body-string

-> content-type: text/plain
-> message body: hello
-> 결과: messageBody = hello

HTTP 요청 데이터 - API 메시지 바디 - JSON

JSON 형식 전송
-> POST http://localhost:8080/request-body-json
-> content-type: application/json
-> message body: {“username” : “hello”, “age”:20}
-> 결과: messageBody = { “username”: “hello”, “age”: 20}

Lombok이 제공하는 @Getter @Setter 때문에 Getter, Setter 추가 안 해줘도 됨

Lombok: 반복적인 Getter/Setter, ToString과 같은 반복적인 자바 코드를 컴파일할 때 자동으로 생성해주는 라이브러리임. Lombok 라이브러리를 사용하면 반복적인 소스코드를 제거할 수 있으므로 코드를 깔끔하게 짤 수 있음

-> Lombok annotation library which helps to reduce boilerplate code

JSON 결과를 파싱해서 사용할 수 있는 자바 객체로 변환하려면 Jackson, Gson 같은 JSON 변환 라이브러리를 추가해서 사용해야 함. 스프링 부트로 Spring MVC를 선택하면 기본으로 Jackson 라이브러리 (ObjectMapper)를 함께 제공함

(HTML form 데이터도 메시지 바디를 통해 전송되므로 직접 읽을 수 있음. 하지만 편리한 파라미터 조회 기능 (request.getParameter(...))을 제공하기 때문에 조회 기능을 사용하면 됨

-> WAS는 웹 브라우저로부터 Servlet 요청을 받으면, 요청할 때 가지고 있는 정보를 HTTPServletRequest 객체를 생성하여 저장함. 웹 브라우저에게 응답을 보낼 때 사용하기 위해 HTTPServletResponse 객체를 생성함. 생성된 HTTPServletRequest, HTTPServletResponse 객체를 서블릿에게 전달함

HTTPServletRequest
-> HTTP 프로토콜의 request 정보를 서블릿에게 전달하기 위한 목적으로 사용함
-> 헤더정보, 파라미터, 쿠키, URI, URL 등의 정보를 읽어 들이는 메소드를 가지고 있음
-> Body의 Stream을 읽어 들이는 메소드를 가지고 있음
-> HTTPServletRequest을 사용하면 값을 받아올 수가 있는데 만약 회원 정보를 컨트롤러로 보냈을 때 HTTPServletRequest 객체 안에 모든 데이터가 들어감 그리고 데이터를 꺼낼 때는 getParameter()을 이용하면 됨 (반환 타입은 String임)

HTTPServletResponse
-> WAS는 어떤 클라이언트가 요청을 보냈는지 알고 있고 해당 클라이언트에게 응답을 보내기 위한 HTTPServletResponse 객체를 생성하여 서블릿에게 전달함
-> 서블릿은 해당 객체를 이용하여 content type, 응답코드, 응답 메세지 등을 전송함

JSON 결과를 파싱해서 사용할 수 있는 자바 객체로 변환하려면 Jackson, Gson 같은 JSON 변환 라이브러리를 추가해서 사용해야 함. 스프링 부트로 Spring MVC를 선택하면 기본으로 Jackson 라이브러리(ObjectMapper)를 함께 제공함

***HTTPServletResponse**:

HTTPServletResponse 역할

1) HTTP 응답 메시지 생성
-> HTTP 응답코드 지정함
-> 헤더 생성함
-> 바디 생성함

2) 편의 기능 제공
-> Content-Type, 쿠키, Redirect

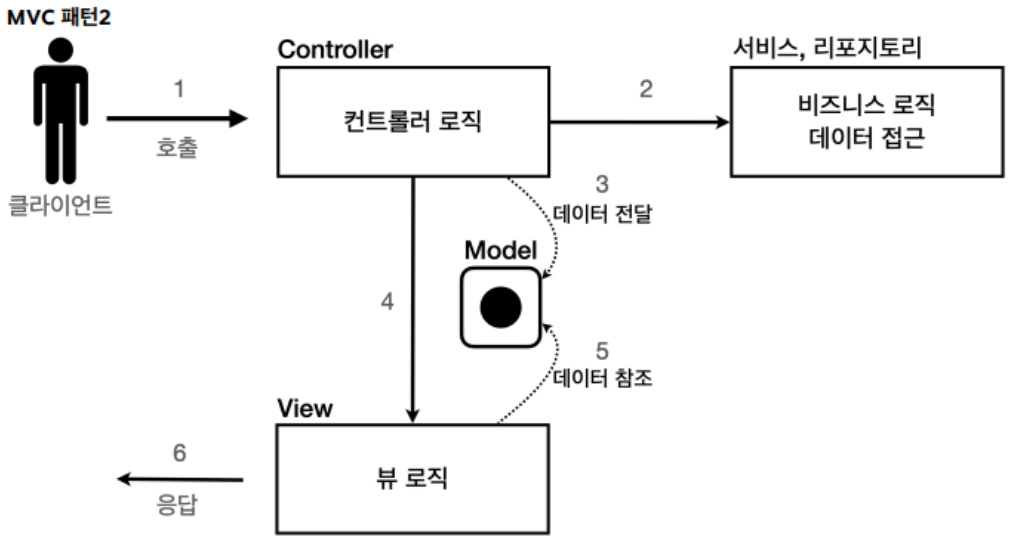
HTTP 응답 메시지는 주로 다음 내용을 담아서 전달함

1) 단순 텍스트 응답
-> 예를 들어 writer.println("ok");

2) HTML 응답
-> HTTP 응답으로 HTML을 반환할 때는 content-type을 text/html로 지정해야 함

3) HTTP API - MessageBody JSON 응답
-> HTTP 응답으로 JSON을 반환할 때는 content-type을 application/json로 지정해야 함. Jackson 라이브러리가 제공하는 objectMapper.writeValueAsString()를 사용하면 객체를 JSON 문자로 변경할 수 있음

***MVC (Model View Controller)**: MVC 패턴은 Servlet을 Controller와 JSP를 View 특화된 쪽으로 하여 역할을 나눔



Controller: HTTP 요청을 받아서 파라미터를 검증하고 비즈니스 로직을 실행함. 그리고 뷰에 전달할 결과 데이터를 조회해서 모델에 담는다

Model: 뷰에 출력할 데이터를 담아둠. 뷰가 필요한 데이터를 모두 모델에 담아서 전달해주는 덕분에 뷰는 비즈니스 로직이나 데이터 접근을 몰라도 되고 화면을 렌더링 하는 일에 집중할 수 있음

View: 모델에 담겨있는 데이터를 사용해서 화면을 그리는 일에 집중함 (화면을 렌더링 하는 일에 집중) 여기서는 HTML을 생성하는 부분을 말함

(컨트롤러에 비즈니스 로직을 둘 수 있지만 이렇게 되면 컨트롤러가 너무 많은 역할을 담당함. 그래서 일반적으로 비즈니스 로직은 서비스(Service)라는 계층을 별도로 만들어서 처리함 그리고 컨트롤러는 비즈니스 로직이 있는 서비스를 호출하는 역할을 담당함. 참고로 비즈니스 로직을 변경하면 비즈니스 로직을 호출하는 컨트롤러의 코드도 변경할 수 있음)

Model은 HTTPServletRequest 객체를 사용함. request 내부에 데이터 저장소를 가지고 있는데 request.setAttribute(), request.getAttribute()를 사용하면 데이터를 보관하고 조회할 수 있음

HttpServletRequest를 Model로 사용함. request가 제공하는 setAttribute()를 사용하면 request 객체에 데이터를 보관해서 뷰에 전달할 수 있음. 뷰는 request.getAttribute()를 사용해서 데이터를 꺼내면 됨

/WEB-INF : 이 경로 안에 JSP가 있으면 외부에서 직접 JSP를 호출할 수 없음. 우리가 기대하는 것은 항상 컨트롤러를 통해서 JSP를 호출함

***DispatcherServlet**:

-> 스프링 MVC도 프론트 컨트롤러 패턴으로 구현되어 있음
-> 스프링 MVC의 프론트 컨트롤러가 바로 디스패처 서블릿(DispatcherServlet)임
-> 그리고 이 디스패처 서블릿이 바로 스프링 MVC의 핵심임

DispatcherServlet 서블릿 등록

-> DispatcherServlet도 부모 클래스에서 HttpServlet을 상속 받아서 사용하고 서블릿으로 동작함

-> 스프링 부트는 DispatcherServlet을 서블릿으로 자동으로 등록하면서 모든 경로(urlPatterns="/")에 대해서 매핑함 (더 자세한 경로가 우선순위가 높다. 그래서 기존에 등록한 서블릿도 함께 동작함)

요청 흐름

-> 서블릿이 호출되면 HttpServlet이 제공하는 service()가 호출 됨

-> 스프링 MVC는 DispatcherServlet의 부모인 FrameworkServlet에서 service()를 오버라이드 해주었음

-> FrameworkServlet.service()를 시작으로 여러 메서드가 호출되면서 DispatacherServlet . doDispatch()가 호출 됨

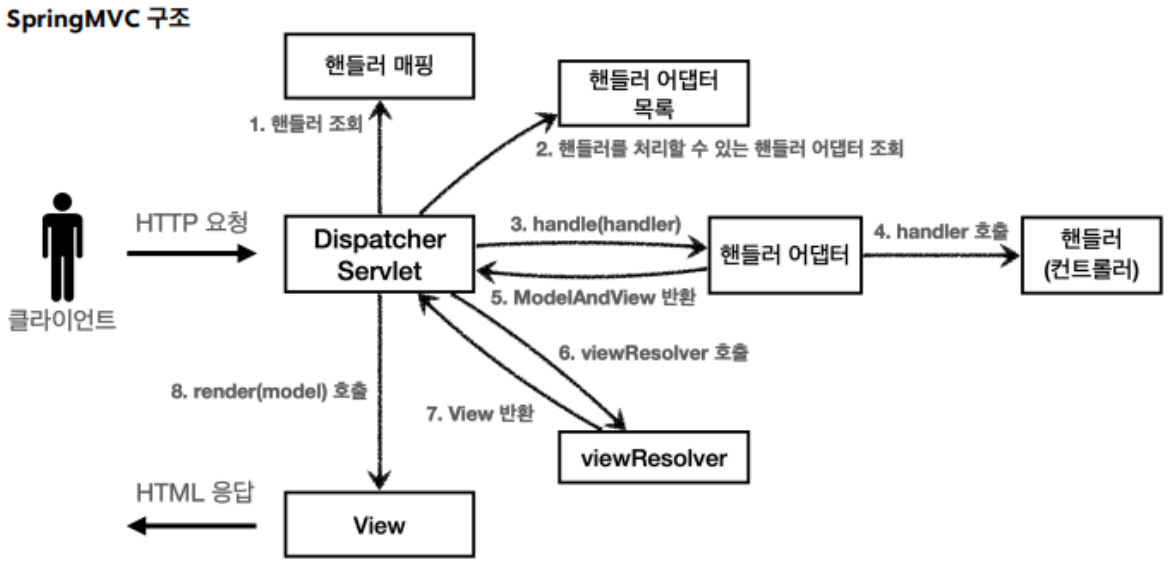
FrontController 패턴 특징

-> 프론트 컨트롤러 서블릿 하나로 클라이언트의 요청을 받음

-> 프론트 컨트롤러가 요청에 맞는 컨트롤러를 찾아서 호출함
-> 입구를 하나로, 공통 처리 가능함

-> 프론트 컨트롤러를 제외한 나머지 컨트롤러는 서블릿을 사용하지 않아도 됨

Spring MVC 구조**:**



1) 핸들러 조회: 핸들러 매핑을 통해 요청 URL에 매핑된 핸들러(컨트롤러)를 조회함
- 핸들러 매핑에서 이 컨트롤러를 찾을 수 있어야 함
- 스프링 빈의 이름으로 핸들러를 찾을 수 있는 핸들러 매핑이 필요함

2) 핸들러 어댑터 조회: 핸들러를 실행할 수 있는 핸들러 어댑터를 조회함
- 핸들러 매핑을 통해서 찾은 핸들러를 실행할 수 있는 핸들러 어댑터가 필요함

3) 핸들러 어댑터 실행: 핸들러 어댑터를 실행함

4) 핸들러 실행: 핸들러 어댑터가 실제 핸들러를 실행함

5) ModelAndView 반환: 핸들러 어댑터는 핸들러가 반환하는 정보를 ModelAndView로 변환해서 반환함

6) ViewResolver 호출: 뷰 리졸버를 찾고 실행함 (JSP인 경우 InternalResourceViewResolver가 자동 등록되고 사용됨)

7) View 반환: 뷰 리졸버는 뷰의 논리 이름을 물리 이름으로 바꾸고 렌더링 역할을 담당하는 뷰 객체를 반환함 (JSP인 경우 InternalResourceView(JstlView)를 반환하는데 내부에 forward() 로직이 있음)

8) 뷰 렌더링: 뷰를 통해서 뷰를 렌더링함

인터페이스 살펴보기

-> 스프링 MVC의 가장 큰 강점은 DispatcherServlet 코드의 변경 없이 원하는 기능을 변경하거나 확장할 수 있음. 지금까지 설명한 대부분을 확장 가능할 수 있게 인터페이스로 제공함

-> 이 인터페이스들만 구현해서 DispatcherServlet에 등록하면 나만의 컨트롤러 만들 수 있음

주요 인터페이스 목록

핸들러 매핑: org.springframework.web.servlet.HandlerMapping
핸들러 어댑터: org.springframework.web.servlet.HandlerAdapter
뷰 리졸버: org. springframework.web.servlet.ViewResolver
뷰: org.springframework.web.servlet.View

뷰 리졸버**(View Resolver):**

스프링 부트는 InternalResourceViewResolver라는 뷰 리졸버를 자동으로 등록하는데, 이때 application.properties에 등록한 spring.mvc.view.prefix, spring.mvc.view.suffix 설정 정보를 사용해서 등록함

뷰 리졸버 동작 방식

스프링 부트가 자동 등록하는 뷰 리졸버
-> BeanNameViewResolver: 빈 이름으로 뷰를 찾아서 반환함 (예: 엑셀 파일 생성 기능에 사용)
-> InternalResourceViewResolver: JSP를 처리할 수 있는 뷰를 반환함

1) 핸들러 어댑터 호출
-> 핸들러 어댑터를 통해 new-form이라는 논리 뷰 이름을 획득함

2) ViewResolver 호출
-> new-form이라는 뷰 이름으로 viewResolver를 순서대로 호출함
-> BeanNameViewResolver는 new-form이라는 이름의 스프링 빈으로 등록한 뷰를 찾아야 하는데 없음
-> InternalResourceViewResolver가 호출 됨

3) InternalResourceViewResolver
-> 이 뷰 리졸버는 InternalResourceView를 반환함

4) 뷰 - InternalResourceView
-> InternalResourceView는 JSP처럼 포워드 forward()를 호출해서 처리할 수 있는 경우에 사용함

5) view.render()
-> view.render()가 호출되고 InternalResourceView는 forward()를 사용해서 JSP를 실행함

(InternalResourceViewResolver는 만약 JSTL 라이브러리가 있으면 InternalResourceView를 상속받은 JstlView를 반환함. JstlView는 JSTL 태그 사용시 약간의 부가 기능이 추가 됨)

(다른 뷰는 실제 뷰를 렌더링하지만 JSP의 경우 forward()를 통해서 해당 JSP로 이동(실행)해야 렌더링이 됨. JSP를 제외한 나머지 뷰 템플릿들은 forward() 과정없이 바로 렌더링 됨)

(Thymeleaf 뷰 템플릿을 사용하면 ThymeleafViewResolver를 등록해야 함. 최근에는 라이브러리만 추가하면 스프링 부트가 이런 작업도 모두 자동화해줌)

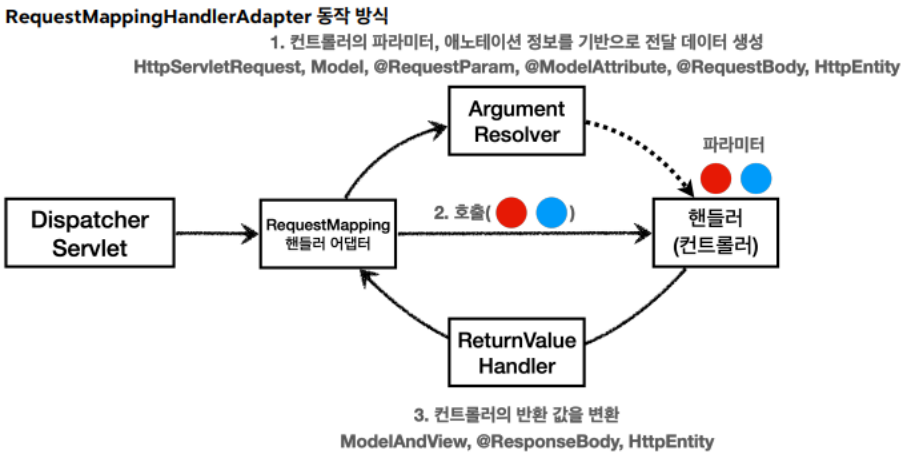
***Spring MVC -** 실용적인 방식**:**

****@RequestMapping**: 요청 정보를 매핑한다. 해당 URL이 호출되면 이 메서드가 호출 됨. 애노테이션을 기반으로 동작하기 때문에 메서드의 이름을 임의로 지으면 됨

두 가지 기능을 포함하고 있음. 가장 우선순위가 높은 핸들러 매핑과 핸들러 어댑터는 아래 두 가지임

-> RequestMappingHandlerMapping

-> RequestMappingHandlerAdapter



- ArgumentResolver: 애노테이션 기반의 컨트롤러는 매우 다양한 파라미터를 사용할 수 있음. HttpServletRequest, Model은 물론이고, @RequestParam, @ModelAttribute 같은 애노테이션 그리고 @RequestBody, HttpEntity 같은 HTTP 메시지를 처리하는 부분까지 매우 큰 유연함을 보여줄 수 있음
  이렇게 파라미터를 유연하게 처리할 수 있는 이유가 ArgumentResolver 덕분임
  애노테이션 기반 컨트롤러를 처리하는 RequestMappingHandAdapter는 바로 이 ArgumentResolver를 호출해서 컨트롤러(핸들러)가 필요로 하는 다양한 파라미터의 값(객체)을 생성함. 그리고 파라미터의 값이 모두 준비되면 컨트롤러를 호출하면서 값을 넘김

- 동작 방식: ArgumentResolver의 supportsParameter()를 호출해서 해당 파라미터를 지원하는지 체크하고 지원하면 resolveArgument()를 호출해서 실제 객체를 생성함. 그리고 이렇게 생성된 객체가 컨트롤러 호출시 넘어감

  원하면 직접 이 인터페이스를 확장해서 원하는 ArgumentResolver를 만들 수 있음

- ReturnValueHandler: HandlerMethodReturnValueHandler를 줄여서 ReturnValueHandler라 부름. ArgumentResolver와 비슷한데 이것은 응답 값을 변환하고 처리함. 컨트롤러에서 String 뷰 이름을 반환해도 동작하는 이유가 ReturnValueHandler 덕분임

-> @RequestMapping은 Class와 Method에 붙일 수 있음
-> @GetMapping, @PostMapping은 Method에서만 붙일 수 있음
-> @RequestMapping에 method 속성으로 HTTP 메서드를 지정하지 않으면 HTTP 메서드와 무관하게 호출 됨. GET, HEAD, POST, PUT, PATCH, DELETE 모두 허용함

**@Controller:

-> 스프링이 자동으로 스프링 빈으로 등록함 (내부에 @Component 애노테이션이 있어서 컴포넌트 스캔의 대상이 됨)
-> 스프링 MVC에서 애노테이션 기반 컨트롤러로 인식함

ModelAndView: 모델과 뷰 정보를 담아서 반환하면 됨

RequestMappingHandlerMapping은 스프링 빈 중에서 @RequestMapping 또는 @Controller가 클래스 레벨에 붙어 있는 경우에 매핑 정보로 인식함

Model 파라미터
-> save(), member()를 보면 Model을 파라미터로 받는 것을 확인할 수 있음. 스프링 MVC도 이런 편의 기능을 제공함

**@RequestParam 사용
-> @RequestParam("username")은 request.getParameter("username")와 거의 같은 코드라 생각하면 됨
-> Get 쿼리 파라미터, POST Form 방식을 모두 지원함

**@RequestMapping -> @GetMapping, @PostMapping

-> @RequestMapping은 URL만 매칭하는 것이 아니라 HTTP Methods도 함께 구분할 수 있음
-> 예를 들어서 URL이 /new-form이고 HTTP Method가 GET인 경우를 모두 만족하는 매핑을 하려면 다음과 같이 처리하면 됨
-> @RequestMapping(value = "/new-form", method = RequestMethod.GET) 이것을 @GetMapping, @PostMapping으로 더 편리하게 사용할 수 있음

-> 참고로 Get, Post, Put, Delete, Patch 모두 애노테이션이 준비되어 있음

**@RestController

-> @Controller는 반환 값이 String이면 뷰 이름으로 인식 됨. 그래서 뷰를 찾고 뷰가 랜더링 됨
-> @RestController는 반환값으로 뷰를 찾는 것이 아니라, HTTP 메시지 바디에 바로 입력한다. 따라서 실행 결과로 ok 메시지를 받을 수 있음

**@ResponseEntity

-> Spring 에서 제공하는 HTTP 응답을 나타내는 클래스임
-> REST 방식으로 호출하는 경우 화면 자체가 아니라 데이터 자체를 전송하는 방식으로 처리되기 때문에 데이터를 요청하는 쪽에서는 정상적인 데이터인지 비정상적인 데이터인지를 구분할 수 있는 방법을 제공해야 함

**ResponseEntity는 데이터와 함께 HTTP 헤더의 상태 메시지 등을 같이 전달하려는 용도와 사용함. HTTP의 상태 코드와 에러 메시지 등을 함께 데이터를 전달할 수 있기 때문에 받는 입장에서는 확실히 알 수 있음

**ResponseEntity<Void>에서 'Void'는 해당 응답이 어떠한 데이터를 반환하지 않음을 나타냄. 즉 해당 메서드가 실행되고 나면 클라이언트에게 어떠한 데이터도 전달되지 않고 단순히 성공 여부만을 나타낼 것임

@PathVariavble
스프링 MVC에서는 @PathVariable 어노테이션을 사용해서 URL 상에 경로의 일부를 파라미터로 사용할 수 있음

http://localhost:8080/sample/{sno}

위의 URL에서 '{ }' 처리된 부분은 컨트롤러의 메서드에서 변수로 처리가 가능함.
@PathVariable은 '{ }'의 이름을 처리할 때 사용함

**@AllArgsConstructor: 선언된 모든 필드를 파라미터로 갖는 생성자를 자동으로 만들 어 줌
**@NoArgsConstructor: 파라미터가 아예 없는 기본 생성자를 자동으로 만들어준다

**@RequiredArgsConstructor

의존성 주입에는 크게 3가지 방법이 존재함

1) 필드 주입
2) setter 주입
3) 생성자 주입

이 중 스프링에서 공식적으로 추천하는 방법은 생성자 주입임. 그 이유는 바로 한 번 의 존성을 주입받은 객체는 프로그램이 끝날때까지 변하지 않는 특징을 가지므로 불변 성을 표시해두는 것이 좋음. 그래서 의존성을 주입할 객체는 final 키워드를 사용하는 것임

근데 매번 @Autowired 애노테이션을 사용하고 생성자를 생성해주고 이러기에는 번거로움이 발생함. 그래서 @RequiredArgsConstructor이 발생함. final 혹은 @NotNull이 붙은 필드에 생성자를 만들어 줌

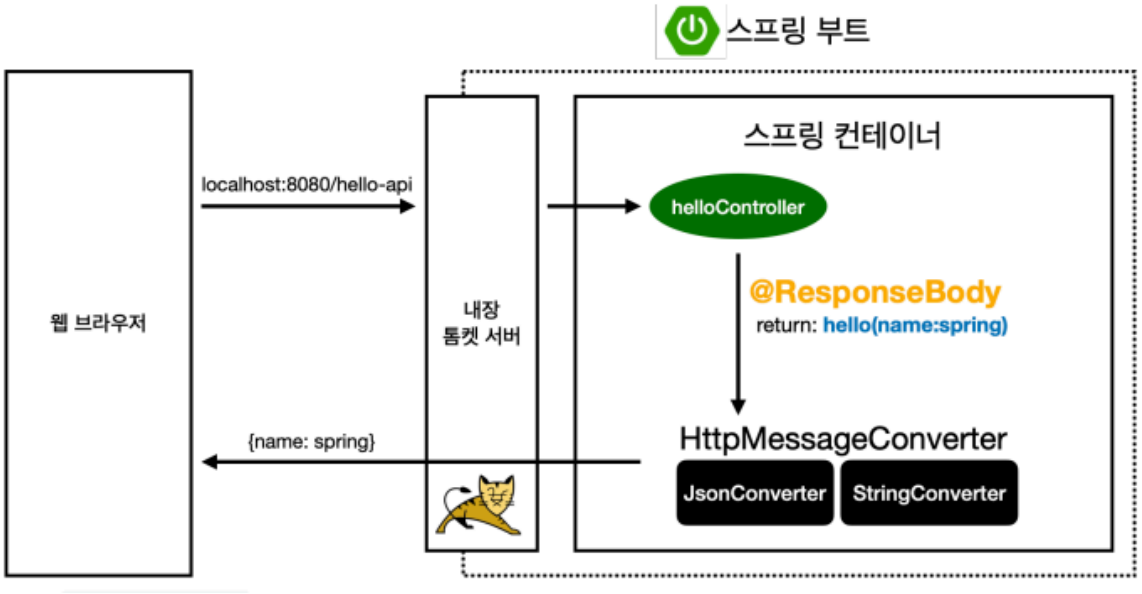
```
// @RequiredArgsConstructor 사용함
@Service
@RequiredArgsConstructor
public class TestService {
  private final TestRepository1 testRepository1;
  private final TestRepository2 testRepository2;
}

// @RequiredArgsConstructor 사용 안 함 @생성자 방식
@Service
public class TestService {
  private final TestRepository1 testRepository1;
  private final TestRepository2 testRepository2;
}

@Autowired
  public class TestService(TestRepository1 testRepository1, TestRepository2
testRepository2)  {
        this.testRepository1 = testRepository1;
        this.testRepository2 = testRepository2;
}
```

****@RequestBody vs @ResponseBody:**

-> @RequestBody 애노테이션은 HTTP 요청의 body 내용을 자바 객체로 매핑하는 역할을 하고 @ResponseBody 애노테이션은 자바 객체를 HTTP 요청의 body 내용으로 매핑하는 역할을 함 (클라이언트 측에서는 요청 데이터를 body에 담고 content-type을 application/json으로 설정해줘야 함)



-> @ResponseBody를 사용하면 뷰 리졸버(viewResolver)를 사용하지 않고

HTTPMessageConverter가 동작함

-> RequestMapping을 처리하는 핸들러 어댑터인 RequestMappingHandlerAdapter(요청 매핑 핸들러 어댑터)에 있음

-> 기본 문자처리: StringHttpMessageConverter

- String 문자로 데이터를 처리함
- 클래스 타입: String, 미디어타입: /
- 요청 예) @RequestBody String data
- 응답 예) @ResponseBody return “ok” 쓰기, 미디어타입: text/plain

-> 기본 객체처리: MappingJackson2HttpMessageConverter

- application/json
- 클래스 타입: 객체 또는 HashMap, 미디어타입: application/json
- 요청 예) @RequestBody Hellodata data
- 응답 예) @ResponseBody return helloData 쓰기, 미디어타입: application/json

-> byte [ ] 처리: ByteArrayMessageConveter

- 클래스 타입: byte[ ], 미디어 타입: */*
- 요청 예) @RequestBody byte[ ] data
- 응답 예) @ResponseBody return byte[ ] 쓰기, 미디어타입: application/octet-stream

-> 대신에 HTTP의 BODY에 문자 내용을 직접 반환함
-> @ResponseBody를 사용하고 객체를 반환하면 객체가 JSON으로 변환됨

-> @RequestBody를 사용하면 HTTP 메시지 바디 정보를 편리하게 조회할 수 있음. 참고로 헤더 정보가 필요하면 HttpEntity를 사용하거나 @RequestHeader를 사용하면 됨

**요청 파라미터 vs HTTP 메시지 바디

-> 요청 파라미터를 조회하는 기능: @RequestParam, @ModelAttribute
-> HTTP 메시지 바디를 직접 조회하는 기능: @RequestBody

**@ResponseBody를 사용하면 응답 결과를 HTTP 메시지 바디에 직접 담아서 전달할 수 있음. 물론 이 경우에도 view를 사용하지 않음

@Builder

@Builder Annotation은 Builder Pattern으로 객체를 생성할 수 있게 해주는 Annotation임. 객체를 생성하는 방법 중에 하나이며 다른 방법에 비해 직관적이고 명시적이기 때문에 가독성을 높여주고 setter가 없기 때문에 객체의 불변성을 보장하여 데이터의 일관성을 유지할 수 있음

1) 생성자를 활용한 객체 생성
TestVo test = new TestVO("권지현", 35, "010-8984-7604", "jihyunkwon@0326@gmail.com")

2) setter를 활용한 객체 생성
TestVo test = new TestVO();
test.setName("권지현");
test.setAge(35);
test.setPhone("010-8984-7604")
test.setEmail("jihyunkwon@0326@gmail.com")

3) Builder를 통한 객체 생성

```
@Builder
public class TestVO {
  private final String name;
  private final int age;
  private final String phone;
  private final String email;
}

TestVO test = TestVO.builder()
                        .name("권지현")
                        .age(35)
                        .phone("010-8984-7604")
                        .email("jihyunkwon@0326@gmail.com").build;
```

Component Scan은 @Component 뿐만 아니라 다음도 추가 대상임 (스프링 빈으로 자동 등록됨)
->@Component, @Controller, @Service, @Repository, @Configuration

@Autowired
1) 의존관계 자동 주입함

@Component
1) Component Scan에서 사용함

@Controller
1) Spring MVC 컨트롤러에서 사용함
2) Spring MVC 컨트롤러 인식함

@Service
1) Spring 비즈니스 로직에서 사용함
2) 특별한 처리를 하지 않음. 대신 개발자들이 핵심 비즈니스 로직이 여기에 있겠구나 비즈니스 계층을 인식하는데 도움이 됨

@Repository
1) Spring 데이터 접근 계층에서 사용함
2) Spring 데이터 접근 계층으로 인식하고 데이터 계층의 예외를 스프링 예외로 변환해줌

@Configuration
1) Spring 설정 정보에서 사용함
2) Spring 설정 정보로 인식하고 스프링 빈이 싱글톤을 유지하도록 추가 처리함

**@Qualifier
-> @Autowired -> Interface를 입력하게 되는데 Interface를 상속 받는 구현체가 여러 개 있을 경우 구현체 하나를 지정해줌

**@SpringBootTest: 스프링 컨테이너와 테스트를 함께 실행함

**@Transactional: 테스트 케이스에 이 애노테이션이 있으면 테스트 시작 전에 트랜잭션을 시작하고 테스트 완료 후에 항상 롤백함. 이렇게 DB에 데이터가 남지 않으므로 다음 테스트에 영향을 주지 않음

**@RequestParam: 스프링에서는 HTTP 요청 파라미터 값을 편리하게 사용하게 해주는 @RequestParam을 지원함

-> HttpServletRequest의 request.getParameter의 기능과 동일함
-> String, int, 등 단순타입이면 @RequestParam 생략 가능함
-> @RequestParam(required = true) 필수임

-> @RequestParam(required = false) 필수가 아님. 값이 없으면 null로 저장되고 int를 사용하게 되면 오류가 발생함

**@Data: @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor를 자동으로 적용해 줌

**@Transactional: 비즈니스 로직을 담당하는 서비스 계층 클래스에 @Transactional 어노테이션을 선언함. 로직을 처리하다가 에러가 발생하였다면 변경된 데이터를 로직을 수행하기 이전 상태로 콜백 시켜줌

**@EqualsAndHashCode: 이 애노테이션을 사용하면 자동으로 아래 메소드를 오버라이딩 해서 사용할 수 있음

equals: 두 객체의 내용이 같은지 동등성(equality)을 비교함

hashCode: 두 객체가 같은 객체인지 동일성(identity)을 비교함

callSuper 속성을 통해 equals와 hashCode 메소드 자동 생성 시 부모 클래스의 필드까지 감안할지의 여부를 설정할 수 있음

@EqualsAndHasCode(callSuper = true)로 설정시 부코 클래스 필드 값들도 동일한치 체크하며 false(기본값)일 경우 자신 클래스의 필드 값만 고려함

***HttpEntity:**

**HTTP header, body 정보를 편리하게 조회함
-> 메시지 바디 정보를 직접 조회함
-> 요청 파라미터를 조회하는 기능과 관계 없음. @RequestParam X, @ModelAttribute X

**HttpEntity는 응답에도 사용 가능함
-> 메시지 바디 정보 직접 반환함
-> 헤더 정보 포함 가능함
-> view 조회 X

HttpEntity를 상속받은 다음 객체들도 같은 기능을 제공함

**RequestEntity
-> HttpMethod, url 정보가 추가 요청에서 사용함

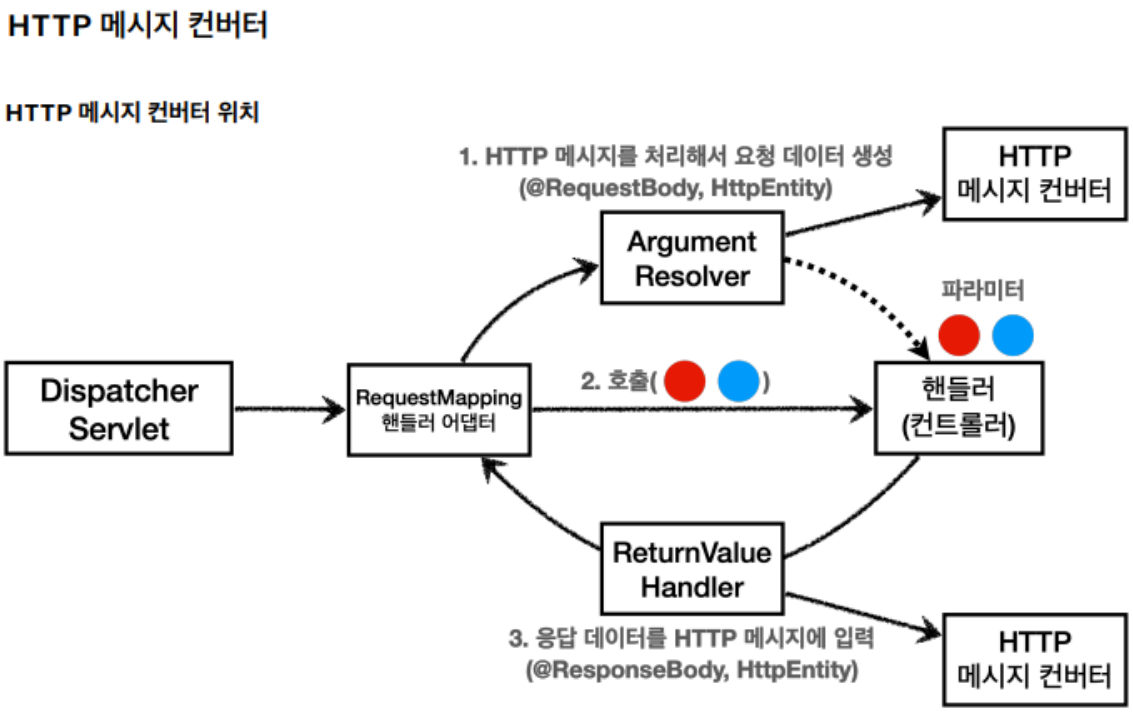
**ResponseEntity
-> HTTP 상태 코드 설정 가능, 응답에서 사용함
-> return new ResponseEntity<String>(“Hello World”, responseHeaders, HttpStatus.CREATED)

**스프링 MVC 내부에서 HTTP 메시지 바디를 읽어서 문자나 객체로 변환해서 전달해주는데 이 때 HTTP 메시지컨버터(HttpMessageConverter)라는 기능을 사용함

***Jar:**

-> Packaging은 War이 아니라 Jar를 선택해야함. JSP를 사용하지 않기 때문에 Jar를 사용하는 것이 좋음. 스프링 부트를 사용하면 이 방식을 주로 사용함. Jar를 사용하면 항상 내장 서버 (톰캣등)를 사용하고 webapp 경로도 사용하지 않음. 내장 서버 사용에 최적화 되어 있는 기능임. War를 사용하면 내장 서버도 사용 가능하지만 주로 외부 서버에 배포하는 목적으로 사용함

***HTTP** 메시지 컨버터**:**



HTTP 메시지 컨버터는 어디쯤 있을까?

-> HTTP 메시지 컨버터를 사용하는 @RequestBody도 컨트롤러가 필요로 하는 파라미터의 값에 사용 됨. @ResponseBody의 경우도 컨트롤러의 반환 값을 이용함

요청의 경우 @RequestBody를 처리하는 ArgumentResolver가 있고 HttpEntity를 처리하는 ArgumentResolver 있음. 이 ArgumentResolver들이 HTTP 메시지 컨버터를 사용해서 필요한 객체를 생성하는 것임

응답의 경우 @ResponseBody와 HTTPEntity를 처리하는 ReturnValueHandler가 있다. 그리고 여기서 HTTP 메시지 컨버터를 호출해서 응답 결과를 만듦

스프링 MVC는 @RequestBody @ResponseBody가 있으면 RequestResponseBodyMethodProcessor() HttpEntity가 있으면 HttpEntityMethodProcessor()를 사용함

스프링은 다음을 모두 인터페이스를 제공함

- HandlerMethodArgumentResolver
- HandlerMethodReturnValueHandler
- HttpMessageConverter

스프링이 필요한 대부분의 기능을 제공하기 때문에 실제 기능을 확장한 일이 많지는 않음. 기능 확장은 WebMvcConfigurer를 상속 받아서 스프링 빈으로 등록하면 됨. 실제 자주 사용하지는 않으니 실제 기능 확장이 필요할 때 WebMvcConfigurer를 검색함

***엔티티(Entity):** 데이터베이스의 테이블에 대응하는 클래스임. @Entity가 붙은 클래스는 JPA에서 관리하며 엔티티라고 함

**엔티티 매니저 팩토리 (Entity Manager Factory): 엔티티 매니저 인스턴스를 관리하는 주체임. 애플리케이션 실행 시 한 개만 만들어지며 사용자로부터 요청이 오면 엔티티 매니저 팩토리부터 엔티티 매니저를 생성함

**엔티티 매니저(Entity Manager): 영속성 컨텍스트에 접근하여 엔티티에 대한 데이터베이스 작업을 제공함. 내부적으로 커넥션을 사용해서 데이터베이스에 접근함

**영속성 컨텍스트(Persistence Context): 엔티티를 영구 저장하는 환경으로 엔티티 매니저를 통해 영속성 컨텍스트에 접근함. 이렇게 애플리케이션과 데이터베이스 사이에 중간 계층을 만들면 버퍼링, 캐싱 등을 할 수 있는 장점이 있음

***Form** 객체와 **DTO** 따로 생성해서 사용하는 이유**?:**

-> 역할 분리 및 단일 책임 원칙(Single Responsibility Principle): 각 객체는 특정한 역할을 수행하며 서로 간의 역할이 분리됨. Form 객체는 주로 사용자 인터페이스에서 사용자로부터의 입력을 수집하는 데 사용되며 DTO는 비즈니스 로직에서 필요한 데이터베이스 전송을 위한 목적으로 사용됨

***CSRF(Cross Site Request Forgery)**: 사이트간 위조 요청으로 사용자가 자신의 의지와 상관없이 해커가 의도한 대로 수정, 등록, 삭제 등의 행위를 웹사이트에 요청하게 만드는 공격임

-> Spring Security를 사용할 경우 기본적으로 CSRF를 방어하기 위해 모든 POST 방식의 데이터 전송에는 CSRF 토큰 값이 있어야 함. CSRF 토큰은 실제 서버에서 허용한 요청이

맞는지 확인하기 위한 토큰임. 사용자의 세션에 임의의 값을 저장하여 요청마다 그 값을 포함하여 전송하면 서버에서 세션에 저장된 값과 요청이 온 값이 일치하는지 확인하여 CSRF를 방어함

***Repository**: DAO(Data Access Object)로 DB의 데이터에 접근하기 위한 객체임

-> Mapper도 이에 해당함
-> Spring Data JPA는 이렇게 인터페이스만 작성하면 런타임 시점에 자바의 Dynamic Proxy를 이용해서 객체를 동적으로 생성해줌. 따로 Data Access Object와 xml 파일에 쿼리문을 작성하지 않아도 됨

***ORM(Object Relational Mapping)**: 자바 클래스와 RDB(Relational DataBase)의 테이블을 연결시켜주는 것임. 기술적으로는 애플리케이션의 객체를 RDB 테이블에 자동으로 영속화 해주는 것임

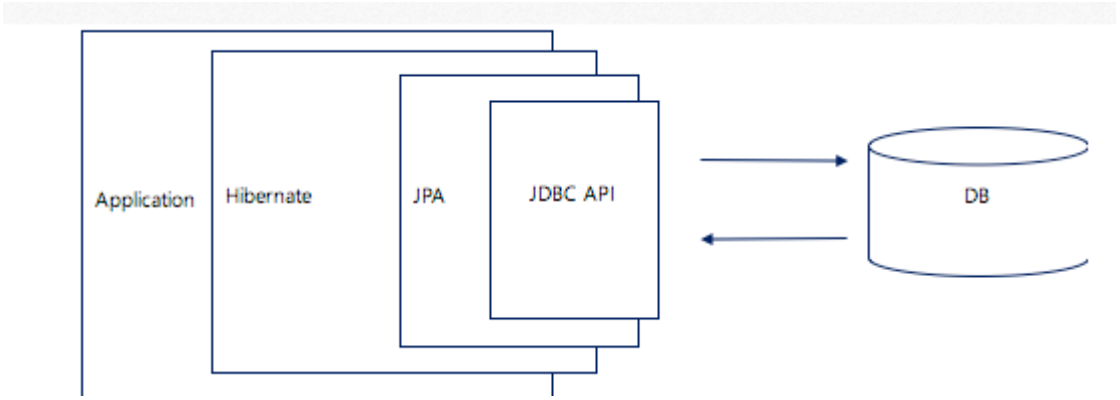
***JPA(Java Persistence API)**: 자바에서 ORM을 사용하기 위해 여러 인터페이스를 모아둔 것

-> 자바는 객체 지향 패러다임으로 만들어졌고 관계형 데이터베이스는 데이터를 정규화해서 잘 보관하는 것을 목표로 함 근데 JPA 전에는 복잡한 SQL문을 통해 데이터베이스에 접근함. 결국 객체를 단순히 데이터 전달 목적으로 사용할 뿐 객체지향적으로 프로그래밍할 수 없음. 이를 해결하기 위해 ORM 등장함. 객체는 객체지향적으로, 데이터베이스는 데이터베이스 대로 설계를 함 그리고 ORM은 중간에서 2개를 매핑함

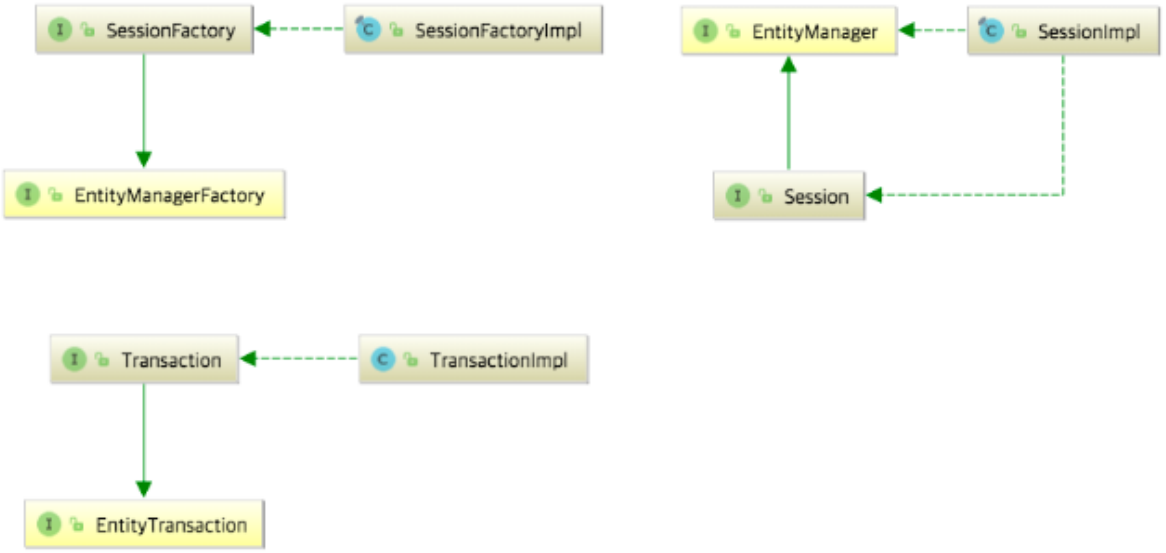
->JSP 기반의 구현체는 1) 하이버네이트, 2) 이클립스 링크, 3) 데이터 뉴클리어스가 있으며 그 중 가장 많이 사용되는 구현체는 하이버네이트임

***하이버네이트(Hibernate)**: 자바의 ORM 프레임워크로 JPA 기반의 구현체임

-> SQL을 사용하지 않고 직관적인 코드(메소드)를 사용해 데이터를 조작할 수 있음
-> SQL을 직접 사용하지 않는다고 JDBC API를 사용하지 않는 것이 아니라 Hibernate가 지원하는 메소드 내부에서 JDBC API가 동작하고 있으며 단지 개발자가 직접 SQL을 작성하지 않음

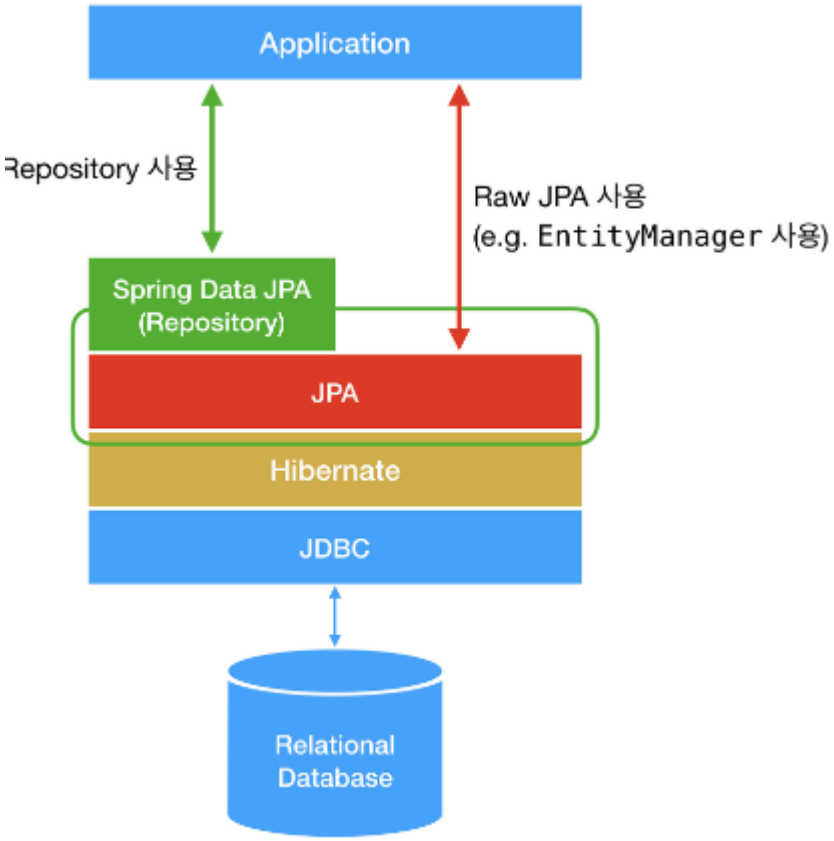


-> JPA와 Hibernate는 자바의 interface와 해당 interface를 구현한 class와 같은 관계임



-> JPA의 핵심인 EntityManagerFactory, EntityManager, EntityTransaction을 Hibernate에서는 SessionFactory, Session, Transaction을 상속받고 각각 Impl로 구현하고 있음

***Spring Data JPA**: Spring에서 제공하는 모듈 중 하나로 JPA를 쉽고 편하게 사용할 수 있도록 도와줌



-> 기존에 JPA를 사용하려면 EntityManager를 주입받아 사용해야 했지만 Spring Data JPA는 한 단계 더 추상화 시킨 Repository 인터페이스를 제공함
-> Spring Data JPA가 JPA를 추상화 했다는 말은 Spring Data JPA의 Repository의 구현에서 JPA를 사용하고 있다는 말임. 사용자가 Repository 인터페이스에 정해진 규칙대로 메소드를 입력하면 Spring이 알아서 해당 메소드 이름에 적합한 쿼리를 날리는 구현체를 만들어서 Bean으로 등록해줌

*영속성 컨텍스트**(Persistence Context)**: 엔티티를 영구 저장하는 환경이라는 뜻임.
애플리케이션과 데이터베이스 사이에서 객체를 보관하는 가상의 데이터베이스 같은 역할을 함. 엔티티 매니저를 통해 엔티티를 저장하거나 조회하면 엔티티 매니저는 영속성 컨텍스트에 엔티티를 보관하고 관리함

**em.persist(member): 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다는 의미임

**영속성 컨텍스트의 특징
-> 엔티티 매니저를 생성할 때 하나 만들어짐
-> 엔티티 매니저를 통해서 영속성 컨텍스트에 접근하고 관리할 수 있음

*연관 관계 매핑: 예를 들어 Member 엔티티와 Team 엔티티가 있을 때 하나의 Team은 여러 Member를 갖는 관계를 가지고 있음. 객체의 참조와 테이블의 외래 키를 매핑하는 것을 의미함

**방향

단방향 관계: 두 엔티티가 관계를 맺을 때 한 쪽의 엔티티만 참조하는 것을 의미함
양방향 관계: 두 엔티티가 관계를 맺을 때 양 쪽이 서로 참조하는 것을 의미함

**다중성

관계가 있는 두 엔티티는 다음 중 하나의 관계를 갖습니다

1) ManyToOne: 다대일 (N :1)
2) One to Many: 일대다 (1 : N)
3) OneToOne: 일대일 (1 : 1)
4) ManytoMany: 다대다 (N : N)

예를 들어 하나의 Team은 여러 Member 구성원으로 갖고 있으므로 Team 입장에서는 Member와 일대다 관계이며 Member의 입장에서는 하나의 Team에 속하므로 다대일 관계임
즉 어떤 엔티티를 중심으로 상대 엔티티를 바라보느냐에 따라 다중성이 다르게 됩니다

**연관관계의 주인 (Owner)

-> 객체를 양방향 연관관계로 만들면 연관관계의 주인을 정해야 함. 연관관계를 갖는 두 테이블에서 외래키를 갖게 되는 테이블이 연관관계의 주인이 됨
연관관계의 주인만이 외래 키를 관리(등록, 수정, 삭제) 할 수 있고, 주인이 아닌 엔티티는 읽기만 할 수 있음

*@JoinColumn(name="TEAM_ID"): @JoinColumn 애노테이션은 외래 키를 매핑할 때 사용함. name 속성에는 매핑 할 외래 키 이름을 지정함

-> 예를 들어 Member 엔티티의 경우 Team 엔티티의 id 필드를 외래 키로 가지므로, TEAM_ID를 작성하였음

-> 연관 관계의 주인을 정하는 방법은 mappedBy 속성을 지정하는 것임
주인은 mappedBy 속성을 사용하지 않고 @JoinColumn을 사용함. 주인이 아닌 엔티티 클래스는 mappedBy 속성을 사용해 주인을 정할 수 있음

***Optional**:

-> 개발을 하다보면 NullPointErexception(NPE)을 만나게 된다. 가장 많이 발생하는 에러 중 하나라고 하는데, 이를 피하기 위해서는 null을 체크하는 로직이 추가되어야 한다. null 체크해야 될 부분이 많다면 코드가 복잡해져 가독성이 떨어지는 단점이 있음

Java 8에서는 Optional<T> 클래스를 도입하였는데 '존재할 수도 있지만 안 할 수도 있는 객체'로 null이 될 수도 있는 객체를 감싸고 있는 Wrapper 클래스임

Optional로 객체를 감싸서 사용하면 NPE 방지를 위해 null 체크를 직접 하지 않아도 되며 명시적으로 해당 변수가 null일 수도 있다는 가능성을 포함하고 있기 때문에 불필요한 방어 로직을 줄일 수 있음

*인증(Authentication) vs 인가(Authorization):

**인증(Authentication): 본인이 맞는지 확인하는 절차

**인가(Authorization): 인증된 사용자가 요청한 자원에 접근 가능한가를 결정하는 절차 (로그인이 유지되는 상태에서 일어나는 일)

*패키지 구조 설계:

package 구조

- hello.login
  - domain
    - item
    - member
    - login
  - web
    - item
    - member
    - login

도메인이 가장 중요함

도메인: 화면, UI, 기술 인프라 등등의 영역을 제외한 시스템이 구현해야 하는 핵심 비즈니스 업무 영역을 말함

향후 web을 다른 기술이 바꾸어도 도메인은 그대로 유지할 수 있어야 함
이렇게 하려면 web은 domain을 알고 있지만 domain은 web을 모르도록 설계해야 함. 이것을 web은 domain을 의존하지만 domain은 의존하지 않는다고 표현함. 예를 들어 web 패키지들 모두 삭제해도 domain에는 전혀 영향이 없도록 의존관계를 설계하는 것이 중요함. 반대로 이야기하면 domain은 web을 참조하면 안 됨

*예외(Exception) vs 에러(Error):