5. 자동 구성(Auto Configuration)

#1.인강/9. 스프링부트/강의#

프로젝트 설정

프로젝트 설정 순서

1. `autoconfig-start`의 폴더 이름을 `autoconfig`로 변경하자.
2. **프로젝트 임포트**

File➜ Open➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
    id 'java'
}
```

```
group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
  - `Lombok`, `Spring Web`, `H2 Database`, `JDBC API`
- 테스트 코드에서 lombok을 사용할 수 있도록 설정을 추가했다.

동작 확인

- 기본 메인 클래스 실행(`AutoConfigApplication.main()`)
- http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

예제 만들기

스프링 부트가 제공하는 자동 구성(Auto Configuration)을 이해하기 위해 간단한 예제를 만들어보자.
`JdbcTemplate`을 사용해서 회원 데이터를 DB에 저장하고 조회하는 간단한 기능이다.

Member

```
package hello.member;

import lombok.Data;

@Data
public class Member {

    private String memberId;
    private String name;

    public Member() {
    }

    public Member(String memberId, String name) {
        this.memberId = memberId;
        this.name = name;
    }

}
```

- `memberId`, `name` 필드가 있는 간단한 회원 객체이다.
- 기본 생성자, `memberId, name`을 포함하는 생성자 이렇게 2개의 생성자를 만들었다.

DbConfig

```
package hello.config;

import com.zaxxer.hikari.HikariDataSource;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.JdbcTransactionManager;
import org.springframework.transaction.TransactionManager;

import javax.sql.DataSource;

@Slf4j
```

```java
@Configuration
public class DbConfig {

    @Bean
    public DataSource dataSource() {
        log.info("dataSource 빈 등록");
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:test");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public TransactionManager transactionManager() {
        log.info("transactionManager 빈 등록");
        return new JdbcTransactionManager(dataSource());
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        log.info("jdbcTemplate 빈 등록");
        return new JdbcTemplate(dataSource());
    }
}
```

- `JdbcTemplate`을 사용해서 회원 데이터를 DB에 보관하고 관리하는 기능이다.
- `DataSource`, `TransactionManager`, `JdbcTemplate`을 스프링 빈으로 직접 등록한다.
- 빈 등록이 실제 호출되는지 확인하기 위해 로그를 남겨두었다.
- DB는 별도의 외부 DB가 아니라 JVM 내부에서 동작하는 메모리 DB를 사용한다.
  - 메모리 모드로 동작 옵션: `jdbc:h2:mem:test`
- `JdbcTransactionManager`는 `DataSourceTransactionManager`와 같은 것으로 생각하면 된다. 여기에 예외 변환 기능이 보강되었다.

MemberRepository

```java
package hello.member;

import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
```

```
import java.util.List;

@Repository
public class MemberRepository {

    private final JdbcTemplate template;

    public MemberRepository(JdbcTemplate jdbcTemplate) {
        template = jdbcTemplate;
    }

    public void initTable() {
        template.execute("create table member(member_id varchar primary key,
name varchar)");
    }

    public void save(Member member) {
        template.update("insert into member(member_id, name) values(?,?)",
                member.getMemberId(),
                member.getName());
    }

    public Member find(String memberId) {
        return template.queryForObject("select member_id, name from member where
member_id=?",
                BeanPropertyRowMapper.newInstance(Member.class),
                memberId);
    }

    public List<Member> findAll() {
        return template.query("select member_id, name from member",
                BeanPropertyRowMapper.newInstance(Member.class));
    }
}
```

- `JdbcTemplate`을 사용해서 회원을 관리하는 리포지토리이다.
- `DbConfig`에서 `JdbcTemplate`을 빈으로 등록했기 때문에 바로 주입받아서 사용할 수 있다.
- `initTable`: 보통 리포지토리에 테이블을 생성하는 스크립트를 두지는 않는다. 여기서는 예제를 단순화 하기 위해 이곳에 사용했다.

MemberRepositoryTest

```
package hello.member;
```

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;

import static org.assertj.core.api.Assertions.*;

@SpringBootTest
class MemberRepositoryTest {

    @Autowired
    MemberRepository memberRepository;

    @Transactional
    @Test
    void memberTest() {
        Member member = new Member("idA", "memberA");
        memberRepository.initTable();
        memberRepository.save(member);
        Member findMember = memberRepository.find(member.getMemberId());
        assertThat(findMember.getMemberId()).isEqualTo(member.getMemberId());
        assertThat(findMember.getName()).isEqualTo(member.getName());
    }
}
```

- `@Transactional`을 사용해서 트랜잭션 기능을 적용했다.
  - 참고로 `@Transactional`을 사용하려면 `TransactionManager`가 스프링 빈으로 등록되어 있어야 한다.
- 테이블을 생성하고, 회원 데이터를 저장한 다음 다시 조회해서, 기존 데이터와 같은지 간단히 검증한다.

테스트가 성공 한다면 정상 동작한 것이다.

정리

회원 데이터를 DB에 보관하고 관리하기 위해 앞서 빈으로 등록한 `JdbcTemplate`, `DataSource`, `TransactionManager`가 모두 사용되었다. 그런데 생각해보면 DB에 데이터를 보관하고 관리하기 위해 이런 객체들을 항상 스프링 빈으로 등록해야 하는 번거로움이 있다. 만약 DB를 사용하는 다른 프로젝트를 진행한다면 이러한 객체들을 또 스프링 빈으로 등록해야 할 것이다.

자동 구성 확인

`JdbcTemplate`, `DataSource`, `TransactionManager`가 스프링 컨테이너에 잘 등록되었는지 간단히 확인해보자.

DbConfigTest

```
package hello.config;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.transaction.TransactionManager;

import javax.sql.DataSource;

import static org.assertj.core.api.Assertions.assertThat;

@Slf4j
@SpringBootTest
class DbConfigTest {

    @Autowired
    DataSource dataSource;
    @Autowired
    TransactionManager transactionManager;
    @Autowired
    JdbcTemplate jdbcTemplate;

    @Test
    void checkBean() {
        log.info("dataSource = {}", dataSource);
        log.info("transactionManager = {}", transactionManager);
        log.info("jdbcTemplate = {}", jdbcTemplate);

        assertThat(dataSource).isNotNull();
        assertThat(transactionManager).isNotNull();
        assertThat(jdbcTemplate).isNotNull();
    }

}
```

- 해당 빈들을 `DbConfig` 설정을 통해 스프링 컨테이너에 등록했기 때문에, `null` 이면 안된다.
  - 사실 `@Autowired` 는 의존관계 주입에 실패하면 오류가 발생하도록 기본 설정되어 있다. 이해를 돕기 위해 이렇게 코드를 작성했다.
- 테스트는 정상이고 모두 의존관계 주입이 정상 처리된 것을 확인할 수 있다.
- 출력 결과를 보면 빈이 정상 등록된 것을 확인할 수 있다.

출력결과

```
hello.config.DbConfig   : dataSource 빈 등록
hello.config.DbConfig   : jdbcTemplate 빈 등록
hello.config.DbConfig   : transactionManager 빈 등록
...
..DbConfigTest: dataSource = HikariDataSource (null)
..DbConfigTest: transactionManager =
org.springframework.jdbc.support.JdbcTransactionManager@5e99e2cb
..DbConfigTest: jdbcTemplate =
org.springframework.jdbc.core.JdbcTemplate@76ac68b0
```

빈 등록 제거

`JdbcTemplate`, `DataSource`, `TransactionManager` 빈은 모두 `DbConfig` 를 통해서 스프링 컨테이너에 빈으로 등록되었다.
이번에는 `DbConfig` 에서 해당 빈들을 등록하지 않고 제거해보자.

`DbConfig` 에서 빈 등록을 제거하는 방법은 2가지 이다.

- `@Configuration` 을 주석처리: 이렇게 하면 해당 설정 파일 자체를 스프링이 읽어들이지 않는다. (컴포넌트 스캔의 대상이 아니다.)
- `@Bean` 주석처리: `@Bean` 이 없으면 스프링 빈으로 등록하지 않는다.

여기서는 간단히 `@Configuration` 을 주석처리해보자.

DbConfig - 수정

```
package hello.config;

//@Configuration //주석처리
public class DbConfig {

    @Bean
    public DataSource dataSource() {
```

```
        HikariDataSource dataSource = new HikariDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setJdbcUrl("jdbc:h2:mem:test");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public TransactionManager transactionManager() {
        return new JdbcTransactionManager(dataSource());
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}
```

- `@Configuration`을 주석처리 했다.

`DbConfigTest.checkBean()` 테스트를 다시 실행해보자.

출력결과

```
...
..DbConfigTest: dataSource = HikariDataSource (null)
..DbConfigTest: transactionManager =
org.springframework.jdbc.support.JdbcTransactionManager@5e99e2cb
..DbConfigTest: jdbcTemplate =
org.springframework.jdbc.core.JdbcTemplate@76ac68b0
```

이번에 실행한 출력 결과를 보면 기존에 있던 빈 등록 로그가 없는 것을 확인할 수 있다.

기존 빈 등록 로그

```
hello.config.DbConfig   : dataSource 빈 등록
hello.config.DbConfig   : jdbcTemplate 빈 등록
hello.config.DbConfig   : transactionManager 빈 등록
```

- 우리가 등록한 `JdbcTemplate`, `DataSource`, `TransactionManager`가 분명히 스프링 빈으로 등록되지 않았다는 것이다.
- 그런데 테스트는 정상 통과하고 심지어 출력결과에 `JdbcTemplate`, `DataSource`, `TransactionManager` 빈들이 존재하는 것을 확인할 수 있다. 어떻게 된 것 일까?
- 사실 이 빈들은 모두 스프링 부트가 자동으로 등록해 준 것이다.

스프링 부트의 자동 구성

스프링 부트는 자동 구성(Auto Configuration)이라는 기능을 제공하는데, 일반적으로 자주 사용하는 수 많은 빈들을 자동으로 등록해주는 기능이다.
앞서 우리가 살펴보았던 `JdbcTemplate`, `DataSource`, `TransactionManager` 모두 스프링 부트가 자동 구성을 제공해서 자동으로 스프링 빈으로 등록된다.
이러한 자동 구성 덕분에 개발자는 반복적이고 복잡한 빈 등록과 설정을 최소화 하고 애플리케이션 개발을 빠르게 시작할 수 있다.

자동 구성 살짝 알아보기

스프링 부트는 `spring-boot-autoconfigure`라는 프로젝트 안에서 수 많은 자동 구성을 제공한다.
`JdbcTemplate`을 설정하고 빈으로 등록해주는 자동 구성을 확인해보자.

JdbcTemplateAutoConfiguration

```
package org.springframework.boot.autoconfigure.jdbc;

import javax.sql.DataSource;

import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnSingleCandidate;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import
org.springframework.boot.sql.init.dependency.DatabaseInitializationDependencyCon
figurer;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

@AutoConfiguration(after = DataSourceAutoConfiguration.class)
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })
@ConditionalOnSingleCandidate(DataSource.class)
```

```
@EnableConfigurationProperties(JdbcProperties.class)
@Import({ DatabaseInitializationDependencyConfigurer.class,
JdbcTemplateConfiguration.class,
        NamedParameterJdbcTemplateConfiguration.class })
public class JdbcTemplateAutoConfiguration {

}
```

- **주의! 여기서 모든 것을 깊이있게 이해하지 않아도 된다. 대략 어떻게 동작하는지 감을 잡을 수 있는 정도면 충분하다. 진행하면서 자동 구성과 관련된 부분들을 단계적으로 풀어나간다.**

- `@AutoConfiguration`: 자동 구성을 사용하려면 이 애노테이션을 등록해야 한다.
  - 자동 구성도 내부에 `@Configuration`이 있어서 빈을 등록하는 자바 설정 파일로 사용할 수 있다.
  - `after = DataSourceAutoConfiguration.class`
    - 자동 구성이 실행되는 순서를 지정할 수 있다. `JdbcTemplate`은 `DataSource`가 필요하기 때문에 `DataSource`를 자동으로 등록해주는 `DataSourceAutoConfiguration` 다음에 실행하도록 설정되어 있다.
- `@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })`
  - IF문과 유사한 기능을 제공한다. 이런 클래스가 있는 경우에만 설정이 동작한다. 만약 없으면 여기 있는 설정들이 모두 무효화 되고, 빈도 등록되지 않는다.
  - `@ConditionalXxx` 시리즈가 있다. 자동 구성의 핵심이므로 뒤에서 자세히 알아본다.
  - `JdbcTemplate`은 `DataSource`, `JdbcTemplate`라는 클래스가 있어야 동작할 수 있다.
- `@Import`: 스프링에서 자바 설정을 추가할 때 사용한다.

`@Import`의 대상이 되는 `JdbcTemplateConfiguration`를 추가로 확인해보자.

JdbcTemplateConfiguration

```
package org.springframework.boot.autoconfigure.jdbc;

import javax.sql.DataSource;

import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.jdbc.core.JdbcOperations;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(JdbcOperations.class)
```

```
class JdbcTemplateConfiguration {

    @Bean
    @Primary
    JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties properties)
{
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcProperties.Template template = properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int)
template.getQueryTimeout().getSeconds());
        }
        return jdbcTemplate;
    }

}
```

- `@Configuration` : 자바 설정 파일로 사용된다.
- `@ConditionalOnMissingBean(JdbcOperations.class)`
  - `JdbcOperations` 빈이 없을 때 동작한다.
  - `JdbcTemplate` 의 부모 인터페이스가 바로 `JdbcOperations` 이다.
  - 쉽게 이야기해서 `JdbcTemplate` 이 빈으로 등록되어 있지 않은 경우에만 동작한다.
  - 만약 이런 기능이 없으면 내가 등록한 `JdbcTemplate` 과 자동 구성이 등록하는 `JdbcTemplate` 이 중복 등록되는 문제가 발생할 수 있다.
  - 보통 개발자가 직접 빈을 등록하면 개발자가 등록한 빈을 사용하고, 자동 구성은 동작하지 않는다.
- `JdbcTemplate` 이 몇가지 설정을 거쳐서 빈으로 등록되는 것을 확인할 수 있다.

자동 등록 설정

다음과 같은 자동 구성 기능들이 다음 빈들을 등록해준다.

- `JdbcTemplateAutoConfiguration` : `JdbcTemplate`
- `DataSourceAutoConfiguration` : `DataSource`
- `DataSourceTransactionManagerAutoConfiguration` : `TransactionManager`

그래서 개발자가 직접 빈을 등록하지 않아도 `JdbcTemplate` , `DataSource` , `TransactionManager` 가 스프링 빈으로 등록된 것이다.

스프링 부트가 제공하는 자동 구성(AutoConfiguration)

- https://docs.spring.io/spring-boot/docs/current/reference/html/auto-configuration-

classes.html

- 스프링 부트는 수 많은 자동 구성을 제공하고 `spring-boot-autoconfigure`에 자동 구성을 모아둔다.
- 스프링 부트 프로젝트를 사용하면 `spring-boot-autoconfigure` 라이브러리는 기본적으로 사용된다.

Auto Configuration - 용어, 자동 설정? 자동 구성?

Auto Configuration은 주로 다음 두 용어로 번역되어 사용된다.

- 자동 설정
- 자동 구성

자동 설정

`Configuration`이라는 단어가 컴퓨터 용어에서는 환경 설정, 설정이라는 뜻으로 자주 사용된다. Auto Configuration은 크게 보면 빈들을 자동으로 등록해서 스프링이 동작하는 환경을 자동으로 설정해주기 때문에 자동 설정이라는 용어도 맞다.

자동 구성

`Configuration`이라는 단어는 구성, 배치라는 뜻도 있다.
예를 들어서 컴퓨터라고 하면 CPU, 메모리등을 배치해야 컴퓨터가 동작한다. 이렇게 배치하는 것을 구성이라 한다.
스프링도 스프링 실행에 필요한 빈들을 적절하게 배치해야 한다. 자동 구성은 스프링 실행에 필요한 빈들을 자동으로 배치해주는 것이다.

자동 설정, 자동 구성 두 용어 모두 맞는 말이다. 자동 설정은 넓게 사용되는 의미이고, 자동 구성은 실행에 필요한 컴포넌트 조각을 자동으로 배치한다는 더 좁은 의미에 가깝다.

- Auto Configuration은 **자동 구성**이라는 단어를 주로 사용하고, 문맥에 따라서 자동 설정이라는 단어도 사용하겠다.
- Configuration이 단독으로 사용될 때는 **설정**이라는 단어를 사용하겠다.

정리

스프링 부트가 제공하는 자동 구성 기능을 이해하려면 다음 두 가지 개념을 이해해야 한다.

- `@Conditional`: 특정 조건에 맞을 때 설정이 동작하도록 한다.
- `@AutoConfiguration`: 자동 구성이 어떻게 동작하는지 내부 원리 이해

자동 구성 직접 만들기 - 기반 예제

자동 구성에 대해서 자세히 알아보기 위해 간단한 예제를 만들어보자.

실시간으로 자바 메모리 사용량을 웹으로 확인하는 예제이다.

Memory

```
package memory;

public class Memory {
    private long used;
    private long max;

    public Memory(long used, long max) {
        this.used = used;
        this.max = max;
    }

    public long getUsed() {
        return used;
    }

    public long getMax() {
        return max;
    }

    @Override
    public String toString() {
        return "Memory{" +
                "used=" + used +
                ", max=" + max +
                '}';
    }
}
```

- `used` : 사용중인 메모리
- `max` : 최대 메모리
- 쉽게 이야기해서 `used` 가 `max` 를 넘게 되면 메모리 부족 오류가 발생한다.

주의!: 여기서는 패키지를 memory로 사용해서 기존과 완전히 다른 곳으로 설정했다. hello 패키지 하위가 아니라는 점에 주의하자, 이후에 다른 곳으로 편하게 때어가기 위해 이렇게 나누었다.

MemoryFinder

```
package memory;

import lombok.extern.slf4j.Slf4j;
import jakarta.annotation.PostConstruct;

@Slf4j
public class MemoryFinder {

    public Memory get() {
        long max = Runtime.getRuntime().maxMemory();
        long total = Runtime.getRuntime().totalMemory();
        long free = Runtime.getRuntime().freeMemory();
        long used = total - free;
        return new Memory(used, max);
    }

    @PostConstruct
    public void init() {
        log.info("init memoryFinder");
    }
}
```

- JVM에서 메모리 정보를 실시간으로 조회하는 기능이다.
- `max`는 JVM이 사용할 수 있는 최대 메모리, 이 수치를 넘어가면 OOM이 발생한다.
- `total`은 JVM이 확보한 전체 메모리(JVM은 처음부터 `max`까지 다 확보하지 않고 필요할 때 마다 조금씩 확보한다.)
- `free`는 `total`중에 사용하지 않은 메모리(JVM이 확보한 전체 메모리 중에 사용하지 않은 것)
- `used`는 JVM이 사용중인 메모리이다. (`used = total - free`)

MemoryController

```
package memory;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
@RequiredArgsConstructor
public class MemoryController {

    private final MemoryFinder memoryFinder;

    @GetMapping("/memory")
    public Memory system() {
        Memory memory = memoryFinder.get();
        log.info("memory={}", memory);
        return memory;
    }
}
```

- 메모리 정보를 조회하는 컨트롤러이다.
- 앞서 만든 `memoryFinder`를 주입 받아 사용한다.

MemoryConfig

```
package hello.config;

import memory.MemoryController;
import memory.MemoryFinder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MemoryConfig {

    @Bean
    public MemoryController memoryController() {
        return new MemoryController(memoryFinder());
    }

    @Bean
    public MemoryFinder memoryFinder() {
        return new MemoryFinder();
    }
}
```

- `memoryController`, `memoryFinder`를 빈으로 등록하자.

- 패키지 위치에 주의하자 `hello.config`에 위치한다.

실행

- http://localhost:8080/memory

결과

```
{"used":24385432,"max":8589934592}
```

간단하게 메모리 사용량을 실시간으로 확인할 수 있다.

패키지 위치

패키지를 이렇게 나눈 이유는, `memory`라는 완전히 별도의 모듈이 있고, `hello` 에서 `memory`의 기능을 불러다 사용한다고 이해하면 된다.

@Conditional

- 앞서 만든 메모리 조회 기능을 항상 사용하는 것이 아니라 특정 조건일 때만 해당 기능이 활성화 되도록 해보자.
- 예를 들어서 개발 서버에서 확인 용도로만 해당 기능을 사용하고, 운영 서버에서는 해당 기능을 사용하지 않는 것이다.
- 여기서 핵심은 소스코드를 고치지 않고 이런 것이 가능해야 한다는 점이다.
  - 프로젝트를 빌드해서 나온 빌드 파일을 개발 서버에도 배포하고, 같은 파일을 운영서버에도 배포해야 한다.
- 같은 소스 코드인데 특정 상황일 때만 특정 빈들을 등록해서 사용하도록 도와주는 기능이 바로 `@Conditional`이다.
- 참고로 이 기능은 스프링 부트 자동 구성에서 자주 사용한다.

지금부터 `@Conditional`에 대해서 자세히 알아보자. 이름 그대로 특정 조건을 만족하는가 하지 않는가를 구별하는 기능이다.

이 기능을 사용하려면 먼저 `Condition` 인터페이스를 구현해야 한다. 그전에 잠깐 `Condition` 인터페이스를 살펴보자.

Condition

```
package org.springframework.context.annotation;

public interface Condition {
    boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);
}
```

- `matches()` 메서드가 `true`를 반환하면 조건에 만족해서 동작하고, `false`를 반환하면 동작하지 않는다.
- `ConditionContext`: 스프링 컨테이너, 환경 정보등을 담고 있다.
- `AnnotatedTypeMetadata`: 애노테이션 메타 정보를 담고 있다.

`Condition` 인터페이스를 구현해서 다음과 같이 자바 시스템 속성이 `memory=on`이라고 되어 있을 때만 메모리 기능이 동작하도록 만들어보자.

```
#VM Options
#java -Dmemory=on -jar project.jar
```

MemoryCondition

```
package memory;

import lombok.extern.slf4j.Slf4j;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

@Slf4j
public class MemoryCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        String memory = context.getEnvironment().getProperty("memory");
        log.info("memory={}", memory);
        return "on".equals(memory);
    }
}
```

- 환경 정보에 `memory=on`이라고 되어 있는 경우에만 `true`를 반환한다.

참고: 환경 정보와 관련된 부분은 뒤에서 아주 자세히 다룬다.

MemoryConfig - 수정

```
package hello.config;
```

```
import memory.MemoryCondition;
import memory.MemoryController;
import memory.MemoryFinder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

@Configuration
@Conditional(MemoryCondition.class) //추가
public class MemoryConfig {

    @Bean
    public MemoryController memoryController() {
        return new MemoryController(memoryFinder());
    }

    @Bean
    public MemoryFinder memoryFinder() {
        return new MemoryFinder();
    }
}
```

- `@Conditional(MemoryCondition.class)`
  - 이제 `MemoryConfig`의 적용 여부는 `@Conditional`에 지정한 `MemoryCondition`의 조건에 따라 달라진다.
  - `MemoryCondition`의 `matches()`를 실행해보고 그 결과가 `true`이면 `MemoryConfig`는 정상 동작한다. 따라서 `memoryController`, `memoryFinder`가 빈으로 등록된다.
  - `MemoryCondition`의 실행결과가 `false`이면 `MemoryConfig`는 무효화 된다. 그래서 `memoryController`, `memoryFinder` 빈은 등록되지 않는다.

먼저 아무 조건을 주지 않고 실행해보자.

실행

- http://localhost:8080/memory

결과

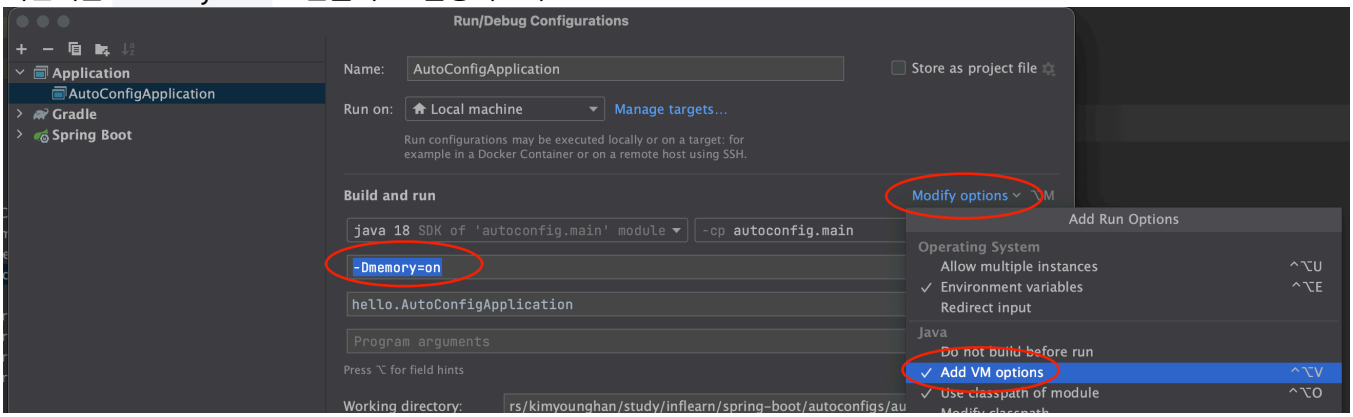
```
Whitelabel Error Page
```

- `memory=on`을 설정하지 않았기 때문에 동작하지 않는다.

다음 로그를 통해서 `MemoryCondition` 조건이 실행된 부분을 확인할 수 있다. 물론 결과는 `false`를 반환한다.

```
memory.MemoryCondition                    : memory=null
```

이번에는 `memory=on` 조건을 주고 실행해보자.



- VM 옵션을 추가하는 경우 `-Dmemory=on`를 사용해야 한다.

실행

- http://localhost:8080/memory

결과

```
{"used":24385432,"max":8589934592}
```

- `MemoryCondition` 조건이 `true`를 반환해서 빈이 정상 등록된다.

다음 로그를 확인할 수 있다.

```
memory.MemoryCondition                  : memory=on
memory.MemoryFinder                     : init memoryFinder
```

참고: 스프링이 로딩되는 과정은 복잡해서 `MemoryCondition`이 여러번 호출될 수 있다. 이 부분은 크게 중요하지 않으니 무시하자.

> **참고**
> 스프링은 외부 설정을 추상화해서 `Environment`로 통합했다. 그래서 다음과 같은 다양한 외부 환경 설정을 `Environment` 하나로 읽어들일 수 있다. 여기에 대한 더 자세한 내용은 뒤에서 다룬다.

```
#VM Options
#java -Dmemory=on -jar project.jar
-Dmemory=on

#Program arguments
# -- 가 있으면 스프링이 환경 정보로 사용
#java -jar project.jar --memory=on
--memory=on
```

```
#application.properties
#application.properties에 있으면 환경 정보로 사용
memory=on
```

@Conditional - 다양한 기능

지금까지 `Condition` 인터페이스를 직접 구현해서 `MemoryCondition`이라는 구현체를 만들었다. 스프링은 이미 필요한 대부분의 구현체를 만들어두었다. 이번에는 스프링이 제공하는 편리한 기능을 사용해보자.

MemoryConfig - 수정

```
package hello.config;

import memory.MemoryCondition;
import memory.MemoryController;
import memory.MemoryFinder;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

@Configuration
//@Conditional(MemoryCondition.class) //추가
@ConditionalOnProperty(name = "memory", havingValue = "on") //추가
public class MemoryConfig {

    @Bean
    public MemoryController memoryController() {
        return new MemoryController(memoryFinder());
    }

    @Bean
    public MemoryFinder memoryFinder() {
        return new MemoryFinder();
    }
}
```

- `@Conditional(MemoryCondition.class)`를 주석처리하자
- `@ConditionalOnProperty(name = "memory", havingValue = "on")`를 추가하자
  - 환경 정보가 `memory=on`이라는 조건에 맞으면 동작하고, 그렇지 않으면 동작하지 않는다.

    - 우리가 앞서 만든 기능과 동일하다.

@ConditionalOnProperty

```
package org.springframework.boot.autoconfigure.condition;

@Conditional(OnPropertyCondition.class)
public @interface ConditionalOnProperty {...}
```

- `@ConditionalOnProperty`도 우리가 만든 것과 동일하게 내부에는 `@Conditional`을 사용한다. 그리고 그 안에 `Condition` 인터페이스를 구현한 `OnPropertyCondition`를 가지고 있다.

실행

- http://localhost:8080/memory

실행해보면 앞서 만든 기능과 동일하게 동작하는 것을 확인할 수 있다.

@ConditionalOnXxx

스프링은 `@Conditional`과 관련해서 개발자가 편리하게 사용할 수 있도록 수 많은 `@ConditionalOnXxx`를 제공한다.

대표적인 몇가지를 알아보자.

- `@ConditionalOnClass`, `@ConditionalOnMissingClass`
    - 클래스가 있는 경우 동작한다. 나머지는 그 반대
- `@ConditionalOnBean`, `@ConditionalOnMissingBean`
    - 빈이 등록되어 있는 경우 동작한다. 나머지는 그 반대
- `@ConditionalOnProperty`
    - 환경 정보가 있는 경우 동작한다.
- `@ConditionalOnResource`
    - 리소스가 있는 경우 동작한다.
- `@ConditionalOnWebApplication`, `@ConditionalOnNotWebApplication`
    - 웹 애플리케이션인 경우 동작한다.
- `@ConditionalOnExpression`
    - SpEL 표현식에 만족하는 경우 동작한다.

> **ConditionalOnXxx 공식 메뉴얼**
> https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.developing-auto-configuration.condition-annotations

이름이 직관적이어서 바로 이해가 될 것이다. `@ConditionalOnXxx`는 주로 스프링 부트 자동 구성에 사용된다.

다음 자동 구성 클래스들을 열어서 소스 코드를 확인해보면 `@ConditionalOnXxx`가 아주 많이 사용되는 것을 확인할 수 있다.
`JdbcTemplateAutoConfiguration`, `DataSourceTransactionManagerAutoConfiguration`, `DataSourceAutoConfiguration`

> **참고**
> `@Conditional` 자체는 스프링 부트가 아니라 스프링 프레임워크의 기능이다. 스프링 부트는 이 기능을 확장해서 `@ConditionalOnXxx`를 제공한다.

정리
스프링 부트가 제공하는 자동 구성 기능을 이해하려면 다음 개념을 이해해야 한다.

- `@Conditional`: 특정 조건에 맞을 때 설정이 동작하도록 한다.
- `@AutoConfiguration`: 자동 구성이 어떻게 동작하는지 내부 원리 이해

지금까지 `@Conditional`에 대해서 알아보았으니, 지금부터는 `@AutoConfiguration`을 알아보자.

순수 라이브러리 만들기

`@AutoConfiguration`을 이해하기 위해서는 그 전에 먼저 라이브러리가 어떻게 사용되는지 이해하는 것이 필요하다.
여러분이 만든 실시간 자바 Memory 조회 기능이 좋다고 소문이 나서, 여러 프로젝트에서 사용하고 싶어한다. 이 기능을 여러곳에서 사용할 수 있도록 라이브러리로 만들어보자.

참고로 라이브러리를 만들 때는 스프링 부트 플러그인 기능을 사용하지 않고 진행한다.

프로젝트 설정 순서

1. `memory-v1-start`의 폴더 이름을 `memory-v1`으로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선

택하자.

build.gradle 확인

```
plugins {
    id 'java'
}

group = 'memory'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web:3.0.2'
    compileOnly 'org.projectlombok:lombok:1.18.24'
    annotationProcessor 'org.projectlombok:lombok:1.18.24'
    testImplementation 'org.springframework.boot:spring-boot-starter-test:3.0.2'
}

test {
    useJUnitPlatform()
}
```

- 스프링 부트 플러그인을 사용하게 되면 앞서 설명한 실행 가능한 Jar 구조를 기본으로 만든다.
- 여기서는 실행 가능한 Jar가 아니라, 다른곳에 포함되어서 사용할 순수 라이브러리 Jar를 만드는 것이 목적이므로 스프링 부트 플러그인을 사용하지 않았다.
- 스프링 컨트롤러가 필요하므로 `spring-boot-starter-web` 라이브러리를 선택했다.
- 스프링 부트 플러그인을 사용하지 않아서 버전을 직접 명시했다.

앞서 개발한 것과 같은 실시간 메모리 조회 기능을 추가하자

Memory

```
package memory;

public class Memory {
    private long used;
    private long max;

    public Memory(long used, long max) {
        this.used = used;
```

```
        this.max = max;
    }

    public long getUsed() {
        return used;
    }

    public long getMax() {
        return max;
    }

    @Override
    public String toString() {
        return "Memory{" +
                "used=" + used +
                ", max=" + max +
                '}';
    }
}
```

MemoryFinder

```
package memory;

import lombok.extern.slf4j.Slf4j;
import jakarta.annotation.PostConstruct;

@Slf4j
public class MemoryFinder {

    public Memory get() {
        long max = Runtime.getRuntime().maxMemory();
        long total = Runtime.getRuntime().totalMemory();
        long free = Runtime.getRuntime().freeMemory();
        long used = total - free;
        return new Memory(used, max);
    }

    @PostConstruct
    public void init() {
        log.info("init memoryFinder");
    }
}
```

MemoryController

```
package memory;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
@RequiredArgsConstructor
public class MemoryController {

    private final MemoryFinder memoryFinder;

    @GetMapping("/memory")
    public Memory system() {
        Memory memory = memoryFinder.get();
        log.info("memory={}", memory);
        return memory;
    }
}
```

MemoryFinderTest

```
package memory;

import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

class MemoryFinderTest {

    @Test
    void get() {
        MemoryFinder memoryFinder = new MemoryFinder();
        Memory memory = memoryFinder.get();
        assertThat(memory).isNotNull();
    }
}
```

- 간단한 테스트를 통해서 데이터가 조회 되는지 정도만 간단히 검증해보자.

빌드하기

- 다음 명령어로 빌드하자.
  - `./gradlew clean build`
- 빌드 결과
  - `build/libs/memory-v1.jar`

- 다음 명령어를 사용해서 압축을 풀어서 내용을 확인해보자.
  - `jar -xvf memory-v1.jar`

JAR를 푼 결과

- `META-INF`
  - `MANIFEST.MF`
- `memory`
  - `MemoryFinder.class`
  - `MemoryController.class`
  - `Memory.class`

`memory-v1.jar`는 스스로 동작하지는 못하고 다른 곳에 포함되어서 동작하는 라이브러리이다. 이제 이 라이브러리를 다른 곳에서 사용해보자.

순수 라이브러리 사용하기1

프로젝트 설정 순서

1. `project-v1-start`의 폴더 이름을 `project-v1`으로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
```

```
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
- `Lombok`, `Spring Web`

프로젝트가 동작하는지 확인하기 위해 간단한 컨트롤러를 하나 추가하자

HelloController

```
package hello.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
```

```
    public String hello() {
        return "hello";
    }
}
```

실행

http://localhost:8080/hello

결과

`hello`

순수 라이브러리 사용하기2

앞서 만든 `memory-v1.jar` 라이브러리를 `project-v1` 에 적용해보자.

라이브러리 추가

- `project-v1/libs` 폴더를 생성하자.
- `memory-v1` 프로젝트에서 빌드한 `memory-v1.jar` 를 이곳에 복사하자.
- `project-v1/build.gradle` 에 `memory-v1.jar` 를 추가하자.

```
dependencies {
    implementation files('libs/memory-v1.jar') //추가
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- 라이브러리를 jar 파일로 직접 가지고 있으면 `files` 로 지정하면 된다.
- gradle을 리로드하자.

주의

- 추가한 폴더 이름이 `lib` 가 아니라 `libs` 인 점을 주의하자!
- 파일로 추가한 라이브러리를 IntelliJ가 잘 인식하지 못하는 경우에는 다음과 같이 프로젝트를 다시 임포트

하자.

프로젝트 임포트

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

라이브러리 설정

라이브러리를 스프링 빈으로 등록해서 동작하도록 만들어보자.

MemoryConfig

```
package hello.config;

import memory.MemoryController;
import memory.MemoryFinder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MemoryConfig {

    @Bean
    public MemoryFinder memoryFinder() {
        return new MemoryFinder();
    }

    @Bean
    public MemoryController memoryController() {
        return new MemoryController(memoryFinder());
    }
}
```

- 스프링 부트 자동 구성을 사용하는 것이 아니기 때문에 빈을 직접 하나하나 등록해주어야 한다.

메모리 조회 기능이 잘 동작하는지 확인해보자.

서버 실행 로그

```
MemoryFinder                    : init memoryFinder
```

실행

- http://localhost:8080/memory

결과

```
{"used": 38174528, "max": 8589934592}
```

메모리 조회 라이브러리가 잘 동작하는 것을 확인할 수 있다.

정리

- 외부 라이브러리를 직접 만들고 또 그것을 프로젝트에 라이브러리로 불러서 적용해보았다.
- 그런데 라이브러리를 사용하는 클라이언트 개발자 입장을 생각해보면, 라이브러리 내부에 있는 어떤 빈을 등록해야하는지 알아야 하고, 그것을 또 하나하나 빈으로 등록해야 한다. 지금처럼 간단한 라이브러리가 아니라 초기 설정이 복잡하다면 사용자 입장에서는 상당히 귀찮은 작업이 될 수 있다.
- 이런 부분을 자동으로 처리해주는 것이 바로 스프링 부트 자동 구성(Auto Configuration)이다.

자동 구성 라이브러리 만들기

우리가 만든 라이브러리를 사용해주는 고마운 고객 개발자를 위해, 프로젝트에 라이브러리를 추가만 하면 모든 구성이 자동으로 처리되도록 해보자. 쉽게 이야기해서 스프링 빈들이 자동으로 등록되는 것이다. 여기에 추가로 `memory=on` 옵션도 적용할 수 있게 해보자.

이렇게 하려면 메모리 라이브러리의 기능을 업그레이드 해야한다.
기존 프로젝트를 유지하기 위해 프로젝트를 복사하고 일부 수정하자

- `memory-v1` 프로젝트를 복사해서 `memory-v2` 를 만들자.

settings.gradle - 수정

```
rootProject.name = 'memory-v2' //v1 -> v2로 수정
```

자동 구성 추가

MemoryAutoConfig

```
package memory;

import org.springframework.boot.autoconfigure.AutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
```

```java
import org.springframework.context.annotation.Bean;

@AutoConfiguration
@ConditionalOnProperty(name = "memory", havingValue = "on")
public class MemoryAutoConfig {

    @Bean
    public MemoryController memoryController() {
        return new MemoryController(memoryFinder());
    }

    @Bean
    public MemoryFinder memoryFinder() {
        return new MemoryFinder();
    }
}
```

- `@AutoConfiguration`
  - 스프링 부트가 제공하는 자동 구성 기능을 적용할 때 사용하는 애노테이션이다.
- `@ConditionalOnProperty`
  - `memory=on` 이라는 환경 정보가 있을 때 라이브러리를 적용한다. (스프링 빈을 등록한다.)
  - 라이브러리를 가지고 있더라도 상황에 따라서 해당 기능을 켜고 끌 수 있게 유연한 기능을 제공한다.

자동 구성 대상 지정

- 이 부분이 중요하다. 스프링 부트 자동 구성을 적용하려면, 다음 파일에 자동 구성 대상을 꼭 지정해주어야 한다.
- 폴더 위치와 파일 이름이 길기 때문에 주의하자

파일 생성

`src/main/resources/META-INF/spring/`
`org.springframework.boot.autoconfigure.AutoConfiguration.imports`

org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
memory.MemoryAutoConfig
```

- 앞서 만든 자동 구성인 `memory.MemoryAutoConfig` 를 패키지를 포함해서 지정해준다.
- 스프링 부트는 시작 시점에 `org.springframework.boot.autoconfigure.AutoConfiguration.imports` 의 정보를 읽어서 자동 구성으로 사용한다. 따라서 내부에 있는 `MemoryAutoConfig` 가 자동으로 실행된다.

빌드하기

- 다음 명령어로 빌드하자.
  - `./gradlew clean build`
- 빌드 결과
  - `build/libs/memory-v2.jar`

스프링 부트 자동 구성 기능이 포함된 `memory-v2.jar`를 이제 프로젝트에 적용해보자.

자동 구성 라이브러리 사용하기1

기존 프로젝트를 유지하기 위해 새로운 프로젝트에 자동 구성 라이브러리를 적용해보자. 앞서 만든 `project-v1`과 비슷한 프로젝트이다.

프로젝트 설정 순서

1. `project-v2-start`의 폴더 이름을 `project-v2`으로 변경하자.
2. **프로젝트 임포트**

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

build.gradle 확인

```
plugins {
    id 'org.springframework.boot' version '3.0.2'
    id 'io.spring.dependency-management' version '1.1.0'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

configurations {
    compileOnly {
```

```
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- 스프링 부트에서 다음 라이브러리를 선택했다.
- `Lombok`, `Spring Web`

프로젝트가 동작하는지 확인하기 위해 간단한 컨트롤러를 하나 추가하자

HelloController

```
package hello.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

실행

http://localhost:8080/hello

결과

`hello`

자동 구성 라이브러리 사용하기2

앞서 만든 `memory-v2.jar` 라이브러리를 `project-v2`에 적용해보자.

라이브러리 추가

- `project-v2/libs` 폴더를 생성하자.
- `memory-v2` 프로젝트에서 빌드한 `memory-v2.jar`를 이곳에 복사하자.
- `project-v2/build.gradle`에 `memory-v2.jar`를 추가하자.

```
dependencies {
    implementation files('libs/memory-v2.jar') //추가
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- 라이브러리를 파일로 직접 가지고 있으면 `files`로 지정하면 된다.
- gradle을 리로드하자.

주의

- 추가한 폴더 이름이 `lib`가 아니라 `libs`인 점을 주의하자!
- 지금처럼 로컬에 파일로 추가한 라이브러리를 IntelliJ가 잘 인식하지 못하는 경우에는 다음과 같이 프로젝트를 다시 임포트 하자.

프로젝트 임포트

File ➜ Open ➜ 해당 프로젝트의 `build.gradle`을 선택하자. 그 다음에 선택창이 뜨는데, Open as Project를 선택하자.

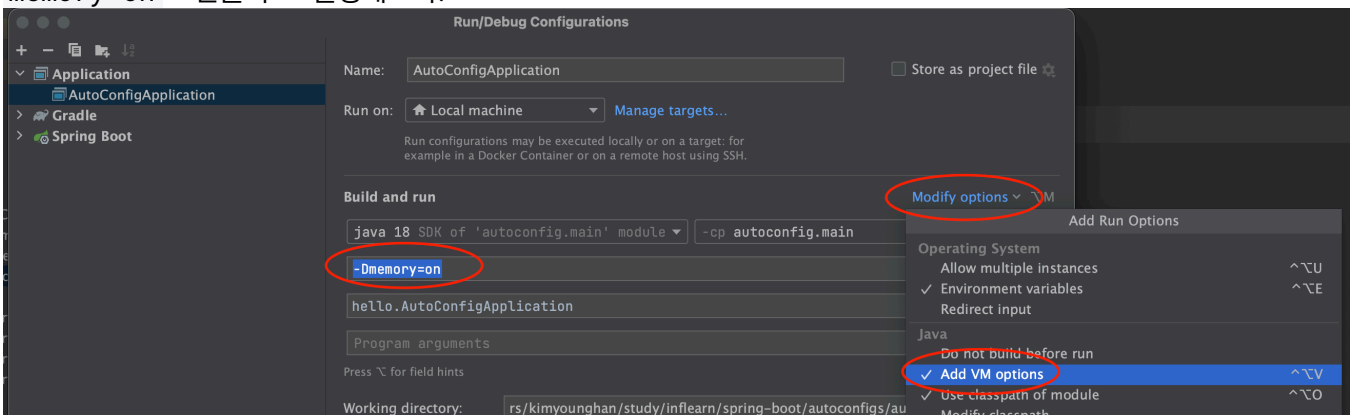
라이브러리 설정

- 앞서 `project-v1`에서는 `memory-v1`을 사용하기 위해 스프링 빈을 직접 등록했다.
- `project-v2`에서 사용하는 `memory-v2` 라이브러리에는 스프링 부트 자동 구성이 적용되어 있다. 따라서 빈을 등록하는 별도의 설정을 하지 않아도 된다.

`memory-v2`의 자동 구성에는 다음과 같이 설정했기 때문에 `memory=on` 조건을 만족할 때만 실행된다.

```
@AutoConfiguration
@ConditionalOnProperty(name = "memory", havingValue = "on")
public class MemoryAutoConfig {...}
```

`memory=on` 조건을 주고 실행해보자.



- VM 옵션을 추가하는 경우 `-Dmemory=on`를 사용해야 한다.

메모리 조회 기능이 잘 동작하는지 확인해보자.

서버 실행 로그

```
MemoryFinder                    : init memoryFinder
```

실행

- http://localhost:8080/memory

결과

```
{"used": 38174528, "max": 8589934592}
```

메모리 조회 라이브러리가 잘 동작하는 것을 확인할 수 있다.

`memory=on` 조건을 끄면 라이브러리를 사용하지 않는 것도 확인할 수 있다.

정리

- 스프링 부트가 제공하는 자동 구성 덕분에 복잡한 빈 등록이나 추가 설정 없이 단순하게 라이브러리의 추가

만으로 프로젝트를 편리하게 구성할 수 있다.

- `@ConditionalOnXxx` 덕분에 라이브러리 설정을 유연하게 제공할 수 있다.
- 스프링 부트는 수 많은 자동 구성을 제공한다. 그 덕분에 스프링 라이브러리를 포함해서 수 많은 라이브러리를 편리하게 사용할 수 있다.

자동 구성 이해1 - 스프링 부트의 동작

스프링 부트는 다음 경로에 있는 파일을 읽어서 스프링 부트 자동 구성으로 사용한다.

`resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

우리가 직접 만든 `memory-v2` 라이브러리와 스프링 부트가 제공하는 `spring-boot-autoconfigure` 라이브러리의 다음 파일을 확인해보면 스프링 부트 자동 구성을 확인할 수 있다.

memory-v2 - org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
memory.MemoryAutoConfig
```

spring-boot-autoconfigure - org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfig
uration
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfigurat
ion
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoCo
nfiguration
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfigurati
on
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfigurati
on
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConf
```

```
iguration
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfigurati
on
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration
...
```

이번에는 스프링 부트가 어떤 방법으로 해당 파일들을 읽어서 동작하는지 알아보자.
이해를 돕기 위해 앞서 개발한 `autoconfig` 프로젝트를 열어보자.

스프링 부트 자동 구성이 동작하는 원리는 다음 순서로 확인할 수 있다.
`@SpringBootApplication` ➜ `@EnableAutoConfiguration` ➜ `@Import(AutoConfigurationImportSelector.class)`

스프링 부트는 보통 다음과 같은 방법으로 실행한다.

AutoConfigApplication

```
@SpringBootApplication
public class AutoConfigApplication {
    public static void main(String[] args) {
        SpringApplication.run(AutoConfigApplication.class, args);
    }
}
```

- `run()`에 보면 `AutoConfigApplication.class`를 넘겨주는데, 이 클래스를 설정 정보로 사용한다는 뜻이다. `AutoConfigApplication`에는 `@SpringBootApplication` 애노테이션이 있는데, 여기에 중요한 설정 정보들이 들어있다.

@SpringBootApplication

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {...}
```

- 여기서 우리가 주목할 애노테이션은 `@EnableAutoConfiguration`이다. 이름 그대로 자동 구성을 활성화 하는 기능을 제공한다.

@EnableAutoConfiguration

```
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {…}
```

- `@Import`는 주로 스프링 설정 정보(`@Configuration`)를 포함할 때 사용한다.
- 그런데 `AutoConfigurationImportSelector`를 열어보면 `@Configuration`이 아니다.

이 기능을 이해하려면 `ImportSelector`에 대해 알아야 한다.

자동 구성 이해2 - ImportSelector

`@Import`에 설정 정보를 추가하는 방법은 2가지가 있다.

- 정적인 방법: `@Import`(클래스) 이것은 정적이다. 코드에 대상이 딱 박혀 있다. 설정으로 사용할 대상을 동적으로 변경할 수 없다.
- 동적인 방법: `@Import`(`ImportSelector`) 코드로 프로그래밍해서 설정으로 사용할 대상을 동적으로 선택할 수 있다.

정적인 방법

스프링에서 다른 설정 정보를 추가하고 싶으면 다음과 같이 `@Import`를 사용하면 된다.

```
@Configuration
@Import({AConfig.class, BConfig.class})
public class AppConfig {...}
```

- 그런데 예제처럼 `AConfig`, `BConfig`가 코드에 딱 정해진 것이 아니라, 특정 조건에 따라서 설정 정보를 선택해야 하는 경우에는 어떻게 해야할까?

동적인 방법

- 스프링은 설정 정보 대상을 동적으로 선택할 수 있는 `ImportSelector` 인터페이스를 제공한다.

ImportSelector

```
package org.springframework.context.annotation;
public interface ImportSelector {
    String[] selectImports(AnnotationMetadata importingClassMetadata);
  //...
```

```
}
```

이해를 돕기 위해 간단하게 `ImportSelector`를 사용하는 예제를 만들어보자.

ImportSelector 예제

다음 예제들은 모두 `src/test` 하위에 만들자

HelloBean

```
package hello.selector;

public class HelloBean {
}
```

- 빈으로 등록할 대상이다.

HelloConfig

```
package hello.selector;

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

@Configuration
public class HelloConfig {

    @Bean
    public HelloBean helloBean() {
        return new HelloBean();
    }
}
```

- 설정 정보이다. `HelloBean`을 스프링 빈으로 등록한다.

HelloImportSelector

```
package hello.selector;

import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.type.AnnotationMetadata;

public class HelloImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
```

```
            return new String[]{"hello.selector.HelloConfig"};
        }
    }
```

- 설정 정보를 동적으로 선택할 수 있게 해주는 `ImportSelector` 인터페이스를 구현했다.
- 여기서는 단순히 `hello.selector.HelloConfig` 설정 정보를 반환한다.
- 이렇게 반환된 설정 정보는 선택되어서 사용된다.
- 여기에 설정 정보로 사용할 클래스를 동적으로 프로그래밍 하면 된다.

ImportSelectorTest

```
package hello.selector;

import org.junit.jupiter.api.Test;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

import static org.assertj.core.api.Assertions.*;

public class ImportSelectorTest {

    @Test
    void staticConfig() {
        AnnotationConfigApplicationContext appContext =
                new AnnotationConfigApplicationContext(StaticConfig.class);
        HelloBean bean = appContext.getBean(HelloBean.class);
        assertThat(bean).isNotNull();
    }

    @Test
    void selectorConfig() {
        AnnotationConfigApplicationContext appContext =
                new AnnotationConfigApplicationContext(SelectorConfig.class);
        HelloBean bean = appContext.getBean(HelloBean.class);
        assertThat(bean).isNotNull();
    }

    @Configuration
    @Import(HelloConfig.class)
    public static class StaticConfig {
    }
```

```
    @Configuration
    @Import(HelloImportSelector.class)
    public static class SelectorConfig {
    }

}
```

staticConfig()

- `staticConfig()`는 이해하는데 어려움이 없을 것이다. 스프링 컨테이너를 만들고, `StaticConfig.class`를 초기 설정 정보로 사용했다. 그 결과 `HelloBean`이 스프링 컨테이너에 잘 등록된 것을 확인할 수 있다.

selectorConfig()

- `selectorConfig()`는 `SelectorConfig`를 초기 설정 정보로 사용한다.
- `SelectorConfig`는 `@Import(HelloImportSelector.class)`에서 `ImportSelector`의 구현체인 `HelloImportSelector`를 사용했다.
- 스프링은 `HelloImportSelector`를 실행하고, `"hello.selector.HelloConfig"`라는 문자를 반환 받는다.
- 스프링은 이 문자에 맞는 대상을 설정 정보로 사용한다. 따라서 `hello.selector.HelloConfig`이 설정 정보로 사용된다.
- 그 결과 `HelloBean`이 스프링 컨테이너에 잘 등록된 것을 확인할 수 있다.

@EnableAutoConfiguration 동작 방식

이제 `ImportSelector`를 이해했으니 다음 코드를 이해할 수 있다.

@EnableAutoConfiguration

```
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {…}
```

- `AutoConfigurationImportSelector`는 `ImportSelector`의 구현체이다. 따라서 설정 정보를 동적으로 선택할 수 있다.
- 실제로 이 코드는 모든 라이브러리에 있는 다음 경로의 파일을 확인한다.
- `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`

memory-v2 - org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
memory.MemoryAutoConfig
```

spring-boot-autoconfigure - org.springframework.boot.autoconfigure.AutoConfiguration.imports

```
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfigurati
on
...
```

그리고 파일의 내용을 읽어서 설정 정보로 선택한다.

스프링 부트 자동 구성이 동작하는 방식은 다음 순서로 확인할 수 있다.

- `@SpringBootApplication` ➜ `@EnableAutoConfiguration` ➜ `@Import(AutoConfigurationImportSelector.class)` ➜
- `resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 파일을 열어서 설정 정보 선택
- 해당 파일의 설정 정보가 스프링 컨테이너에 등록되고 사용

정리

스프링 부트의 자동 구성을 직접 만들어서 사용할 때는 다음을 참고하자.

- `@AutoConfiguration`에 자동 구성의 순서를 지정할 수 있다.
- `@AutoConfiguration`도 설정 파일이다. 내부에 `@Configuration`이 있는 것을 확인할 수 있다.
  - 하지만 일반 스프링 설정과 라이프사이클이 다르기 때문에 컴포넌트 스캔의 대상이 되면 안된다.
  - 파일에 지정해서 사용해야 한다.
  - `resources/META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`
  - 그래서 스프링 부트가 제공하는 컴포넌트 스캔에서는 `@AutoConfiguration`을 제외하는 `AutoConfigurationExcludeFilter` 필터가 포함되어 있다.

@SpringBootApplication

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
```

```
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {...}
```

- 자동 구성이 내부에서 컴포넌트 스캔을 사용하면 안된다. 대신에 자동 구성 내부에서 `@Import`는 사용할 수 있다.

자동 구성을 언제 사용하는가?

- `AutoConfiguration`은 라이브러리를 만들어서 제공할 때 사용하고, 그 외에는 사용하는 일이 거의 없다. 왜냐하면 보통 필요한 빈들을 컴포넌트 스캔하거나 직접 등록하기 때문이다. 하지만 라이브러리를 만들어서 제공할 때는 자동 구성이 유용하다. 실제로 다양한 외부 라이브러리들이 자동 구성을 함께 제공한다.
- 보통 이미 만들어진 라이브러리를 가져다 사용하지, 반대로 라이브러리를 만들어서 제공하는 경우는 매우 드물다. 그럼 자동 구성은 왜 알아두어야 할까?
- 자동 구성을 알아야 하는 진짜 이유는 개발을 진행 하다보면 사용하는 특정 빈들이 어떻게 등록된 것인지 확인이 필요할 때가 있다. 이럴 때 스프링 부트의 자동 구성 코드를 읽을 수 있어야 한다. 그래야 문제가 발생했을 때 대처가 가능하다. 자동화는 매우 편리한 기능이지만 자동화만 믿고 있다가 실무에서 문제가 발생했을 때는 파고 들어가서 문제를 확인하는 정도는 이해해야 한다. 이번에 학습한 정도면 자동 구성 코드를 읽는데 큰 어려움은 없을 것이다.

남은 문제

그런데 이런 방식으로 빈이 자동 등록되면, 빈을 등록할 때 사용하는 설정 정보는 어떻게 변경해야 하는지 의문이 들 것이다. 예를 들어서 DB 접속 URL, ID, PW 같은 것 말이다. 데이터소스 빈을 등록할 때 이런 정보를 입력해야 하는데, 빈이 자동으로 다 등록이 되어 버린다면 이런 정보를 어떻게 입력할 수 있을까? 다음 장을 통해 알아보자.