Cookie, Session, Filter, Interceptor

524730-1
2024년 봄학기
4/10/2024
박경신

HTTP

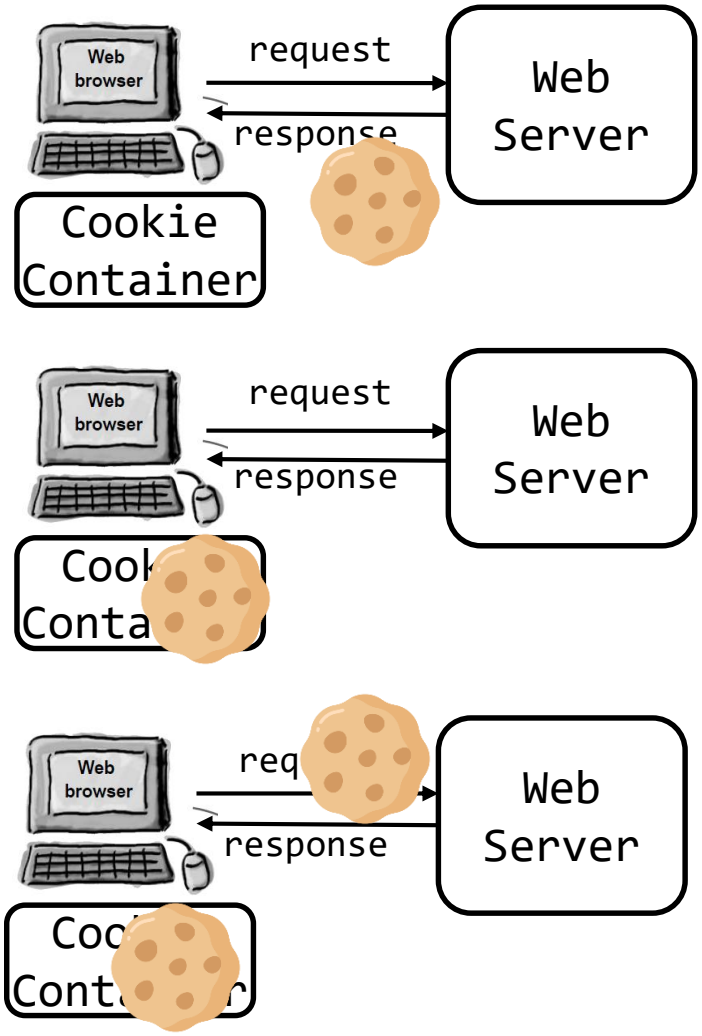
- HTTP 특징
  - HTTP는 클라이언트가 서버에게 요청(request)을 보내면, 서버는 응답(response)을 보냄으로써 데이터를 교환
  - HTTP는 비연결성(Connectionless) 및 무상태성(Stateless) 특징
    - HTTP는 요청 처리 후 연결을 끊어버리기 때문에, 클라이언트의 상태 정보 및 현재 통신 상태가 남아있지 않음.
    - **비연결성**은 서버의 **자원 낭비를 줄일 수 있다**는 장점이 있음
    - **비연결성**은 **클라이언트를 식별할 수 없다**는 단점이 존재함
    - 클라이언트의 상태를 유지하기 위해 쿠키(Cookie), 세션(Session) 등을 이용.

쿠키

- 쿠키
  - 쿠키(Cookie)란 클라이언트가 어떠한 웹사이트를 방문할 경우, 그 사이트가 사용되고 있는 서버를 통해 클라이언트 브라우저에 설치되는 작은 기록 파일임.
  - 서버는 클라이언트 로그인 요청에 대한 응답을 작성할 때, 클라이언트에 저장하고 싶은 정보를 응답 헤더의 SetCookie 저장.
  - 이후 해당 클라이언트는 요청을 보낼 때마다 매번 저장된 쿠키를 요청 헤더의 Cookie에 담아 보냄.
  - 서버는 쿠키에 담긴 정보를 바탕으로 해당 요청의 클라이언트가 누군지 식별할 수 있음.
  - 보안에 취약함.
  - 용량 제한이 있어, 많은 정보를 담을 수 없음.
  - 웹 브라우저마다 쿠키에 대한 지원 형태가 다르기에, 브라우저 간 공유가 불가능함.

쿠키의 동작 과정

- 쿠키(cookie) 동작 과정
  1. **쿠키 생성** 단계 - 쿠키를 사용하려면 먼저 쿠키를 생성해야 함. 쿠키는 주로 웹 서버 측에서 생성. 생성된 쿠키는 응답 데이터에 함께 저장되어 웹 브라우저에 전송됨.
  2. **쿠키 저장** 단계 - 웹 브라우저는 응답 데이터에 포함된 쿠키를 쿠키 저장소에 보관. 쿠키는 종류에 따라 메모리나 파일로 저장.
  3. **쿠키 전송** 단계 - 웹 브라우저는 한 번 저장된 쿠키를 요청이 있을 때마다 웹 서버에 전송. 웹 서버는 웹 브라우저가 전송한 쿠키를 사용하여 필요한 작업을 수행할 수 있음.



쿠키의 구성

- 쿠키(cookie) 구성 요소
  - 이름 - 각각의 쿠키를 구별하는 데 사용되는 이름
  - 값 - 쿠키의 이름과 관련된 값
  - 유효시간 - 쿠키의 유지 시간
  - 도메인 - 쿠키를 전송할 도메인
  - 경로 - 쿠키를 전송할 요청 경로
- 쿠키 이름의 제약
  - 쿠키의 이름은 **아스키 코드의 알파벳과 숫자만**을 포함가능
  - **콤마(,), 세미콜론(;), 공백(' ') 등의 문자는 불가능**
  - **'$'로 시작할 수 없음**

쿠키의 구성

- 쿠키(cookie) 구성 요소
  - 이름 - 각각의 쿠키를 구별하는 데 사용되는 이름
  - 값 - 쿠키의 이름과 관련된 값
  - 유효시간 - 쿠키의 유지 시간
  - 도메인 - 쿠키를 전송할 도메인
  - 경로 - 쿠키를 전송할 요청 경로
- 쿠키 이름의 제약
  - 쿠키의 이름은 **아스키 코드의 알파벳과 숫자만**을 포함가능
  - **콤마(,), 세미콜론(;), 공백(' ') 등의 문자는 불가능**
  - **'$'로 시작할 수 없음**

쿠키의 생성

- 쿠키 종류
  - 영속 쿠키: 만료 날짜를 입력하면 해당 날짜까지 유지
  - 세션 쿠키: 만료 날짜를 생략하면 브라우저 종료 시 까지만 유지
- 쿠키 생성
  - Cookie 클래스를 이용해서 쿠키를 생성한 후
    - 첫 번째 매개변수 name은 쿠키를 식별하기 위한 이름 (String)
    - 두 번째 매개변수 value는 쿠키 값 (String)
  - response의 addCookie() 메소드로 쿠키를 설정해야 함

```
// add cookie
Cookie cookieID = new Cookie("userID", loginUser.getLoginId());
Cookie cookiePW = new Cookie("userPW", loginUser.getPassword());
response.addCookie(cookieID);
response.addCookie(cookiePW);
```

쿠키의 정보 얻기

- 클라이언트가 보낸 쿠키 읽기
  - 쿠키 객체가 여러 개일 때는 배열 형태로 가져옴

```
Cookie[] cookies = request.getCookies();
```

- 쿠키 객체의 정보 얻기
  - 쿠키 객체를 얻어왔다면 이 쿠키 객체에 저장된 쿠키 이름과 값을 가져오기 위해 getName(), getValue() 메소드를 사용

메서드	설명
String getName()	쿠키의 이름을 구한다.
String getValue()	쿠키의 값을 구한다.

쿠키의 정보 얻기

```
// get userID & userPW from cookies
String userID = null;
String userPW = null;
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (int i = 0; i < cookies.length; i++) {
        String name = cookies[i].getName();
        String value = cookies[i].getValue();
        System.out.println("쿠키 속성이름 " + cookies[i].getName());
        System.out.println("쿠키 속성 값 " + cookies[i].getValue());
        System.out.println("----------------------------------------");
        if (name.equals("userID")) {
            userID = value;
        }
        if (name.equals("userPW")) {
            userPW = value;
        }
    }
}
```

쿠키의 정보 얻기

- @CookieValue
  - @CookieValue를 사용하면 name으로 설정되어 있는 쿠키를 꺼내올 수 있음.
  - 쿠키값이 없어도 들어오게 하기 위해 required=false를 씀.

```
@GetMapping("/")
public String home(@CookieValue(name = "userID", required = false) String userID, @CookieValue(name = "userPW", required = false) String userPW, Model model) {
        // if login user is NOT found in cookie, send to cookie/home.html
        if (userID == null || userPW == null) {
            return "cookie/home"; // templates/cookie/home.html
        }

        // if login success, view cookie/loginsuccess.html
        model.addAttribute("loginId", userID);
        return "cookie/loginsuccess";    // templates/cookie/loginsuccess.html
}
```

쿠키 값의 인코딩/디코딩 처리

- 쿠키는 값으로 한글과 같은 문자를 가질 수 없음
  - 쿠키의 값을 인코딩해서 지정할 필요 있음
- 쿠키 값의 처리
  - 값 설정시 : URLEncoder.encode("값", “Euc-kr")
    - new Cookie("name", **URLEncoder.encode("값", “Euc-kr")**);
  - 값 조회시 : URLDecoder.decode("값", “Euc-kr")
    - Cookie cookie = …;
      String value = **URLDecoder.decode(cookie.getValue(), “Euc-kr");**

쿠키 값 변경

- 기존에 존재하는 지 확인 후, 쿠키 값 새로 설정

```
Cookie[] cookies = request.getCookies();
if (cookies != null && cookies.length > 0) {
    for (int i = 0 ; i < cookies.length ; i++) {
        if (cookies[i].getName().equals("name")) {
            Cookie cookie = new Cookie(name, value);
            response.addCookie(cookie);
        }
    }
}
```

쿠키의 도메인과 경로

- 도메인 지정시, 해당 도메인에 쿠키 전달
  - Cookie.setDomain()으로 쿠키 설정
  - 도메인 형식
    - .funschool.net - 점으로 시작하는 경우 관련 도메인에 모두 쿠키를 전송함.
    - java.funschool.net - 특정 도메인에 대해서만 쿠키를 전송함.
- 웹 브라우저는 도메인이 벗어난 쿠키는 저장하지 않음
- 쿠키 도메인에 따라 쿠키가 전달됨

쿠키의 경로 / 유효 시간

- 경로 설정 시 해당 경로를 기준으로 쿠키 전달
  - 경로 미 설정 시, 요청 URL의 경로에 대해서만 쿠키 전달
  - 경로 설정 시, 설정한 경로 및 그 하위 경로에 대해서 쿠키 전달
  - Cookie.setPath()로 경로 설정
- 유효 시간
  - 유효 시간 미 지정 시, 웹 브라우저 닫을 때 쿠키도 함께 삭제
  - Cookie.setMaxAge()로 쿠키 유효 시간 설정
    - 유효 시간이 지나지 않을 경우 웹 브라우저를 닫더라도 쿠키가 삭제되지 않고, 이후 웹 브라우저를 열었을 때 해당 쿠키 전송됨
    - 유효 시간 : 초 단위로 설정

쿠키의 삭제

- 쿠키의 삭제
  - 쿠키의 유효 기간을 결정하는 setMaxAge() 메소드에 유효 기간을 0으로 설정하여 쿠키를 삭제할 수 있음

```
Cookie cookieID = new Cookie("userID", null);
cookieID.setMaxAge(0);
response.addCookie(cookieID);
```

세션의 개요

- 세션(session)
  - 클라이언트와 웹 서버 간의 상태를 지속적으로 유지하는 방법
    - 예를 들면 웹 쇼핑몰에서 장바구니나 주문 처리와 같은 회원 전용 페이지의 경우 로그인 인증을 통해 사용 권한을 부여. 그래서 다른 웹 페이지에 갔다가 되돌아와도 로그인 상태가 유지되므로 회원 전용 페이지를 계속 사용할 수 있음. 이렇게 사용자 인증을 통해 특정 페이지를 사용할 수 있도록 권한 상태를 유지하는 것.
  - **오직 서버에서만 생성**
  - **클라이언트마다 세션이 생성**

세션의 개요

- 세션(session)
  - 웹 서버에서만 접근이 가능하므로 보안 유지에 유리하며 데이터를 저장하는 데 한계가 없음
  - 오직 웹 서버에 존재하는 객체로 웹 브라우저마다 하나씩 존재하므로 웹 서버의 서비스를 제공받는 사용자를 구분하는 단위가 됨
  - 웹 브라우저를 닫기 전까지 웹 페이지를 이동하더라도 사용자의 정보가 웹 서버에 보관되어 있어 사용자 정보를 잃지 않음.

세션의 동작 과정

- 세션 동작 과정
  1. **클라이언트가 서버에 접속 시 세션 ID를 발급**.
  2. 서버에서는 클라이언트로 발급해준 **세션 ID (JSESSIONID)**를 저장. 즉, 세션을 구별하기 위해 ID가 필요하고 **그 ID만 쿠키를 이용해서 저장해놓음**. 쿠키는 자동으로 서버에 전송되니까 서버에서 세션아이디에 따른 처리를 할 수 있음.
  3. 클라이언트는 다시 접속할 때, 이 JSESSIONID를 이용해서 세션 ID값을 서버에 전달. 예를 들면, 게시판에 글을 작성할 때 작성 버튼을 누르면 세션에 있는 아이디를 참조해서 작성자를 지정하게 함

세션의 생성

- 서블릿은 HttpSession을 제공함.
- 서블릿을 통해 HttpSession을 생성하면 쿠키 이름이 JSESSIONID이고, 값은 추정 불가능한 랜덤 값으로 지정됨.
- session 를 생성하고, 속성을 이용해서 클라이언트 관련 정보를 저장함.

```
// if session is already created, return session
// otherwise, create new session
HttpSession session = request.getSession();
// save loginUser to session
session.setAttribute("SESSION_LOGIN_USER", loginUser)
```

세션에 정보 저장

- 세션에 정보 저장
  - session 내장 객체의 setAttribute() 메소드를 사용
  - setAttribute() 메소드를 이용하여 세션의 속성을 설정하면 계속 세션 상태를 유지할 수 있음. 만약 동일한 세션의 속성 이름으로 세션을 생성하면 마지막에 설정한 것이 세션 속성 값이 됨.

    void setAttribute(String name, Object value)
    - 첫 번째 매개변수 name은 세션으로 사용할 세션 속성 이름을 나타내며, 세션에 저장된 특정 값을 찾아오기 위한 키로 사용.
    - 두 번째 매개변수 value는 세션의 속성 값
    - 세션 속성 값은 Object 객체 타입만 가능하기 때문에 int, double, char 등의 기본 타입은 사용할 수 없음

```
session.setAttribute("userName", "Park");
session.setAttribute("userAge", 10);
```

세션의 정보 얻기

- 단일 세션 정보 얻기
  - 세션에 저장된 하나의 세션 속성 이름에 대한 속성 값을 얻어오려면 getAttribute( ) 메소드를 사용
  - getAttribute( ) 메소드는 반환 유형이 Object 형이므로 반드시 형 변환을 하여 사용해야 함

    Object getAttribute(String name)

    - 첫 번째 매개변수 name은 세션에 저장된 세션 속성 이름
    - 해당 속성 이름이 없는 경우 null을 반환

```
String name = (String)session.getAttribute("userName");
int age = (Integer)session.getAttribute("userAge");
```

세션의 정보 얻기

- 다중 세션 정보 얻기
  - getAttributeNames() 메소드를 사용하여 다중 세션 정보를 얻음

```
Enumeration items = session.getAttributeNames();
while(items.hasMoreElements()) {
    String name = items.nextElement().toString();
    String value = session.getAttribute(name).toString();
}
```

세션의 삭제

- 단일 세션 삭제하기
  - 세션에 저장된 하나의 세션 속성 이름을 삭제하려면 removeAttribute( ) 메소드를 사용

    void removeAttribute(String name)

```
session.removeAttribute("userName");
session.removeAttribute("userAge");
```

- 다중 세션 삭제하기
  - 세션에 저장된 모든 세션 속성 이름을 삭제하려면 invalidate( ) 메소드를 사용

    void invalidate()
  - 세션이 종료되면 기존에 생성된 세션이 삭제
  - 이후 접근 시 새로운 세션 생성 됨

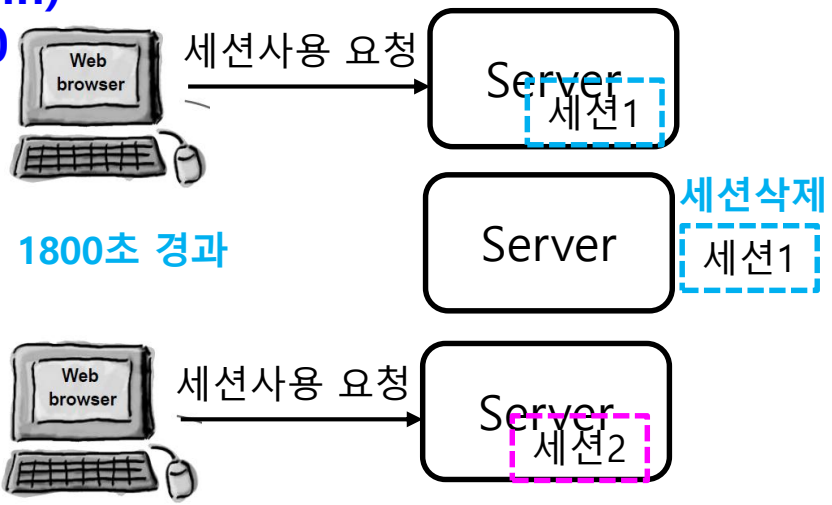
```
session.invalidate(); // 세션 해제
```

세션 유효 시간 설정

- 세션 유효 시간
  - 세션을 유지하기 위한 세션의 일정 시간
  - 웹 브라우저에 마지막 접근한 시간부터 일정 시간 이내에 다시 웹 브라우저에 접근하지 않으면 자동으로 세션이 종료
  - 세션 유효 시간을 설정하기 위해 session 내장 객체의 setMaxInactiveInterval() 메소드를 사용

    void setMaxInactiveInterval(int interval) // 초단위
  - 스프링부트 application.properties 파일에서 지정 **// 초단위**

    #session timeout (1800 sec = 30 min)
    server.servlet.session.timeout=1800
  - 마지막 세션 사용 이후 유효 시간이 지나면 자동 종료



세션의 생성 및 속성 저장

```
@PostMapping("/login")
public String login(@ModelAttribute LoginUser loginUser, Model model,
RedirectAttributes redirectAttributes , HttpServletRequest request) {
        LoginUser user = userRepository. …
        if (user == null) {
            // login failed
            model.addAttribute("msg", "Fail to log in");
            return “/loginform"; // view
        }
        // login success
        // save loginUser to session
        HttpSession session = request.getSession();
        session.setAttribute("SESSION_LOGIN_USER", loginUser);
        // msg
        redirectAttributes.addFlashAttribute("msg","Login Success! Session Creation
Successful~");
        return "redirect:/";
}
```

세션의 정보 얻기

```
@GetMapping("/")
public String home(HttpRequest request, Model model) {
    HttpSession session = request.getSession(false);
    LoginUser loginUser =
                        (LoginUser)session.getAttribute(“SESSION_LOGIN_USER");
    if (loginUser == null) {
        return "/home"; // view
    }
    // print all session data
    session.getAttributeNames().asIterator().forEachRemaining(name ->
log.info("session name={}, value={}", name, session.getAttribute(name)));
    log.info("sessionId={}", session.getId());
    log.info("세션의 유효시간(초)={}", session.getMaxInactiveInterval());
    log.info("세션 생성일시={}", new Date(session.getCreationTime()));
    log.info("최근 서버에 접근한 시간={}", new
Date(session.getLastAccessedTime()));
    log.info("지금 만들어진 세션인가?={}", session.isNew());
    // if login success, view loginsuccess.html
     model.addAttribute("loginId", loginUser.getLoginId());
     return "loginsuccess"; // view
}
```

세션의 정보 얻기

- @SessionAttribute
  - @SessionAttribute를 사용하면 name으로 설정되어 있는 세션속성 정보를 꺼내 올 수 있음.
  - 세션속성 정보가 없어도 처리하기 위해 required=false를 씀.

```
@GetMapping("/")
public String home(@SessionAttribute(name="SESSION_LOGIN_USER",
required=false) LoginUser loginUser, Model model) {
    if (loginUser == null) {
        return "/home"; // view
    }
    // if login success, view loginsuccess.html
     model.addAttribute("loginId", loginUser.getLoginId());
     return "loginsuccess"; // view
}
```

세션의 삭제

```
@PostMapping("/logout")
public String logout(HttpServletRequest request, HttpServletResponse response) {
        // remove session
        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate();
        }
        return "redirect:/";
}
```

세션 TrackingModes

- TrackingModes
  - application.properties에서 세션의 tracking-modes를 cookie로 설정해서 사용할 것.

    # url을 통해 세션을 유지하는 jsessionid가 생성되는 것을 방지

    server.servlet.session.tracking-modes=cookie

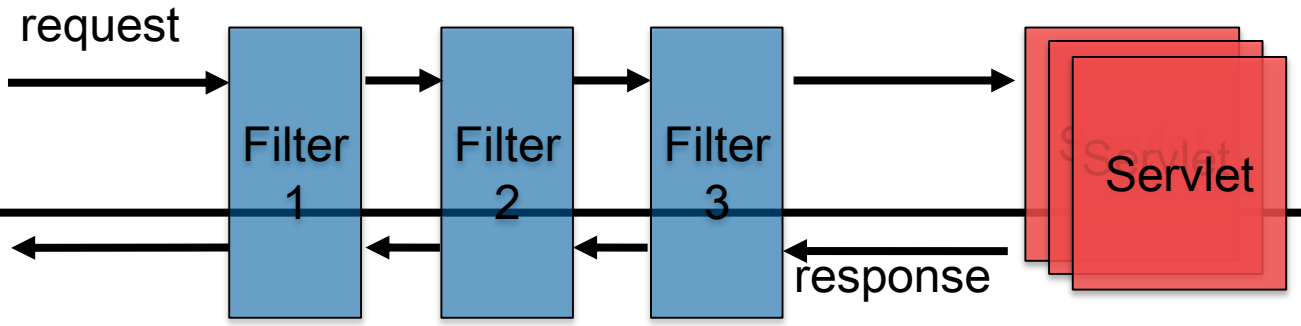
  - 이를 설정하지 않았을 시, 아래와 같이 URL을 통해 세션을 유지하기 위해 처음 웹사이트에 접근을 시도하면 URL이 JSESSIONID를 포함하고 있음. 웹브라우저가 쿠키를 지원하지 않을 경우를 대비하여 쿠키 대신 URL을 통해 세션을 유지하는 기능임.

    http://localhost:8080/;jsessionid=AF3038EAFFD4E56BD2E9C2CFF9E29043

쿠키와 세션의 차이

구분	쿠키	세션
사용클래스	Cookie 클래스	HttpSession 인터페이스
저장 형식	텍스트	Object형
저장 장소	클라이언트	서버(세션아이디만 클라이언트에 저장)
종료 시점	쿠키도 만료시간이 있지만 파일로 저장되기 때문에 웹브라우저를 종료해도 계속해서 정보가 남아 있을 수 있음. 또한 만료기간을 넉넉하게 잡아두면 쿠키삭제를 할 때까지 유지될 수도 있음.	세션도 만료시간을 정할 수 있지만 웹브라우저가 종료되면 만료시간에 상관없이 종료됨.
리소스	클라이언트의 리소스 사용	서버의 리소스 사용
보안	클라이언트에 저장되므로 사용자 변경이 가능하여 보안에 취약	서버에 저장되어 상대적으로 안정적

필터



- 필터(Filter)
  - 클라이언트와 서버 사이에서 request와 response 객체를 먼저 받아 **사전/사후 작업 등 공통적으로 필요한 부분**을 처리하는 것.
  - 필터는 클라이언트 **요청이** 웹서버의 서블릿에 **도달하기 전**과, 반대로 클라이언트로 **응답하기 전**에 필요한 전처리를 수행 가능함.
  - 필터는 HTTP 요청과 응답을 변경할 수 있는 코드로 재사용이 가능함. 한편 여러 개의 필터로 이루어진 **필터 체인**을 제공함
  - HTTP 요청 -> WAS -> 필터1 -> 필터2 -> 필터3 -> 서블릿 -> 컨트롤러

필터	기능
Request 필터	인증(사용자인증) 요청정보를 로그 파일로 작성 암호화 인코딩 작업
Response 필터	응답결과 데이터 압축 응답결과에 내용 추가/수정 총 서비스 시간 측정

필터 인터페이스

- Filter 인터페이스
  - 필터 기능을 구현하는 데 핵심적인 역할을 하는 인터페이스.
  - 클라이언트와 서버의 리소스 사이에 위치한 필터의 기능을 제공하기 위해 자바 클래스로 구현해야 함

```
import javax.servlet.Filter;

public class LogFilter implements Filter {
 // 내부구현
}
```

메서드	설명
init(..)	필터 인스턴스 초기화 메소드
doFilter(..)	필터 기능을 작성하는 메소드
destroy()	필터 인스턴스의 종료 전에 호출되는 메소드

필터 인터페이스 구현클래스

- **public void init(FilterConfig config) throw ServletException**
  - 서블릿 컨테이너가 필터를 초기화할 때 호출되는 메소드
  - init 메소드는 서블릿 컨테이너 내에서 초기화 작업을 수행할 필터 인스턴스를 생성한 후 한 번만 호출
  - init 메소드는 서블릿 컨테이너에 의해 호출되어 필터의 서비스가 시작되고 있음을 나타냄
- **public void destroy()**
  - 필터 인스턴스를 종료하기 전에 호출하는 메소드
  - 서블릿 컨테이너가 필터 인스턴스를 삭제하기 전에 청소 작업을 수행하는 데 사용되며, 이는 필터로 열린 리소스를 모두 닫을 수 있는 방법
  - destroy 메소드는 필터의 수명 동안 한 번만 호출

필터 인터페이스 구현클래스

- **public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throw ServletException**
  - 서블릿 컨테이너가 필터를 리소스에 적용할 때마다 호출되는 메소드
  - init 메소드 후에 호출되며, 필터가 어떤 기능을 수행할 필요가 있을 때마다 호출
  - ServletRequest 객체는 체인을 따라 전달하는 요청
  - ServletResponse 객체는 체인을 따라 전달할 응답
  - **FilterChain** 객체는 체인에서 다음 필터를 호출하는 데 사용
    - 만약 호출 필터가 체인의 마지막 필터이면 체인의 끝에서 리소스를 호출

필터 인터페이스 구현클래스

```
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain chain) throws IOException, ServletException {
    log.info("Before doFilter LogFilter, uri : {} ",
                    ((HttpServletRequest)servletRequest).getRequestURI());

    chain.doFilter(servletRequest, servletResponse);

    log.info("AFTER doFilter LogFilter, uri : {}",
                    ((HttpServletRequest)servletRequest).getRequestURI());
}
```

WebConfig 클래스

- **WebConfig** 사용
  - @Component 스프링컨테이너에 @Bean을 등록하기 위함
  - setFilter 필터 등록
  - setOrder 필터 적용 순서
  - addUrlPatterns 적용할 URL

```
@Component // @ComponentScan 사용시 호출
public class WebConfig {
    @Bean
    public FilterRegistrationBean logFilter() {
        FilterRegistrationBean<Filter> filterRegistrationBean =
                                                    new FilterRegistrationBean<>();
        filterRegitrationBean.setFilter(new LogFilter()); // logFilter 등록
        filterRegitrationBean.setOrder(1); // 순서 등록
        filterRegitrationBean.addURLPatterns("/*"); // 모든 url에 다 적용
        return filterRegitrationBean;
    }
…
```

WebConfig 클래스

```
@Component // @ComponentScan 사용시 호출
public class WebConfig {
    @Bean
    public FilterRegistrationBean logFilter() {
        FilterRegistrationBean<Filter> filterRegistrationBean =
                                                    new FilterRegistrationBean<>();
        filterRegitrationBean.setFilter(new LogFilter()); // logFilter 등록
        filterRegitrationBean.setOrder(1); // 순서 등록
        filterRegitrationBean.addURLPatterns("/*"); // 모든 url에 다 적용
        return filterRegitrationBean;
    }
    @Bean
    public FilterRegistrationBean encodingFilter() {
        FilterRegistrationBean<Filter> filterRegistrationBean = new
                        FilterRegistrationBean<Filter>(new CharacterEncodingFilter());
        filterRegistrationBean.setOrder(2);
        filterRegistrationBean.addUrlPatterns("/filter/*");
        return filterRegistrationBean;
    }
…
```

@WebFilter

- **@WebFilter + @ServletComponentScan 사용**
  - urlPatterns 필터를 적용할 URL 패턴 목록을 지정
  - servletNames 필터를 적용할 서블릿 이름 목록을 지정
  - filterName 필터의 이름을 지정
  - initParams 초기화 파라미터 목록을 지정
  - dispatcherType 필터를 적용할 범위를 지정. 열거 타입인 Dispatcher에 정의된 값을 사용. 기본값은 DispatchType.REQUEST

```
@ServletComponentScan // @WebFilter + @ServletComponentScan
@SpringBootApplication
public class SpringBootFilterApplication {
…
@WebFilter(urlPatterns = "/*") // 모든 요청에 LogFilter 적용
public class LogFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("init LogFilter");
    }
…
```

@WebFilter

- **@WebFilter + @ServletComponentScan 사용**
  - **initParams** 요소는 필터의 초기화 매개변수와 값 목록을 지정하려면 아래와 같이 작성

```
@WebFilter(urlPatterns = "/admin/*", initParams = @WebInitParam(name =
"adminUserID", value = "admin"))
public class AdminCheckFilter implements Filter {
    private String adminUserID;
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        adminUserID = filterConfig.getInitParameter("adminUserID");
        System.out.println("init AdminCheckFilter");
    }
.....
```

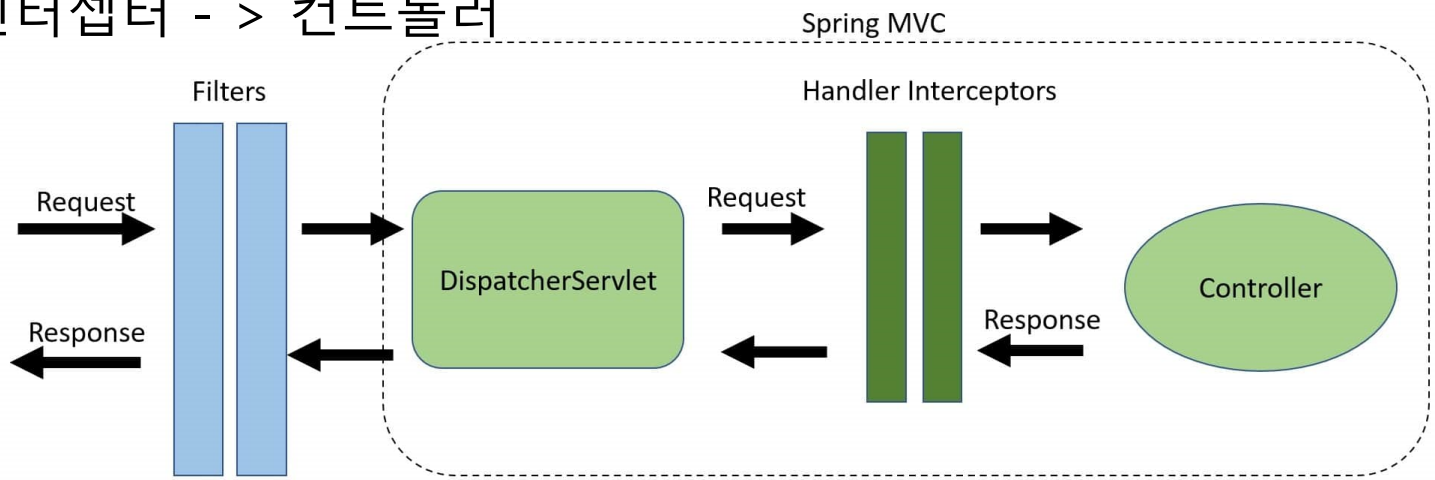
@WebFilter

- **@WebFilter + @ServletComponentScan 사용**
  - **initParams** 요소는 필터의 초기화 매개변수와 값 목록을 지정하려면 아래와 같이 작성

```
@WebFilter(urlPatterns = "/*", initParams = { @WebInitParam(name = "encoding", value = "UTF-8"), @WebInitParam(name = "encoding2", value = "EUC-KR") })
public class CharacterEncodingFilter implements Filter {
    private String encoding;
    private String encoding2;
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        encoding = filterConfig.getInitParameter("encoding");
        encoding2 = filterConfig.getInitParameter("encoding2");
        System.out.println("init CharacterEncodingFilter encoding=" + encoding);
    }
....
```

인터셉터

- 인터셉터(Interceptor)
  - 스프링 인터셉터도 서블릿 필터와 같이 웹과 관련된 공통 관심 사항을 효과적으로 해결 할 수 있는 기술
  - 서블릿 필터가 서블릿이 제공하는 기술이라면, 스프링 인터셉터는 Spring MVC가 제공하는 기술
  - 스프링 인터셉터는 DispatcherServlet이 컨트롤러를 호출하기 전과 후에 요청과 응답을 처리하는 기능을 제공
  - HTTP 요청 -> WAS -> 필터 -> 디스패쳐 서블릿 -> 스프링 인터셉터 - > 컨트롤러



https://www.baeldung.com/spring-mvc-handlerinterceptor-vs-filter

HandlerInterceptor 인터페이스

- HandlerInterceptor 인터페이스
  - 인터셉터 기능을 구현하는 데 핵심적인 역할을 하는 인터페이스.
  - 컨트롤러 호출전 preHandle(전처리), 컨트롤러 호출후 postHandle(후처리), afterCompletion(종료후) 메서드를 구현

메서드	설명
preHandle(HttpServletRequest, HttpServletResponse, Object)	지정된 Controller가 호출되기 전에 실행되는 메소드. Object Handler는 현재 실행하려는 메소드 자체를 의미하는데, 이를 활용하면 **현재 실행되는 컨트롤러를 파악**하거니, **추가적인 메소드를 싱행하는 등의 작업이 가능**
postHandle(HttpServletRequest, HttpServletResponse, Object, ModelAndView)	Controller가 호출된 후 view rendering 전에 실행되는 메소드
afterCompletion(…)	인터셉터 종료후 기능을 작성하는 메소드 view rendering 된 후, 즉 모든 작업이 완료된 후에 실행되는 메소드

HandlerInterceptor 인터페이스

```
@Slf4j
public class LogInterceptor implements HandlerInterceptor  {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        String requestURI = request.getRequestURI();
        log.info("preHandle [interceptor] requestURI : {}" , requestURI);
        return true; // true면 핸들러 다음 동작이 실행 false면 이후에 진행하지 않음
    }
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle [interceptor]");
    }
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        log.info("afterCompletion [interceptor]");
    }
}
```

WebConfig implements WebMvcConfigurer

- WebConfig 클래스
  - WebMvcConfigurer를 implements한 @Configuration 클래스를 생성
  - addInterceptors를 오버라이딩하여 Interceptor를 추가해주면 됨

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LogInterceptor()) // logInterceptor 등록
                .order(1) // 순서 등록
                .addPathPatterns("/**"); // 모든 url에 다 적용
        registry.addInterceptor(new PerformanceInterceptor())
                .order(2)
                .addPathPatterns("/**")
                .excludePathPatterns("/", "/admin/**");
    }
…
```

필터와 인터셉터의 차이

구분	필터	인터셉터
사용클래스	Filter 인터페이스	HandlerInterceptor 인터페이스
기술	Servlet에서 제공하는 기술	Spring MVC에서 제공하는 기술
실행영역	Servlet Context에 위치 Spring Context 외부에 위치	Spring Context (WebApplicationContext) 내부에 위치
호출시점	DispatcherServlet에 요청이 전달되기 전, 후의 모든 요청에 대한 부가 작업이 가능	DistpatcherServlet이 컨트롤러를 호출하기 전, 후로 끼어들기 때문에 스프링 컨텍스트 (WebApplicationContext)에 위치한 모든 Bean(자바 객체)에 접근 가능
사용예시	일반적으로 보안 관련 작업, 이미지/데이터 압축 및 문자열 인코딩 변환, 인증 및 인가 등의 요청에 대한 처리로 사용	로그인 체크, 권한 체크, Controller로 넘겨주는 데이터 가공 등의 요청에 대한 처리로 사용