Spring

**Spring Framework Documentation**

<https://docs.spring.io/spring/docs/current/spring-framework-reference/>

STSのプロジェクト

|  |  |
| --- | --- |
| Spring starter project | Spring boot を使ったプロジェクト |
| Spring legacy project | Spring boot を使わないプロジェクト |

Mavenプロジェクト

・アーキタイプはmaven-archetype-quickstartを選択する

・pomにjunitやエンコーディングの設定が最初から記述されている

MavenプロジェクトのJREシステムライブラリを1.8にする

・<properties>タグに追加する方法

・maven-compiler-pluginを追加する方法

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

DIコンテナ

ApplicationContextの事

ApplicationContext インターフェース

AnnotationConfigApplicationContext JavaConfig用

ClassPathXmlApplicationContext Xml用

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

Bean

DIコンテナに登録するコンポーネント

ルックアップ

DIコンテナからBeanを取得すること

ApplicationContext#getBean

UserService userService = context.getBean(UserService.class);

UserService userService = context.getBean("userService", UserService.class);

UserService userService = (UserService) context.getBean("userService");

DIコンテナへのBeanの登録

・<bean>

・@Bean

xml

<beans ...>

...

<bean id="userService" class="com.example.demo.UserServiceImpl">

<beans>

JavaConfig

@Configuration

public class AppConfig {

@Bean(name = "bCryptPasswordEncoder")

PasswordEncoder passwordEncoder() {

new BCryptPasswordEncoder();

}

@Bean

UserService userService() {

new UserServiceImpl();

}

}

コンポーネントスキャン

xml

<beans ...>

...

<context:component-scan base-package="com.example.demo">

<beans>

JavaConfig

@Configuration

@ComponentScan("com.example.demo")

public class AppConfig {

}

スキャン対象の@

@Controller

@Service

@Repository

@Component

@Configuration

@RestController

オートワイヤリング

@Component

public class UserServiceImpl implements UserService {

@Autowired

UserRepository userRepository;

@Autowired(required = false)

@Qualifier("sha256PasswordEncoder")

PasswordEncoder passwordEncoder;

}

Configurationのインポート

<import>

@Import

<beans ...>

...

<import resource="classpath:domain-config.xml">

<import resource="classpath:infra-config.xml">

<beans>

Java Config

@Configuration

@Import({DomainConfig.class, InfrastructureConfig.class})

public class AppConfig {

}

リソース

StaticApplicationContext

とりあえずApplicationContextを用意したい場合

Resource

Resource res = app.getResource(“classpath:sample.txt”);

リソース文字列

|  |  |
| --- | --- |
| classpath: | src/main/resources |

・src/main/javaとsrc/main/resourcesはビルドするとtarget/classesにまとめられる

ClassPathXmlApplicationContext

src/main/resourcesから読み込む場合

AOP

ロギングとかに使える

|  |  |
| --- | --- |
| アドバイス | 処理 |
| ジョインポイント | タイミング |
| ポイントカット | 条件 |

dependency

・spring-aop

・aspectjrt

・aspectjweaber

enable

|  |  |
| --- | --- |
| JavaConfig | @EnableAspectJAutoProxyを使う |
| XML | <aop:aspectj-autoproxy>を使う |

Aspectクラス

・@Aspect

・@Component

・@Before, @After, @Around

@Aspect

@Component

public class LoggingAspect {

@Before(“execution(\* \*..\*ServiceImpl.\*(..))”)

public void startLog(JoinPoint jp) {

System.out.println(“メソッド開始:” + jp.getSignature());

}

}

・アスペクトメソッドの引数はJoinPoint

・@Aroundの場合はProceedingJoinPoint

・ポイントカット式は色々な表現がある

SpringMVC

コントローラー

@Controller

・Webページ用

・JSPやテンプレートエンジンのViewに遷移

・戻り値はView名を指定

String

@RequestMapping("/index")

public String index() {

return "index";

}

ModelAndView

@RequestMapping("/index")

public ModelAndView index() {

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject(new User("紀伊", "太郎"));

modelAndView.setViewName("index");

return modelAndView;

}

フォワード

@RequestMapping("/forward")

public String forward() {

return "forward:/index";

}

リダイレクト

@RequestMapping("/redirect")

public String redirect() {

return "redirect:/index";

}

@ResponseBody

@RequestMapping(value="/orders", method=RequestMethod.GET)

@ResponseBody

public List<Account> accountSummary() {

return accountManager.getAllAccounts();

}

・戻り値がレスポンスのボディを構成する

・Content-Typeは以下次第

①produces属性

　②Acceptヘッダー

・Javaオブジェクトの場合にはJSONかXMLに勝手にシリアライズされる

<https://stackoverflow.com/questions/28646332/how-does-the-spring-responsebody-annotation-work-in-this-restful-application-ex>

ResponseEntity

@RequestMapping("/responseEntity")

public ResponseEntity<String> responseEntity() {

HttpHeaders headers = new HttpHeaders();

headers.add("header1", "heaer1-value");

HttpStatus status = HttpStatus.NOT\_FOUND;

return new ResponseEntity<>("text content", headers, status);

}

HTTPステータスやコンテンツタイプ以外のレスポンスヘッダも指定したい時は、戻り値をResponseEntityにする。

@RestController

@RestController

@RequestMapping("/book")

public class BookRestController {

@GetMapping("/{id}", produces = "application/json")

public Book getBook(@PathVariable int id) {

return findBookById(id);

}

・WebAPI用

・@Contorollerと@ResponseBodyの合成

・戻り値はレスポンスのコンテンツになる

参考リンク

<https://qiita.com/tag1216/items/3680b92cf96eb5a170f0>

@RequestMapping

・クラス、メソッドに着ける

・ショートカット

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

・属性

|  |  |
| --- | --- |
| value | URIを指定 |
| method | HTTPメソッドを指定 |
| consumes | リクエストのContent-Typeを指定 |
| produces | リクエストのAcceptを指定 |

※consumesとproducesにはMediaType定数を使うのが良い

メソッド引数につけるアノテーション

@RequestParam

@PathVariable

@CookieValue

@RequestHeader

@RequestBody

@SessionAttribute

@RequestParam

@GetMapping("/hello")

public String hello(@RequestParam("name") String name) {

URLに含まれるクエリパラメータや、メッセージボディーに含まれるポストパラメータを受け取ることができる。

例）/hello?name=world

@PathVariable

@GetMapping("/hello/{name}")

public String hello(@PathVariable("name") String name) {

URLに含まれる動的なパラメータを受け取ることができる。

@CookieValue

@GetMapping("/hello")

public String hello(@CookieValue("name") String name) {

@RequestHeader

@GetMapping("/hello")

public String hello(@RequestHeader("User-Agent") String userAgent) {

@RequestBody

@PostMapping("/hello")

public String hello(@RequestBody String body) {

@SessionAttribute

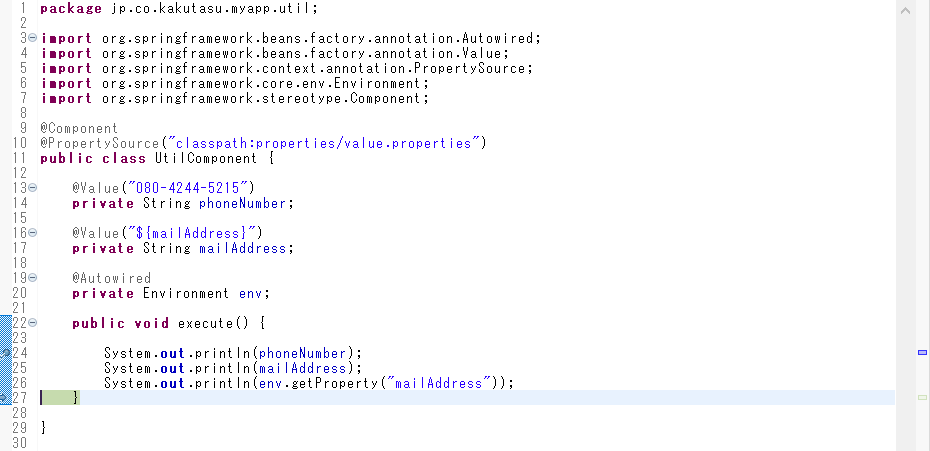
@GetMapping("/hello")

public String hello(@SessionAttribute("age") Integer age) {

参考リンク

<https://qiita.com/MizoguchiKenji/items/2a041f3a3eb13274e55c>

プロパティファイル



|  |  |
| --- | --- |
| @PropertySource | プロパティファイルの読み込み |
| @Value | ・固定値のinject  ・EL式でプロパティファイルの値をinject |
| Environment | プロパティファイルにアクセスできる |

application.properties

・Spring Bootにおいてsrc/main/resource配下に置くと自動で登録されるプロパティファイル

|  |  |
| --- | --- |
| @ContextConfiguration | テストクラスでコンテキストを生成する場合に使うアノテーション |
| locations属性でxmlを指定 |
| classes属性でjava configを指定 |

|  |
| --- |
| @ContextConfiguration("/test-config.xml")  class Test {  } |

|  |
| --- |
| @ContextConfiguration(classes = TestConfig.class)  class Test {  } |

|  |  |
| --- | --- |
| @TestPropertySource | テストクラスでプロパティファイルを登録する場合に使うアノテーション。 |
| プロパティの定義で最も優先順位が高い（@PropertySourceやapplocation.propertiesなど全てに先行する。 |
| locations属性でプロパティファイルのパスを定義 |
| properties属性で"Key"="Value"の形式でプロパティを直接指定することも可能（インラインプロパティ。ファイル指定よりも優先順位が高い） |

|  |
| --- |
| @ContextConfiguration  @TestPropertySource("/test.properties")  class Test {  } |

|  |
| --- |
| @ContextConfiguration  @TestPropertySource(properties = { "timezone = GMT", "port: 4242" })  class Test {  } |

|  |  |
| --- | --- |
| @RunWith(SpringRunner.classs) | junitで@Autwiredなどspringの@が使用可能になる |

|  |  |
| --- | --- |
| @SpringBootTest | SpringBootのテストで使うアノテーション |
| コンテキストを生成する |

|  |  |
| --- | --- |
| @SpringBootApplication | @Configuration |
| @EnableAutoConfiguration |
| @ComponentScan |
| を合わせたアノテーション |