Annotation Summary

Innova Lee(이상훈)
gcccompil3r@gmail.com

Annotation Lists

@SpringBootApplication
@Bean
@Configuration
@Autowired
@Value
@Component
@Controller
@RestController
@Service
@RequestMapping
@GetMapping
@PostMapping
@PathVariable
@RequestBody
@JsonCreator
@JsonProperty
@RequestParam
@ModelAttribute
@PostConstruct
@Scheduled
@EnableScheduling
@Repository

SpringBootApplication

해당 Annotation 은 아래 3 개의 Anntoation 을 포함한다.

@SpringBootConfiguration
@ComponentScan
@EnableAutoConfiguration

여기서 ComponentScan 과 EnableAutoConfiguration 이 중요하다.

ComponentScan

패키지내에 application 컴포넌트가 어디에 위치한지 검사한다.
즉 Bean 이라는 녀석을 검색한다.

Bean

Bean 이란 일반적으로 XML 파일에 정의한다.

주요 속성으로는 아래와 같은 것들이 있다.

class: 정규화된 자바 클래스 이름
id: Bean 의 식별자
scope: 객체의 범위(singleton, prototype)
constructor-arg: 생성시 생성자에 전달할 인수
property: 생성시 Bean Setter 에 전달할 인수
init method 와 destroy method

Configuration

Bean 에 대해 Context 에 추가하거나 특정 Class 를 참조할 수 있다.

hello.c | DemoApplication.java | HelloSpringConfiguration.java | Outp

```java
package com.example.demo.decoupled;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HelloSpringConfiguration {

    @Bean
    public MessageProvider provider() {
        return new HelloSpringMessageProvider();
    }

    @Bean
    public MessageRenderer renderer(){
        MessageRenderer renderer = new OutputMessageRenderer();
        renderer.setMessageProvider(provider());
        return renderer;
    }
}
```

hello.c | DemoApplication.java | HelloSpringConfiguration.java | HelloSpring.java | Output

```java
package com.example.demo;

import ...

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(
                HelloSpringConfiguration.class
        );

        MessageRenderer mr = ctx.getBean( name: "renderer", MessageRenderer.class);
        mr.render();

        SpringApplication.run(DemoApplication.class, args);
    }
}
```

EnableAutoConfiguration

Spring Boot 의 자동화 기능(Spring 설정)을 활성화시킨다.

10 년전엔 content.xml 에 component-scan 이라는 속성과
scan 할 패키지명을 적어야 했다.

그러나 지금은 @SpringBootApplication 으로 xml 을 사용할 필요가 없다.
즉 Spring 활용에 있어 모든 XML 을 없애버릴 수 있음을 의미한다.

Autowired

기존에 XML 코드와 @Configuration 과 @Bean 을 사용해야 했다면 @Autowired 의 출현으로 단순히 Autowired Annotation 을 적는것만으로 알아서 필요한 Bean 들을 설정해서 배치한다.

사용전

```
@Service
public class BookService {

    private BookRepository bookRepository;

    public BookService(BookRepository bookRepository){
        this.bookRepository = bookRepository;
    }

}
```

사용전

```
<bean id="bookRepository" class="com.keesun.spring.BookRepository"/>

<bean id="bookService" class="com.keesun.spring.BookService">
    <constructor-arg name="bookRepository" ref="bookRepository"/>
</bean>
```

사용전

```
@Configuration
public class ApplicationConfig {
    @Bean
    public BookRepository bookRepository(){
        return new BookRepository();
    }

    @Bean
    public BookService bookService(){
        return new BookService(bookRepository());
    }
}
```

사용후

```
@Service
public class BookService {
    private BookRepository bookRepository;

    @Autowired
    public BookService(BookRepository bookRepository){
        this.bookRepository = bookRepository;
    }
}
```

사용후

```
@Repository
public class BookRepository { ... }
```

Repository

앞서서 Autowired 에서도 확인했듯이
BookService 에서 BookRepository 를 참조해야 하는데
BookRepository 클래스를 자동으로 Bean 으로 등록할 때 사용한다.

```
@Service
public class BookService {
    private BookRepository bookRepository;

    @Autowired
    public BookService(BookRepository bookRepository){
        this.bookRepository = bookRepository;
    }
}
```

```
@Repository
public class BookRepository { ... }
```

Value

보통 properties 파일에서 값을 가져오는데 많이 활용한다.

```
@Service
public class FCMInitializer {
    @Value("${app.firebase-configuration-file}")
    private String firebaseConfigPath;

    Logger logger = LoggerFactory.getLogger(FCMInitializer.class);

    @PostConstruct
    public void initialize() {
        try {
            FirebaseOptions options = new FirebaseOptions.Builder()
                    .setCredentials(GoogleCredentials.fromStream(
                            new ClassPathResource(firebaseConfigPath).getInputStream())).build();
            if (FirebaseApp.getApps().isEmpty()) {
                FirebaseApp.initializeApp(options);
                logger.info("Firebase application has been initialized");
            }
        } catch (IOException e) {
            logger.error(e.getMessage());
        }
    }
}
```

Component

Bean 의 경우 개발자가 제어할 수 없는 외부 라이브러리들을 Bean 으로 사용한다.
반면 Component 는 직접 만든 Class 를 활용할 경우엔 Component 를 사용한다.

Controller

다음에 나올 RestController 와 마찬가지로
Client 의 요청으로부터 View 를 반환하는 역할을 수행한다.
차이점이라면 Rest 는 @RequestBody 없이도 JSON 타입으로 처리를 한다는 것이다.
JSON 타입이 필요한 경우가 데이터를 필요로 하는 경우에 해당한다.

Book.java | BookController.java | Singer.java | OutputMessageRenderer.java | MessageRenderer

```java
package com.example.demo.library;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class BookController {

    private final BookService bookService;

    public BookController(BookService bookService) { this.bookService = bookService; }

    @GetMapping("/books.html")
    public String all(Model model) {
        model.addAttribute( attributeName: "books", bookService.findAll());
        return "books/list";
    }

    @GetMapping(value = "/books.html", params = "isbn")
    public String get(@RequestParam("isbn") String isbn, Model model) {

        bookService.find(isbn)
                .ifPresent(book -> model.addAttribute( attributeName: "book", book));

        return "books/details";
    }

    @PostMapping("/books")
    public Book create(@ModelAttribute Book book) { return bookService.create(book); }
}
```

RestController

매서드를 자동으로 RequestBody 없이도 JSON 타입으로 전송하게 해준다.

application.properties | DemoApplication.java | PushNotificationController.java | FCMInitializer.java

```java
package com.example.demo.fcm.controller;

import com.example.demo.fcm.model.PushNotificationRequest;
import com.example.demo.fcm.model.PushNotificationResponse;
import com.example.demo.fcm.service.PushNotificationService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PushNotificationController {
    private PushNotificationService pushNotificationService;

    public PushNotificationController(PushNotificationService pushNotificationService) {
        this.pushNotificationService = pushNotificationService;
    }

    @PostMapping("/notification/topic")
    public ResponseEntity sendNotification(@RequestBody PushNotificationRequest request) {
        pushNotificationService.sendPushNotificationWithoutData(request);
        return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
                message: "Notification has been sent."), HttpStatus.OK);
    }
```

Service

Controller 로 넘어온 요청을 처리하기 위해 Service 가 호출되며
Service 는 적절하게 정보를 가공하여 Controller 에게 데이터를 전달해준다.

application.properties | DemoApplication.java | PushNotificationController.java | FCMInitializer.java

```java
package com.example.demo.fcm.controller;

import com.example.demo.fcm.model.PushNotificationRequest;
import com.example.demo.fcm.model.PushNotificationResponse;
import com.example.demo.fcm.service.PushNotificationService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PushNotificationController {
    private PushNotificationService pushNotificationService;

    public PushNotificationController(PushNotificationService pushNotificationService) {
        this.pushNotificationService = pushNotificationService;
    }

    @PostMapping("/notification/topic")
    public ResponseEntity sendNotification(@RequestBody PushNotificationRequest request) {
        pushNotificationService.sendPushNotificationWithoutData(request);
        return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
                message: "Notification has been sent."), HttpStatus.OK);
    }
```

application.properties | DemoApplication.java | PushNotificationService.java | fir-pushapp-2375

```java
package com.example.demo.fcm.service;

import com.example.demo.fcm.firebase.FCMService;
import com.example.demo.fcm.model.PushNotificationRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutionException;

@Service
public class PushNotificationService {
    //@Value("${app.notifications.defaults}")
    @Value("#{${app.notifications.defaults}}")
    private Map<String, String> defaults;
    //@Value("${app.notifications.defaults.topic}")

    private Logger logger = LoggerFactory.getLogger(PushNotificationService.class);
    private FCMService fcmService;

    public PushNotificationService(FCMService fcmService) {
        this.fcmService = fcmService;
    }
```

RequestMapping

요청으로 들어온 URL 을 어떤 Method 가 처리할지 결정하는데 사용한다.

GetMapping

아래와 동일한 기능을 수행하는 것으로
GET 형식으로 HTTP Request 를 처리한다.

@RequestMapping(method = RequestMethod.GET)

```
27
28      @PostMapping("/notification/token")
29      public ResponseEntity sendTokenNotification(@RequestBody PushNotificationRequest request) {
30          pushNotificationService.sendPushNotificationToToken(request);
31          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
32                  message: "Notification has been sent."), HttpStatus.OK);
33      }
34
35      @PostMapping("/notification/data")
36      public ResponseEntity sendDataNotification(@RequestBody PushNotificationRequest request) {
37          pushNotificationService.sendPushNotification(request);
38          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
39                  message: "Notification has been sent."), HttpStatus.OK);
40      }
41
42      @GetMapping("/notification")
43      public ResponseEntity sendSampleNotification() {
44          pushNotificationService.sendSamplePushNotification();
45          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
46                  message: "Notification has been sent."), HttpStatus.OK);
47      }
48  }

PushNotificationController 〉 sendNotification()
```

PostMapping

앞서 살펴본 GetMapping 과 유사하다.
POST 형식으로 HTTP Request 를 처리한다.

```
27
28      @PostMapping("/notification/token")
29      public ResponseEntity sendTokenNotification(@RequestBody PushNotificationRequest request) {
30          pushNotificationService.sendPushNotificationToToken(request);
31          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
32                  message: "Notification has been sent."), HttpStatus.OK);
33      }
34
35      @PostMapping("/notification/data")
36      public ResponseEntity sendDataNotification(@RequestBody PushNotificationRequest request) {
37          pushNotificationService.sendPushNotification(request);
38          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
39                  message: "Notification has been sent."), HttpStatus.OK);
40      }
41
42      @GetMapping("/notification")
43      public ResponseEntity sendSampleNotification() {
44          pushNotificationService.sendSamplePushNotification();
45          return new ResponseEntity<>(new PushNotificationResponse(HttpStatus.OK.value(),
46                  message: "Notification has been sent."), HttpStatus.OK);
47      }
48  }
```

PushNotificationController 〉 sendNotification()

PathVariable

URL 경로를 변수화하고자 하는 경우에 사용한다.
책의 ISBN 같은것을 URL 로 걸고자 하는 경우 유용하다.

```
18
19      @GetMapping
20      public Iterable<Book> all() {
21          return bookService.findAll();
22      }
23
24      @GetMapping("/{isbn}")
25      public ResponseEntity<Book> get(@PathVariable("isbn") String isbn) {
26          return bookService.find(isbn)
27                  .map(ResponseEntity::ok)
28                  .orElse(ResponseEntity.notFound().build());
29
30      }
31
32      @PostMapping
33  @   public ResponseEntity<Book> create(@RequestBody Book book,
34                                         UriComponentsBuilder uriBuilder) {
35          Book created = bookService.create(book);
36          URI newBookUri = uriBuilder.path("/books/{isbn}").build(created.getIsbn());
37          return ResponseEntity
38                  .created(newBookUri)
39                  .body(created);
40      }
41  }
```

RequestBody

RequestBody 를 활용하여 HTTP Request Body 를 자바 객체로 전달받을 수 있다.

```
19      @GetMapping
20      public Iterable<Book> all() {
21          return bookService.findAll();
22      }
23
24      @GetMapping("/{isbn}")
25      public ResponseEntity<Book> get(@PathVariable("isbn") String isbn) {
26          return bookService.find(isbn)
27                  .map(ResponseEntity::ok)
28                  .orElse(ResponseEntity.notFound().build());
29
30      }
31
32      @PostMapping
33  @   public ResponseEntity<Book> create(@RequestBody Book book,
34                                         UriComponentsBuilder uriBuilder) {
35          Book created = bookService.create(book);
36          URI newBookUri = uriBuilder.path("/books/{isbn}").build(created.getIsbn());
37          return ResponseEntity
38                  .created(newBookUri)
39                  .body(created);
40      }
41  }
```

JsonCreator

Jackson 의 ObjectMapper 를 이용해 객체를 Serialize/Deserialize 한다.
Asynchornous Application 에서 Message Queue Worker 를 활용하면
Message Model 에 @JsonCreator 를 사용하는 것을 볼 수 있다.
병렬처리 등에서는 동기화가 매우 중요한데 Dserialize 이후
객체가 Immutable(불변성) 하길 원한다면 Setter 가 없어야 하며
@JsonCreator 는 생성자 + Setter 로 대신 생성해주는 역할을 수행한다.
그러므로 불변성이 보장되는 객체를 얻을 수 있다.

```
private static void saveTodo(Todo todo) throws IOException {
    final ObjectMapper mapper = new ObjectMapper();
    final String json = mapper.writeValueAsString(todo);
    final String filename = String.format("%s.json", todo.getId());
    final File file = new File(filename);
    FileUtils.write(file, json);
}

private static Todo readTodoFromFile(String id) throws IOException {
    System.out.println(String.format("id: %s", id));
    String raw = FileUtils.readFileToString(new File(String.format("%s.json", id)));
    final ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(raw, Todo.class);
}
```

JsonProperty

Serialize/Deserialize 를 수행할 때 JSON Key 를 결정하는데 사용한다.

```
@Test
public void test() throws IOException {
    String json = "{₩"name₩":₩"hello world₩"}";

    TestClass result = this.objectMapper.readValue(json, TestClass.class);

    assertThat(result.getName(), is("hello world"));
}

@Data
@AllArgsConstructor
@NoArgsConstructor
public static class TestClass{
    @JsonProperty("name")
    private String name;
}
```

RequestParam

단일 파라미터를 전달받고자 할 경우 사용하는 Annotation 이다.

ModelAttribute

ModelAttribute 는 JSP 파일에 반환되는 Model 객체에 값을 바인딩할 때 사용한다.
매서드에 붙일 경우엔 매서드의 반환값을 바인딩한다.
매개변수에 붙일 경우엔 HTTP Request 에 있는 속성값을 자동으로 바인딩한다.

PostConstruct

객체가 생성된 이후 별도의 초기화 작업을 위해 실행하는 매서드에 붙인다.
또한 초기화 부분이 오직 한 번만 수행됨을 보장하여
여러 차례 초기화가 되는 성능 낭비를 막을 수 있다.

Scheduled

주기적인 작업이 있을 때 사용하는 Annotation 이다.
하드웨어 분야로 치면 Timer Interrupt 라 생각해도 무방하다.
동적으로 사용하기 위해 ScheduledFuture 와 TaskScheduler 를 함께 사용한다.

EnableScheduling

Task Scheduling 을 설정하는 일환으로 main 파트에 EnableScheduling 을 걸어준다.
이를 걸어줌으로써 앞서 살펴본 Scheduled 부분이 적용 가능해진다.