**웹 앱 개발**

**\* 자바스크립트 배열**

배열의 선언은 아래 두 가지 방법으로 할 수 있다.

|  |
| --- |
| var arr= [];  var arr= [1,2,3,4,”hello”,null,[]]; |

배열은 length, push, indexOf, join 등등의 메소드들이 존재한다. 배열은 for 문을 통해 방문할 수 있지만 forEach를 통해 방문할 수도 있다.

|  |
| --- |
| arr.forEach(fuction(value, index, object){  console.log(value);  }); |

배열에는 forEach 외에도 map 이라는 함수가 존재한다. map 함수는 배열에 새로운 값을 가진 배열로 만들어줄 수 있는 함수이다.

|  |
| --- |
| var mapped= arr.map(function(value, index, obj){  return value \* 2; <!-- arr 배열에 곱하기 2를 한 배열을 만든다. -->  }); |

배열에 map과 다르게 filter라는 함수가 존재한다. filter의 입력 파라미터 함수는 리턴 값이 bollean 타입으로 함수의 이름에서 알 수 있듯이 만들어지는 배열에 추가될 인자를 결정한다.

|  |
| --- |
| var filtered= arr.filter(function(value, index, obj) {  return value % 2 === 0 ? true : false;  }); |

**\* 자바스크립트 객체**

객체는 key, value를 가지고 있는 객체이다. 배열은 [] 표기법을 사용하고 객체는 {} 표기법을 사용한다. 아래와 같이 선언하고 불러올 수 있다.

|  |
| --- |
| var obj= {name : “crong”, age : 20};  console.log( obj.name );  console.log( obj[“name”] ); |

객체를 탐색할 때는 for-in 문을 많이 사용한다. 아래와 같이 사용할 수 있다.

|  |
| --- |
| for(key in obj) {  console.log(“key [“ + key + “], value [“ + obj[key] + “]”);  } |

객체의 key 값을 아래와 같이 배열 형태로 반환받을 수 있다. 이렇게 반환 받은 후 아래 예제처럼 forEach를 활용하여 탐색할 수 있다.

|  |
| --- |
| var keys= Object.keys(obj);  keys.forEach(function(value, index, obj){  console.log(obj[value]);  }); |

**\* DOM Node 조작하기**

DOM, 즉 HTML은 동적으로 다양하게 변경될 수 있다. DOM APIs에서는 다양한 메서드와 속성을 지원하고 있으니 이를 활용하는 방법을 배울 필요가 있다. DOM API는 라이브러리(React, Angular JS 등등)를 쓰는 것보다 대체로 더 빠르다. 물론 라이브러리보다 그 사용법이 불편하고 어려울 수는 있지만 low-level의 DOM API를 이해한다면, 라이브러리나 프레임워크를 사용하면서도 그 동작을 이해하고 유추하기가 쉽고 어떤 문제를 해결할때도 더 유리하다.

**\* Ajax 응답 처리와 비동기**

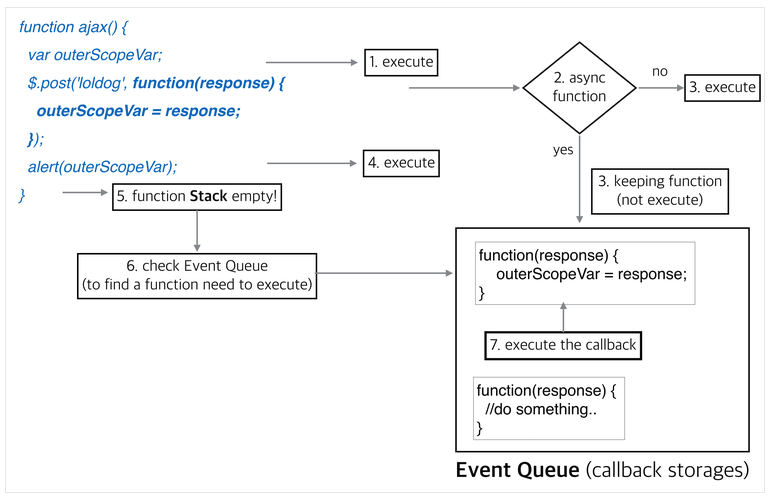
화면 전환 없이 서버에 데이터를 요청하여 서버측 데이터를 화면에 그려줄 수 있도록 해주는 기술이다. 이를 처리하기 위해 Ajax와 같은 요청이 있다면 Ajax 요청을 서버에 보내고 응답 전까지 다른 작업을 수행하고 응답이 오면 해당 데이터를 그리는 작업을 수행한다(비동기)

즉, 아래 코드를 예로 들면 addEventListener 함수의 콜백함수가 Event 큐에 들어가게 되고, 실제 서버에 요청을 보낸 후 ajax 함수는 수행을 끝내게 된다. 이 후 서버에 응답이 오면 콜백함수의 로직이 처리되게 된다.

|  |
| --- |
| function ajax() {  var req= new XMLHttpRequest();  req.addEventListener(“load”, function(){  var jsonobj= JSON.parse(this.responseText);  }  req.open(“GET”, “http://localhost:8080/study”);  req.send();  } |

위의 예제를 보면 Ajax를 통해 서버에 보낸 요청은 JSON 타입의 문자열 형태로 응답이 온다. 서버로부터의 응답이 load 이벤트와 함께 응답이 오면 콜백함수에서 this.responseText를 통해 응답 문자열을 가져와 JSON 함수를 통해 JSON 객체로 파싱할 수 있다.

실제 자바스크립트에서 콜백함수를 비동기로 수행하는 전체적인 로직은 아래와 같다.



지금 배운 Ajax 통신은 보안 문제로 인해 서로 다른 도메인 간 통신에 사용하기는 어려운 부분이 많다(예를 들어, 네이버 특정 앱에서 네이버의 검색 결과 데이터를 요청) 이를 해결하기 위해 요청 헤더에 특정 데이터가 있는 경우에만 응답을 해준다거나 하는 CORS 라는 표준이 생기게 된다. 또한 보안 문제를 해결하기 위핸 JSONP 기술을 제공하기도 한다.

**\* 웹 애니메이션 이해와 setTimeout 활용**

애니메이션이란 반복적인 움직임의 처리를 말한다. CSS3의 transition, transform 속성을 사용해서 간단한 애니메이션 처리를 할 수 있다. 자바스크립트를 통해 엘리먼트를 우측으로 1px 씩 움직인다와 같은 애니메이션 처리를 할 수 있는데 CSS3의 속성을 사용함이 더 성능이 좋다.

애니메이션 처리를 하기 위해선 하나의 화면(프레임)을 약 16ms 동안 유지해야 사용자가 자연스러움을 느낄 수 있다. 이러한 시간 처리를 위해서 자바스크립트에선 setInterval, setTimeout과 같은 함수들을 사용한다.

주의할 점은 setInterval 함수를 통한 로직이 누락 없이 수행됨이 보장될 수 없다는 점과 설정한 시간에 맞춰 정확히 수행된다는 보장이 없다는 것이다. 그 이유는 두 가지가 있다. 첫 번째로setInterval 함수는 비동기이기 때문에 다른 동기 함수보다 우선순위가 밀려 정해진 시간에 수행이 될 수 없기 때문이다. 두 번째로 하나의 setInterval 함수가 실행되기 위해 큐에서 대기 중일 때엔 정해진 시간이 지나 다시 한번 해당 콜백함수를 큐에 쌓으려고 시도할 때, 이미 쌓여있는 작업이 있다면 해당 작업을 무시해버리기 때문이다. 이 문제를 해결하기 위한 방법으로 setTimeout을 reculsive하게 구현하는 방법이 있다.

|  |
| --- |
| setTimeout(() => {  console.log('현재시각은', new Date());  }, 500); |
| let count = 0;  function animate() {  setTimeout(() => {  if(count >= 20) return;  console.log('현재시각은', new Date());  count++;  animate();  },500);  } |

위의 방법으로 구현하면 setInterval과 다르게 콜백 함수가 이벤트 큐에 누적되지 않고 순차적으로 실행될 수 있기 때문에 누락되는 로직이 사라진다.(약간의 시간 지연이 있지만)

**\* requestAnimationFrame 활용**

사실, setInterval, setTimeout은 애니메이션 처리를 위해 만들어진 함수가 아니다. 브라우저에서 지원하는 requestAnimationframe을 사용하면 delay 없이 부드러운 애니메이션 처리를 할 수 있다. 아래 예시를 보면 이 방법도 reculsive 하게 처리됨을 확인할 수 있다.

|  |
| --- |
| function run() {  if(count-- === 0) return;  console.log(‘현재시각은’, new Date());  window.requestAnimationFrame(run);  }  window.requestAnimationFrame(run); |

**\* CSS3 transition 활용**

이번에 배울 transition 속성은 엘리먼트의 속성을 변경할 때 transition 속성에 준 시간 값 동안 변경을 천천히 하라는 의미이다(애니메이션 효과와 함께)

**\* DOMContentLoaded 이벤트**

브라우저가 HTML 페이지를 가져온 후 그외 CSS, 자바스크립트 파일을 가져와 레이아웃, 이미지 랜더링을 하는데 그 사이에 자바스크립트가 DOM API를 통해 작업을 하게 된다면 에러가 발생할 수 있다. 때문에 브라우저의 랜더링 작업이 끝났음을 이벤트로 알 수 있어야 한다. 이러한 이벤트가 DOMContentLoaded와 load 이벤트이다.

DOMContentLoaded는 DOM 트리 분석이 끝나면 발생하는 이벤트를 의미하고 Load 이벤트는 그 외 모든 페이지 자원들이 다 받아져서 브라우저에 렌더링까지 다 끝난 시점에 발생하는 이벤트이다.

**\* Event delegation**

이벤트를 좀 더 효율적으로 등록하는데 사용한다. 예를 들어 리스트 성의 복수개의 엘리먼트에 대하여 이벤트를 등록해야 하는 상황에서 어떻게 이벤트를 등록해야 할까??

일반적으로 생각할 수 있는 방법으로 반복문을 통해 이벤트를 등록할 수가 있다. 하지만 그만큼 브라우저가 기억해야 하는 list가 많아지면서 비효율적인 메모리 사용이 있을 수 있다. 이 문제 해결을 위해 이벤트 버블링 특성을 사용한다. 이벤트 버블링이란 클릭한 엘리먼트가 하위 엘리먼트라 할지라도 그것을 감싸고 있는 상위 엘리먼트까지 올라가면서 이벤트리스너가 있는지 찾는 것을 의미한다.

|  |
| --- |
| var log = document.querySelector(".log");  var lists = document.querySelectorAll("ul > li");  for(var i=0, len=lists.length; i < len; i++) {  lists[i].addEventListener("click", function(evt) {  log.innerHTML = "clicked" + evt.currentTarget.firstChild.src;  });  } |
| var ul= document.querySelector(“ul”);  ul.addEventListener(“click”, function(evt){  <!— 이벤트 버블링 :  evt.target.tagName은 실제 클릭한 ul의 자식 엘리먼트가  evt.currentTarget엔 ul 엘리먼트가 저장되어 있다. -->  console.log(evt.target.tagName, evt.currentTarget.tagName);  }); |

**\* HTML templating**

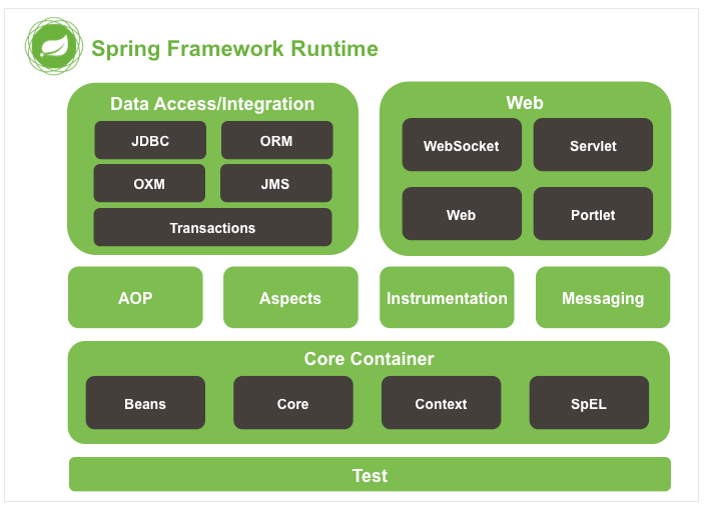
반복적인 HTML 부분을 template으로 만들어두고 서버에서 온 데이터를 결합해서 화면에 추가하는 것을 HTML templating이라고 한다. 아래와 같은 방식으로 templating은 일반적으로 문자열 조작을 통해서 수행하게 된다.

|  |
| --- |
| var data= { title : “hello”, “content : “blah blah” ...}  var html= “<li><h4>{title}</h4><p>{content}...  var resultHtml= html.replace(“{title}”, data.title)  .replace(...)...; |

**\* Spring Framework란**

엔터프라이즈 급(굉장히 큰) 어플리케이션을 구축할 수 있는 솔루션이다. 원하는 부분만 가져다 사용할 수 있도록 모듈화가 잘 되어있고, IoC 컨테이너, 선언적 트랜잭션 관리, AOP와 같은 장점을 가지고 있다.

스프링은 모듈화가 잘 되어있기 때문에 Spring Core를 학습하고 나면, 나머지 Web, Data Acess와 같은 모듈들은 필요하다면 추가해서 사용할 수 있다. 스프링이 지원하는 모듈은 아래와 같다.



**\* Spring IoC/DI 컨테이너**

컨테이너란 인스턴스의 생명주기를 관리하는 관리자를 의미한다. 예를 들어 서블릿 클래스를 작성하면 이 서블릿을 통해 인스턴스를 만들고 관리하는 tomcat을 서블릿 컨테이너라고 한다.

스프링 IoC란 개발자가 만든 인터페이스의 구현체를 생성, 관리하는 것을 의미하고 DI란 이렇게 생성된 구현체를 개발자가 사용할 인터페이스 타입의 필드에 주입 시켜주는 것을 의미한다. 즉 아래와 같은 코드 작성이 가능해진다.

|  |
| --- |
| class CarA implements Car { }  class CarB implements Car { }  class User {  @Autowired  privte Car myCar; // new 연산자를 통해 구현체를 직접 정의 안한다.  } |

**\* JavaConfig를 통한 스프링 의존성 주입**

스프링에서 사용하는 Config 객체가 되기 위해선 클래스에 @Configuration 어노테이션을 붙인다. 마찬가지로 @Bean 어노테이션을 메소드에 붙이면 이 메소드가 리턴하는 객체가 스프링 컨테이너가 관리하는 Bean으로 등록된다. car 메소드를 보면 파라미터로 또다른 Bean 객체를 필요로 하고 있다. 스프링의 ApplicationContext는 먼저 파라미터가 없는 Bean을 먼저 생성한 후에 파라미터를 필요로 하는 Bean을 생성한다.

|  |
| --- |
| @Configuration  public class ApplicationConfig {  @Bean  public Car car(**Engine engine**) {  Car car= new Car();  car.setEngine(engine);  return car;  }  @Bean  public Engine engine() {  return new Engine();  }  } |

또 다른 방법으로 @ComponentScan 어노테이션을 활용하는 방법이 있다. 이 ComponentScan은 클래스패스 내에 제한된 패키지 내의 @Bean 어노테이션과 같이 스프링 컨테이너가 관리하는 자원이 있는지 스캔한 후 있다면 Bean으로 등록하게 된다.

|  |
| --- |
| @Configuration  @ComponentScan(“com.ikth.apps...”)  class ApplicationConfig {  }  @Component  class Car {  @Autowired  private Engine engine;  }  @Component  class Engine { } |

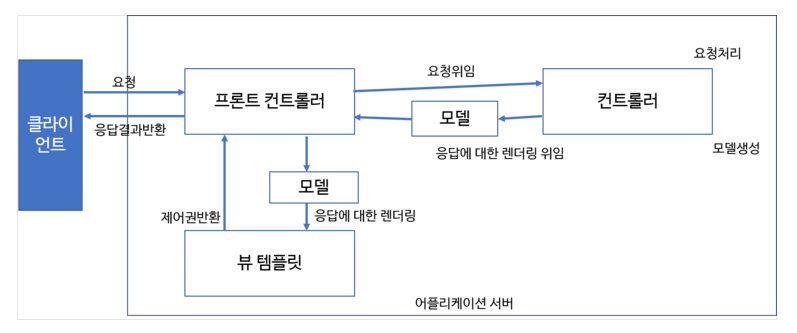
**\* Spring JDBC**

검색을 통해 해결하자. 나는 mybatis 사용할 예정..

**\* Spring MVC**

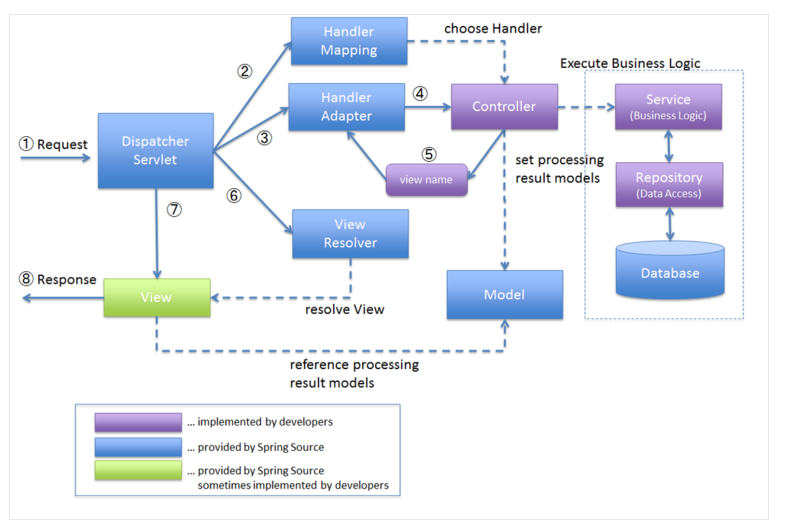
|  |  |
| --- | --- |
| **Model** | 뷰가 렌더링 하는데 필요한 데이터 |
| **View** | 뷰는 실제로 사용자에게 보이는 부분으로 JSP, XML 등으로 결과 표현 |
| **Controller** | 사용자의 액션에 응답하는 컴포넌트 |

MVC엔 다양한 타입의 모델이 존재하는데 스프링에선 아래와 같은 모델 아키텍처를 기본으로 하여 기능을 제공해준다.

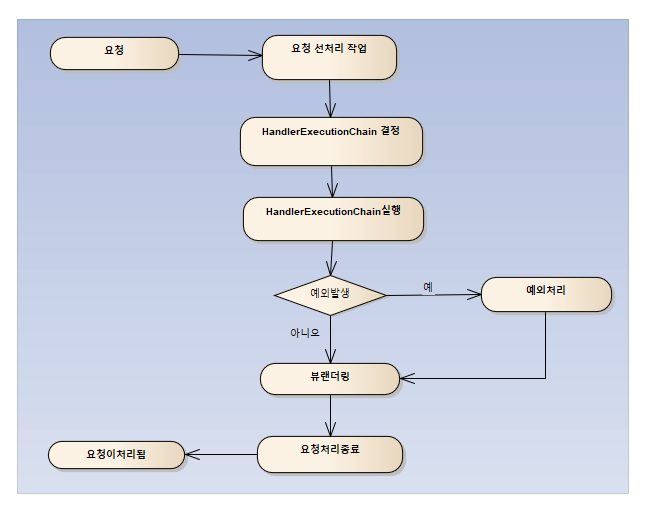


**\* Spring MVC 구성요소**

Spring MVC의 기본 동작 흐름은 아래와 같다.



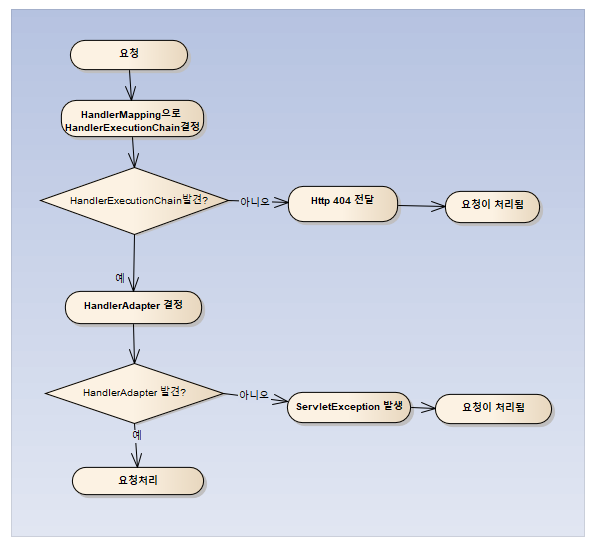
Dispatcher Servlet은 일반적으로 프론트 컨트롤러라고 부른다. 클라이언트의 모든 요청을 받은 후 처리할 핸들러에게 요청을 넘기고, 처리 결과를 받으면 사용자에게 응답 결과를 보여주는 역할을 한다. Dispatcher Servlet의 내부 동작 흐름은 아래와 같다.



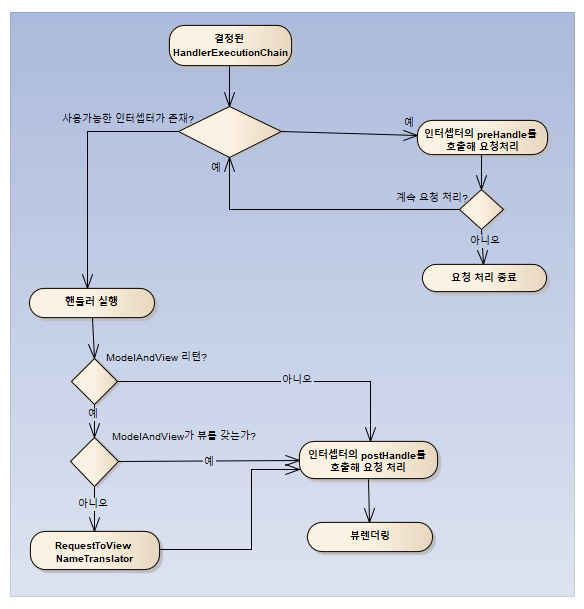
요청 선처리 작업은 아래와 같은 작업을 일반적으로 처리한다.

|  |  |
| --- | --- |
| Locale 결정 | 브라우저의 헤더 정보를 통해 Locale을 가져오고 이 정보를 이용해서 다국어 처리를 할 수 있다. |
| HandlerExecutionChain | HttpServletRequest, Response 객체의 정보를 저장한다. |
| FlashMap 복원 | redirect 시 넘기고 싶은 데이터를 넘길 수 있도록 지원 해준다. |
| Multipart form data | 파일 업로드/다운로드와 같은 서버, 클라이언트의 대용량 데이터를 주고받을 수 있도록 지원 |

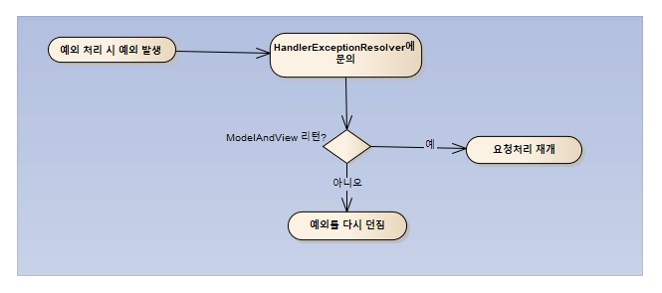
요청을 전달하는 HandlerExecutionChain 결정 및 실행의 상세 흐름은 아래와 같다.



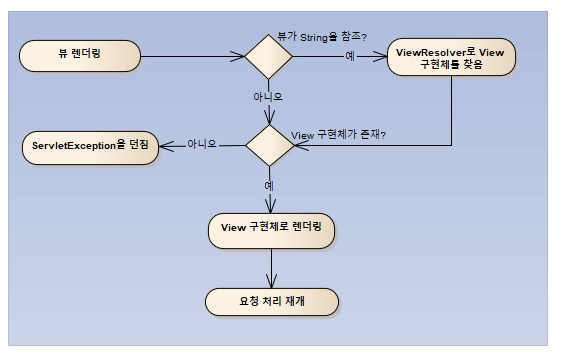
핸들러가 결정되어 실제 실행이 되는 상세 흐름은 아래와 같다.



요청 처리 과정 중 예외 Exception이 발생되면 아래와 같은 흐름으로 처리하게 된다.



응답 모델을 받은 Dispatcher Servlet은 아래와 같은 과정을 통해 실제 뷰를 랜더링 한 후 클라이언트에게 응답하게 된다.



**\* Spring MVC 환경 설정하기**

처음 WAS 또는 서블릿 컨테이너가 기동될 때 DispatcherServlet을 web.xml 을 통해 서블릿을 설정 하거나 Spring의 WebApplicationInitializer를 구현 하는 방법이 있다. 두 방법 중 한 가지로 서블릿을 설정한 후 WebMvcConfigurerAdapter 클래스를 상속 및 @Configuration 어노테이션을 추가하여 설정 클래스를 만든다. 그 후 @EnableWebMvc 어노테이션을 추가하여 스프링에서 기본적으로 제공하는 MVC 패턴을 사용하기 위한 기본 Bean 들을 사용하도록 한다.

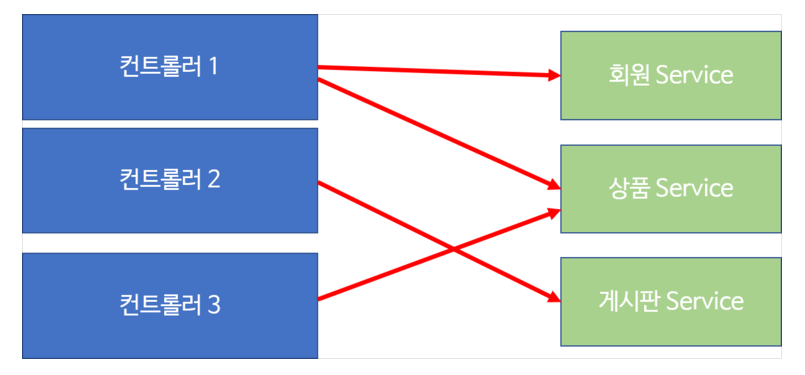
|  |
| --- |
| @Configuration  @EnableWebMvc  @ComponentScan(basePackages = {...})  class WebMvcConfiguration extends WebMvcConfigurerAdapter {  } |

컨트롤러와 URI 매핑을 위해서 일반적으로 어노테이션을 사용한다. 또한 컨트롤러에 추가되는 메소드의 입력 파라미터에도 어노테이션이 사용될 수 있다(입력 파라미터 데이터를 URI path에서 가져올지, Http 요청 파라미터에서 가져올지 등등..)

|  |
| --- |
| @Controller  public class PlusController {  @GetMapping(path = "/plusform")  public String plusform() {  return "plusForm";  }  **@PostMapping(path = "/plus")**  public String plus(**@RequestParam(name = "value1", required = true)** int value1,  @RequestParam(name = "value2", required = true) int value2, ModelMap modelMap) {  int result = value1 + value2;  modelMap.addAttribute("value1", value1);  modelMap.addAttribute("value2", value2);  modelMap.addAttribute("result", result);  return "plusResult";  }  } |

**\* 레이어드 아키텍처란?**

URL은 다르지만 중복으로 필요한 정보가 있다면?? 예를 들어 쇼핑몰 게시판 및 상품 목록 보기에서도 회원 정보가 필요할 수 있다. 이렇게 컨트롤러가 중복으로 필요로 하는 모듈을 서비스 라는 객체로 별도로 만들게 된다. 즉 아래와 같은 구조가 되는 것이다.



서비스 객체란 비즈니스 로직을 수행하는 메서드를 가지고 있는 객체를 의미하고 보통 하나의 비즈니스 로직은 하나의 트랜잭션으로 동작한다.

스프링에선 트랜잭션 처리를 하기 위해 @EnableTransactionManagement라는 어노테이션을 설정 클래스에 추가한다. 그러면 스프링은 PlatformTransactionManager의 구현체 객체 중 하나를 찾아 트랜잭션을 관리하게 된다.

마찬가지로 서비스 객체도 여러 종류의 데이터소스에 접근해야 하는 경우가 있는데 이 때는 DAO라는 객체를 사용하게 된다.

위와 같이 프레젠테이션(컨트롤러), 서비스, Repository(DAO) 로 구조를 나누는 것을 레이어드 아키텍처라고 하고, 보통 객체 뿐 아니라 설정 파일도 동일하게 분리한다.

**\* 스프링 설정 분리 시 주의할 점**

특정 경우 DispatcherServlet을 복수 개 선언할 수 있다. 단 이 땐 각각의 ApplicationContext가 독립적이기 때문에 서로 다른 설정 파일의 Bean을 호출할 수는 없다. 각 서블릿이 공통으로 사용하도록 하려면 ContextLoaderListener를 통해 설정해주면 된다. ContextLoaderListener가 사용하는 ApplicatoionContext는 root가 되고 DispatcherServlet의 ApplicationContext는 자식이 되어 자식 Context는 부모 Context의 Bean을 사용할 수 있는 구조이다. ContextLoaderListener를 통한 설정 등록은 아래와 같이 web.xml에 Listener를 등록하면 된다.

|  |
| --- |
| <web-app>  <display-name>Spring JavaConfig Sample</display-name>  <context-param>  <param-name>contextClass</param-name>  <param-value>  org.springframework.web.context.support.AnnotationConfigWebApplicationContext  </param-value>  </context-param>  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>  kr.or.connect.guestbook.config.ApplicationConfig  </param-value>  </context-param>  <listener>  <listener-class>  org.springframework.web.context.ContextLoaderListener  </listener-class>  </listener>  <servlet>  <servlet-name>mvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet  </servlet-class>  <init-param>  <param-name>contextClass</param-name>  <param-value>  org.springframework.web.context.support.AnnotationConfigWebApplicationContext  </param-value>  </init-param>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>  kr.or.connect.guestbook.config.WebMvcContextConfiguration  </param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>mvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>  org.springframework.web.filter.CharacterEncodingFilter  </filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  </web-app> |

**\* RestController란?**

RestAPI 또는 WebAPI를 개발하기 위해 스프링4부터 @RestController라는 어노테이션을 추가하여 API를 위한 컨트롤러를 등록할 수 있다. 요청 또는 응답을 JSON과 객체 사이 메시지 변환을 위해 다양한 Message Convertor를 제공한다.