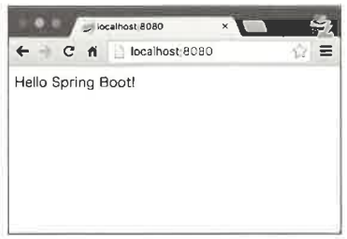
```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.0.RELEASE)

2016-08-07 23:16:28.977  INFO 43118 --- [  restartedMain] com.example.Haj
ibootApplication          : Starting HajibootApplication on xxx with PID
43118 (/xxx/workspace/hajiboot/target/classes started by makit in /xxx/
workspace/hajiboot)
(略)
2016-08-07 23:16:29.611  INFO 43118 --- [  restartedMain] s.b.c.e.t.T
omcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2016-08-07 23:16:29.613  INFO 43118 --- [  restartedMain] com.examp
le.HajibootApplication        : Started HajibootApplication in 0.67
 seconds (JVM running for 27.45)
```

*

もう一度、Web ブラウザで「http://localhost:8080」にアクセスしてください。



「http://localhost:8080」にアクセス

アプリケーションを再起動することなく、「ソース・コード」の変更が反映されました。

これで効率良く開発を進めることができるでしょう。

*

「Spring Dev Tools」は「Hot Reloading」の他に、「テンプレート・エンジン」のキャッシュ無効化も行ないます。

【3.3「Thymeleaf」を使った、「画面のある Web アプリ」の開発】で扱う「Thymeleaf」はデフォルトでテンプレートである HTML をキャッシュします。

開発中に HTML がキャッシュされると、HTML を修正するたびにアプリケーションを再起動しなくてはならず、効率が悪いです。

「Spring Dev Tools」を導入することで、再起動することなく HTML の変更を起動中のアプリケーションに反映させることができます。

また、「Spring Dev Tools」によるリロード機能は、「jar ファイル」を実行する際には無効化されるので、安心して開発中のみ本機能を利用できます。

ただし、100% リロードが成功するわけではありません。リロードがうまく行なわれない場合は、アプリケーションを再起動してください。

第2章 速習「Spring Framework」

前章で説明したように、「Spring Boot」は「Spring Framework」でアプリケーションを簡単に作るための仕組みであり、「Spring Boot」単品では、アプリケーションは作れません。

次章で「Web アプリケーション」を作りますが、まずは本章で、「DI」や「データ・アクセス」といった「Spring Framework」の基本的な要素を説明します。

2.1 「Spring Framework」による DI

「DI」とは「Dependency Injection」(依存性の注入)の略で、「Spring Framework」の根幹となる技術です。

「DI」によってクラス間の依存関係が自動で解決されます。

「インスタンス」の管理は「DI コンテナ」が行ないます。
「DI コンテナ」が「インスタンス」を生成し、その「インスタンス」に必要な「インスタンス」を設定した状態で、アプリケーションに返します。

「DI コンテナ」経由で「インスタンス」が生成されることによって、

- インスタンスの「スコープ」を制御できる
  (「シングルトンオブジェクトなのか」「毎回新規生成するのか」など)
- インスタンスの「ライフ・サイクル」をイベント制御できる
  (「インスタンス生成時」「破棄時」のイベント処理など)
- 共通的な処理を埋め込める
  (「トランザクション管理」や「ロギング処理」など)

といった副次効果を加えることができます。

また、オブジェクト間が疎結合になるため、「ユニット・テスト」がしやすくなるというメリットもあります。

「Spring Framework」による DI を理解するために、唐突ですが、「何らかの計算して、結果を表示する簡単なアプリケーション」を作りましょう。

このアプリケーションをステップ・バイ・ステップで変更していくことで、「Spring Framework」が提供する DI の機能を説明します。

[2.1.1] プロジェクトの作成

[1.3] 節の「はじめての「Spring Boot」」と同様に、「Spring Initializr」でプロジェクトの雛形を作ります。

今回は、次の項目を入力してください。

「Spring Initializr」の入力項目

Artifact	Search for dependencies
hajiboot-di	未入力

ダウンロードした「hajiboot-di.zip」を展開してください。

*

展開されたフォルダに存在する「pom.xml」を確認しましょう。

【pom.xml】雛形の確認

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>hajiboot-di</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hajiboot-di</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <!-- (1) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
```



```
    </dependency>
  </dependencies>
</project>
```

プログラム解説

項　番	説　明
(1)	1 章では「spring-boot-starter-web」を指定したが、今回は「spring-boot-starter」を指定。 「spring-boot-starter-web」では、「Web アプリ」に必要な「ライブラリ」が含まれるが、今回は「DI」のみ扱うので、不要。

「hajiboot-di」プロジェクトを、[1.3.5] と同様に、STS にインポートしてください。

*

まずは、以下のようなインターフェイスを作ります。

Calculator インターフェイス

```
package com.example.app;

public interface Calculator {
    int calc(int a, int b);
}
```

次に、このインターフェイスの「実装クラス」を作ります。
ここでは簡単な「足し算」を実装します。

AddCalculator クラス

```
package com.example.app;

public class AddCalculator implements Calculator {
    @Override
    public int calc(int a, int b) {
        return a + b;
    }
}
```

●「Bean 定義ファイル」の作成

「Calculator インターフェイス」に対して、どのような「実装」(Bean) を提供するかを、「DI コンテナ」に管理させます。

「DI コンテナ」に「Bean」を管理させるために、「Bean 定義ファイル」を作ります。

「Spring Framework」には「Bean 定義ファイル」の形式が大きく分けて、

(a)「XML」で定義する。
(b)「Java クラス」(JavaConfig) で定義する。

の 2 つがあります。

(b) の「JavaConfig」は、「Spring Framework」の「バージョン 3」から導入された形式で、本書は「JavaConfig」を使って説明します。

AppConfig クラス

```
package com.example;
import com.example.app.AddCalculator;
import com.example.app.Calculator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration // (1)
public class AppConfig {
    @Bean // (2)
    Calculator calculator() {
        return new AddCalculator(); // (3)
    }
}
```

プログラム解説

項 番	説 明
(1)	「@Configuration」アノテーションを付けることで、このクラスが「JavaConfig」用のクラスであることを示す。
(2)	「DI コンテナ」に管理させたい「Bean」を生成するメソッドに、「@Bean」アノテーションを付ける。 デフォルトでは「メソッド名」が「Bean 名」になる。 また、デフォルトではこのメソッドで生成された「インスタンス」は「singleton」として管理され、「DI コンテナ」につき「1 つのインスタンス」のみ生成される。
(3)	この例では「AddCalculator インスタンス」が「Calculator 型」で「calculator」という名前で「DI コンテナ」に「singleton」として管理される。

●「エントリ・ポイント」の作成

この「JavaConfig」を読み込んで、「Calculator」を実行するアプリケーションのエントリポイントを作りましょう。

HajibootDiApplication クラス

```
package com.example;

import com.example.app.Calculator;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Import;

import java.util.Scanner;

@EnableAutoConfiguration // (1)
@Import(AppConfig.class) // (2)
public class HajibootDiApplication {

  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(HajibootDiAppl
ication.class, args); // (3)
    Scanner scanner = new Scanner(System.in); // (4)
    System.out.print("Enter 2 numbers like 'a b' : ");
    int a = scanner.nextInt();
    int b = scanner.nextInt();

    Calculator calculator = context.getBean(Calculator.class); // (5)
    int result = calculator.calc(a, b);

    System.out.println("result = " + result);
  }
}
```

プログラム解説

項 番	説 明
(1)	「@EnableAutoConfiguration」アノテーションが Spring Boot の自動設定を有効するにするためのアノテーションである。第 1 章で見てきたように、Spring Initializr で作成された雛形クラスには「@EnableAutoConfiguration」ではなく、「@SpringBootApplication」アノテーションが付与されている。「@SpringBootApplication」は Spring Boot 1.2 から導入され、「@EnableAutoConfiguration」、「@Configuration」及び、[2.1.4] で説明する「@ComponentScan」アノテーションの 3 つを組み合わせたものある。説明の順序上、ここでは「@EnableAutoConfiguration」を用いる。
(2)	「JavaConfig」を読み込むために、「@Import」で「@Configuration」アノテーションを付けた対象のクラスを指定。
(3)	「SpringApplication.run」で「Spring Boot」アプリケーションを起動。 「第一引数」には「@EnableAutoConfiguration」を付けたクラスを指定。 このメソッドの返り値は「DI コンテナ」の本体である「ApplicationContext」になる。
(4)	「Scanner」クラスを使って、簡単に標準入力からデータを取得※。
(5)	「getBean」メソッドを用いて、「DI コンテナ」から「Calculator 型」のインスタンスを取得。 「Calculator」の実体は「DI コンテナ」によって解決され、アプリケーションは意識しない。

※ 本書では省略しましたが、本来は「Scanner」オブジェクトは Close すべきです。

●実 行

このクラスを実行します。

【コマンド・プロンプト】「100 200」と入力し、「Enter」キーを実行

```
Enter 2 numbers like 'a b' :
```

と聞かれるので、「100 200」と入力し、「Enter」キーを実行すると、

【コマンド・プロンプト】出力結果

```
result = 300
```

という結果が出力されます。

「HajibootDiApplication クラス」は「DI コンテナ」から作成ずみの「Calculator」オブジェクトを取得するだけであるため、「App クラス」からは「Calculator」がど

のように作られているかが隠蔽されました。
「DI コンテナ」を使うことでアプリケーション内のモジュール結合度が低くなり、独立性が高くなります。

[2.1.2] アプリケーションの抽象化

次に、もう少しアプリケーションを抽象化してみましょう。
「Calculator」の引数を作るために「ArgumentResolver」インターフェイスを作ります。

ArgumentResolver インターフェイス

```
package com.example.app;

import java.io.InputStream;

public interface ArgumentResolver {
    Argument resolve(InputStream stream);
}
```

「引数オブジェクト」として「Argument クラス」を作ります。

ここで、「Java クラス」の作成を容易にするため、「Lombok」を使います[1]。
「Lombok」を使うために、「pom.xml」に以下を追加する必要があります。

[pom.xml]「Lombok」の依存関係の追加

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

「STS」に「Lombok」の設定をしていない場合は、「**[附録 C]「Lombok」のインストール**」を参照して、セットアップしてください。

Argument クラス

```
package com.example.app;

import lombok.Data;

@Data // (1)
public class Argument {
    private final int a;
    private final int b;
}
```

[1]「Spring Boot」とは関係ありませんが、今後の開発が簡単になるため、この段階でセットアップしておきます。

プログラム解説

項 番	説 明
(1)	「@Data」アノテーションを付けることで、コンパイル時 (class ファイル生成時) に、各フィールドの「setter/getter」「toString メソッド」「equals メソッド」「hashCode メソッド」が生成される。 そのため、ソース・コードから冗長な部分がなくなり、スッキリする。 (今回の例ではフィールドに「final 修飾子」がついているので「setter」は生成されない)。

今回の例では「final 修飾子」が付いたフィールドをもつため、「setter」は生成されず、「a」と「b」を引数にもつ「コンストラクタ」が生成されます。

「ArgumentResolver」の実装として、先ほどと同様に「java.util.Scanner」を使って引数を取得するクラスを作ります。

ScannerArgumentResolver クラス

```
package com.example.app;

import java.io.InputStream;
import java.util.Scanner;

public class ScannerArgumentResolver implements ArgumentResolver {
    @Override
    public Argument resolve(InputStream stream) {
        Scanner scanner = new Scanner(stream);
        int a = scanner.nextInt();
        int b = scanner.nextInt();
        return new Argument(a, b);
    }
}
```

「Bean 定義ファイル」に「ArgumentResolver」の「実装クラス」を定義します。

AppConfig クラス

```
package com.example;

import com.example.app.AddCalculator;
import com.example.app.ArgumentResolver;
import com.example.app.Calculator;
import com.example.app.ScannerArgumentResolver;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
    @Bean
    Calculator calculator() {
        return new AddCalculator();
    }

    @Bean
```

```
    ArgumentResolver argumentResolver() {
        return new ScannerArgumentResolver();
    }
}
```

「App クラス」内で「ArgumentResolver」を「DI コンテナ」から取得するように修正します。

HajibootDiApplication クラス

```
package com.example;

import com.example.app.Argument;
import com.example.app.ArgumentResolver;
import com.example.app.Calculator;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Import;

@EnableAutoConfiguration
@Import(AppConfig.class)
public class HajibootDiApplication {

  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(HajibootDiAppl
ication.class, args);
    System.out.print("Enter 2 numbers like 'a b' : ");
    ArgumentResolver argumentResolver = context.getBean(ArgumentResol
ver.class);
    Argument argument = argumentResolver.resolve(System.in);
    Calculator calculator = context.getBean(Calculator.class);
    int result = calculator.calc(argument.getA(), argument.getB());
    System.out.println("result = " + result);
  }
}
```

これで「どのような計算をする」「どのようにして引数を取得するか」を外部化し、「DI コンテナ」でどの実体を使うかを解決することによって、「App クラス」の汎用性が高まりました。

*

ここまで「DI コンテナ」から「Bean」を取得する方法について説明してきました。

実は、まだ「Dependency Injection」自体は行なっていません。

さて、HajibootDiApplication クラス内で「context.getBean」の呼び出しが多くなってきました。この調子で関連するモジュールが増えていくと、スッキリしないコードになります。

これを解決するのが「DI」です。次に「DI」を用いてこのアプリケーションを修正します。

[2.1.3] 「オート・ワイヤリング」による DI

これまでは App クラス内で「DI コンテナ」から明示的に「Bean」を取得してきました。

こんどは「DI コンテナ」に、「インスタンス」へ「Bean」をインジェクションしてもらいます。

「App クラス」で実装した処理を、集約する「Frontend クラス」に作ります。

この「Frontend クラス」に、作成ずみの「Calculator」と「ArgumentResolver」を注入してもらいましょう。

Frontend クラス

```
package com.example.app;

import org.springframework.beans.factory.annotation.Autowired;

public class Frontend {
    @Autowired // (1)
    ArgumentResolver argumentResolver;
    @Autowired
    Calculator calculator;

    public void run() {
        System.out.print("Enter 2 numbers like 'a b' : ");
        Argument argument = argumentResolver.resolve(System.in);
        int result = calculator.calc(argument.getA(), argument.getB());
        System.out.println("result = " + result);
    }
}
```

プログラム解説

項 番	説 明
(1)	「@Autowired」アノテーションを付け、「DI コンテナ」が「インジェクション」すべき「フィールド」であることを示す。

「@Autowired」アノテーションを付けた「フィールド」[2] をもつクラスが「DI コンテナ」で管理されるとします。

すると、「DI コンテナ」は自動的に「@Autowired」アノテーションを付けた「フィールド」に対して、合致する型のオブジェクトを管理内のオブジェクトから探し出して、インジェクションします。

このような仕組みを「オート・ワイヤリング」と言います。

*

「Frontend」クラスも「Bean 定義ファイル」に定義しましょう。

AppConfig クラス

```
package com.example;

import com.example.app.*;
```

[2]「Spring Framework」では「フィールド」だけでなく、「セッター」「コンストラクタ」にも「インジェクション」が可能です。本書では「フィールド」に対する「インジェクション」についてのみ説明します。

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

@Configuration
public class AppConfig {
    @Bean
    Calculator calculator() {
        return new AddCalculator();
    }

    @Bean
    ArgumentResolver argumentResolver() {
        return new ScannerArgumentResolver();
    }

    @Bean
    Frontend frontend() {
        return new Frontend();
    }
}
```

これまでの例ではインターフェイスに対して実装を提供する形で定義していました。

しかし、「Bean 定義ファイル」の定義方法にそのような制約はなく、任意のクラスを定義できます。

ノート 「オート・ワイヤリング」の仕組みを使わない場合は、「Frontend」クラスの「@Autowired」アノテーションを外した上で、セッターメソッドを作り、「Bean 定義ファイル」で、以下のように「Frontend」オブジェクトにフィールドを直接設定します。

Bean 定義ファイル

```
    @Bean
    Frontend frontend() {
        Frontend frontend = new Frontend();
        frontend.setCalculator(calculator());
        frontend.setArgumentResolver(argumentResolver());
        return frontend;
    }
```

「Spring Framework」に依存しないライブラリを作る場合、他のライブラリからそのライブラリを使う場合に「@Autowired」は使えないので、通常「セッター」か「コンストラクタ」でプロパティを設定できるようにします。

それらのライブラリに含まれるクラスを「Bean 定義」する場合は、この例のように「セッター」を用いて定義します。

また、「オート・ワイヤリング」対象のフィールドの型がインターフェイスである必要もありません。

*

アプリケーションの主処理は「Frontend」クラスに集約したので、「HajibootDiApplication」クラスは以下のように、「Frontend」を実行するだけになります。

HajibootDiApplication クラス

```
package com.example;

import com.example.app.Frontend;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Import;

@EnableAutoConfiguration
@Import(AppConfig.class)
public class HajibootDiApplication {

  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(HajibootDiAppl
ication.class, args);
    Frontend frontend = context.getBean(Frontend.class);
    frontend.run();
  }
}
```

「アプリケーション」の「エントリ・ポイント」を、すっきりさせることができました。

[2.1.4] 「コンポーネント・スキャン」で自動 Bean 登録

次に出てくる課題は、「DI コンテナに登録したい Bean を 1 つ 1 つ定義するのは手間がかかる」という点です。

この課題に対して、「Spring Framework」は、自動で「Bean」を「DI コンテナ」に登録する、「コンポーネント・スキャン」という仕組みをもっています。

次のように「HajibootDiApplication クラス」にも「@ComponentScan」アノテーションを付けることで、そのクラスのパッケージ以下を走査して、「@Component」など特定の「アノテーション」が付いている「Java クラス」を検出し、それらを「DI コンテナ」に登録します。

HajibootDiApplication クラス

```
package com.example;

import com.example.app.Frontend;
import org.springframework.boot.SpringApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Import;

@EnableAutoConfiguration
@ComponentScan // (1)
public class HajibootDiApplication {

  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(HajibootDiAppl
ication.class, args);
    Frontend frontend = context.getBean(Frontend.class);
    frontend.run();
  }
}
```

プログラム解説

項 番	説 明
(1)	「@ComponentScan」を付けることで、このクラスと同じパッケージ以下のクラスを走査。 対象のパッケージを変更する場合は、「basePackages 属性」で指定。

次に、「DI コンテナ」に登録したいクラス(「AddCalculator」「ScannerArgument Resolver」「Frontend」)に「@Component」アノテーションを付けます。

AddCalculator クラス

```
package com.example.app;

import org.springframework.stereotype.Component;

@Component
public class AddCalculator implements Calculator {
    @Override
    public int calc(int a, int b) {
        return a + b;
    }
}
```

ScannerArgumentResolver クラス

```
package com.example.app;

import org.springframework.stereotype.Component;

import java.io.InputStream;
import java.util.Scanner;

@Component
public class ScannerArgumentResolver implements ArgumentResolver {
    @Override
    public Argument resolve(InputStream stream) {
        Scanner scanner = new Scanner(stream);
        int a = scanner.nextInt();
        int b = scanner.nextInt();
        return new Argument(a, b);
    }
}
```

Frontend クラス

```
package com.example.app;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Frontend {
    @Autowired
    ArgumentResolver argumentResolver;
    @Autowired
    Calculator calculator;

    public void run() {
        System.out.print("Enter 2 numbers like 'a b' : ");
        Argument argument = argumentResolver.resolve(System.in);
```



```
        int result = calculator.calc(argument.getA(), argument.getB());
        System.out.println("result = " + result);
    }
}
```

*

以上により、「AppConfig クラス」は不要になりました。

「AppConfig クラス」を削除して「HajibootDiApplication クラス」がこれまで通り動作することを確認してください。

Spring Boot 1.2 からは

- @EnableAutoConfiguration」
- @Configuration
- @ComponentScan

の 3 つを合成した「@SpringBootApplication」アノテーションが導入されました。通常はこちらを使用します。

前述の「HajibootDiApplication クラス」を「@SpringBootApplication」アノテーションを用いて書き直すと次のようになります。今後はこのアノテーションを使用します。

HajibootDiApplication クラス

```
package com.example;

import com.example.app.Frontend;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class HajibootDiApplication {

  public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(HajibootDiAppl
ication.class, args);
    Frontend frontend = context.getBean(Frontend.class);
    frontend.run();
  }
}
```

[2.1.5] 「CommandLineRunner」の利用

アプリケーションで使う「クラス」を「DI コンテナ」から取得するコードから、「DI コンテナ」に「インジェクション」するコードに修正するために「Frontend クラス」を作り、「HajibootDiApplication クラス」からは「Frontend クラス」を呼び出すようにしました。

実は「Spring Boot」には「Frontend クラス」相当の処理を「HajibootDiApplication クラス」ができるように「CommandLineRunner」というインターフェイスが用意されています。

このインターフェイスを「HajibootDiApplication クラス」が実装することで、

「HajibootDiApplication クラス」にも「DI コンテナ」がインジェクションできるようになります。

HajibootDiApplication クラス

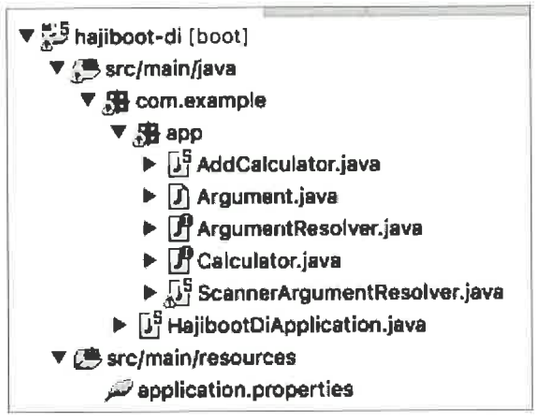
```
package com.example;

import com.example.app.Argument;
import com.example.app.ArgumentResolver;
import com.example.app.Calculator;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HajibootDiApplication implements CommandLineRunner /* (1) */
 {
  @Autowired // (2)
  ArgumentResolver argumentResolver;
  @Autowired // (2)
  Calculator calculator;

  @Override
  public void run(String... strings) throws Exception { // (1)
    System.out.print("Enter 2 numbers like 'a b' : ");
    Argument argument = argumentResolver.resolve(System.in);
    int result = calculator.calc(argument.getA(), argument.getB());
    System.out.println("result = " + result);
  }

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

プログラム解説

項　番	説　明
(1)	「CommandLineRunner」インターフェイスの「run」メソッドが、先ほど作った「Frontend クラス」の「run」メソッド相当の役割を担う。
(2)	「CommandLineRunner」インターフェイスをもつクラスは「Dependency Injection」が可能。

これで「Frontend クラス」は不要になりました。
本章のこの後の説明では、この「CommandLineRunner」を使います。

最終的なプロジェクト構成は以下のようになりました。



最終的なプロジェクト構成

「計算アプリケーション」としてはかなり大袈裟で重厚なつくりになっていますが、アプリケーションの規模が大きくなるにつれ、ここで説明した考え方は有効になってきます。

[2.1.6] 「レイヤー化」した「コンポーネント」の「インジェクション」

「Spring Framework」では「@Component」以外にも「コンポーネント・スキャン」の対象にするためのアノテーションが用意されています。クラスの責務に応じて使い分けるといいでしょう。

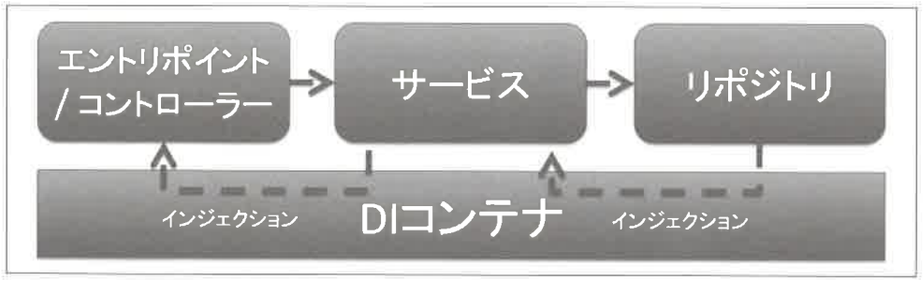
「コンポーネント・スキャン」の対象にするためのアノテーション

アノテーション	説　明
@Controller	「Web MVC フレームワーク」である「Spring MVC」の、「コントローラ」であることを示すアノテーション。 「Spring4」からは、「REST Web サービス」用に、「@RestController」も追加された。
@Service	「サービス・クラス」であることを示すアノテーション。 「@Component」と機能的な違いはない。
@Repository	「リポジトリ・クラス」であることを示すアノテーション。 このアノテーションを付けたクラス内で発生した例外は、特例のルールに従って、「Spring」が提供する「DataAccessException」に変換される。

「Bean 定義ファイル」の「JavaConfig クラス」に付けた「@Configuration」も「コンポーネント・スキャン」の対象になります。

ノート 「リポジトリ」は、もともと「Domain-Driven Design (Eric Evans, 2003)」で定義されており、「ドメインオブジェクトの保存、取得、検索」といった操作をカプセル化し、「コレクション・オブジェクト」のように振る舞う役割をもちます。
ロジックを「リポジトリ」に含めるべきではありません。

「Spring Framework」を用いたアプリケーションでは、次の図のようにレイヤーごとに「クラス」を作り、「DI コンテナ」にインジェクションさせる構成にすることが多いです。



アプリケーションの構成

伝統的なアプリケーションでは、エンティティなど「ドメイン・オブジェクト」のコレクションである「リポジトリ・クラス」を「サービス・クラス」が使ってロジックを組み立て、「エントリ・ポイント」(Web アプリケーションではコントローラ) を通じてユーザーからのリクエストに応じて「サービス・クラス」を呼び出します。
依存性の解決は「DI コンテナ」に任せます。

ここでは、単純化した「顧客管理システム」をイメージして、「リポジトリ」「サービス」を作ってアプリケーションを構築してみましょう。

Spring Initializr で再度プロジェクトを生成しましょう。

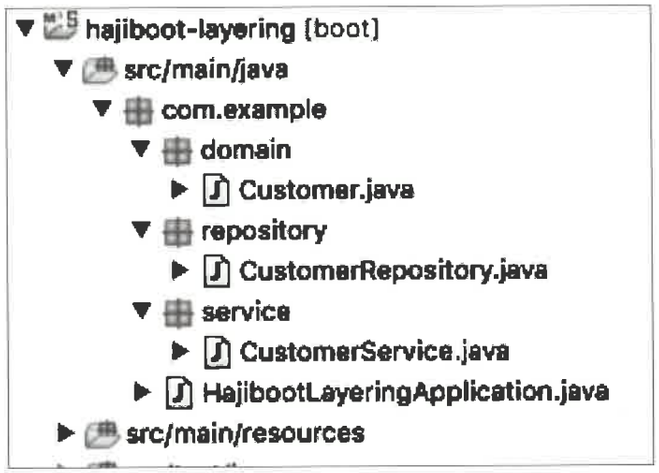
今回は次の項目を入力してください。

Artifact	Search for dependencies
hajiboot-layering	Lombok

「Generate Project」をクリックし、「hajiboot-layering.zip」をダウンロードしてください。
hajiboot-layering,zip を展開し、STS へインポートしてください。

*

最終的には次のようなプロジェクト構成になります。



最終的なプロジェクト構成

まずは「ドメイン・オブジェクト」として「Customer」クラスを作ります。

Customer クラス

```
package com.example.domain;

import java.io.Serializable;
import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor // (1)
public class Customer implements Serializable {
    private Integer id;
    private String firstName;
    private String lastName;
}
```

プログラム解説

項 番	説 明
(1)	「Lombok」の「@AllArgsConstructor」アノテーションで、全フィールドを引数にもつコンストラクタを生成させる。

*

次に、「Customer クラス」に対応する「リポジトリ・クラス」を作ります。

CustomerRepository クラス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
```

```
import java.util.concurrent.ConcurrentMap;

@Repository
public class CustomerRepository {
     private final ConcurrentMap<Integer, Customer> customerMap = new
ConcurrentHashMap<>();

    public List<Customer> findAll() {
        return new ArrayList<>(customerMap.values());
    }

    public Customer findOne(Integer customerId) {
        return customerMap.get(customerId);
    }

    public Customer save(Customer customer) {
        return customerMap.put(customer.getId(), customer);
    }

    public void delete(Integer customerId) {
        customerMap.remove(customerId);
    }
}
```

*

この「リポジトリ・クラス」を利用して「サービス・クラス」を構築します。

今回は「Customer」オブジェクトの登録と全件取得を行なうメソッドのみ作ります。

CustomerService クラス

```
package com.example.service;

import com.example.domain.Customer;
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class CustomerService {
    @Autowired
    CustomerRepository customerRepository;

    public Customer save(Customer customer) {
        return customerRepository.save(customer);
    }

    public List<Customer> findAll() {
        return customerRepository.findAll();
    }

    // その他のメソッドは次章で定義します
}
```

*

「サービス・クラス」をアプリケーションのエントリ・ポイントである「HajibootLayering Application クラス」にインジェクションして、簡単なアプリケーションを作りましょう。

HajibootLayeringApplication クラス

```
package com.example;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HajibootLayeringApplication implements CommandLineRunner {
  @Autowired
  CustomerService customerService;

  @Override
  public void run(String... strings) throws Exception {
    // データ追加
    customerService.save(new Customer(1, "Nobita", "Nobi"));
    customerService.save(new Customer(2, "Takeshi", "Goda"));
    customerService.save(new Customer(3, "Suneo", "Honekawa"));

    // データ表示
    customerService.findAll()
        .forEach(System.out::println);
  }

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

● 実 行

実行すると、

【コマンド・プロンプト】実行結果

```
Customer(id=1, firstName=Nobita, lastName=Nobi)
Customer(id=2, firstName=Takeshi, lastName=Goda)
Customer(id=3, firstName=Suneo, lastName=Honekawa)
```

が出力されます。

2.2 「Spring JDBC」による DB アクセス

次に、「データベース・アクセス」について説明します。

「Spring Framework」には、標準で「Spring JDBC」というモジュールが用意されています。

この「Spring JDBC」には、「JDBC API」を簡単に使うための「JdbcTemplate」というクラスが含まれています。

このクラスは、「JDBC」の煩雑な処理が簡潔に記述できるため、DB へのちょっとしたアクセスに向いています。

*

「Spring Initializr」で再度プロジェクトを生成しましょう。
今回は次の項目を入力してください。

「Spring Initializr」の入力項目

Artifact	Search for dependencies
hajiboot-jdbc	JDBC, H2, Lombok※

※1項目ずつ、入力してクリック

「Generate Project」をクリックし、ダウンロードした「hajiboot-jdbc.zip」を展開してください。
展開されたフォルダに存在する、「pom.xml」を確認しましょう。

【pom.xml】「依存ライブラリ」の定義

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>hajiboot-jdbc</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hajiboot-jdbc</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <!-- (1) -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
    </dependency>
    <!-- (2) -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```


プログラム解説

項　番	説　明
(1)	「spring-boot-starter-jdbc」を依存関係に追加することで、「Spring JDBC」によるDBアクセスに必要なライブラリが追加される。
(2)	今回は、「H2データベース」を使う。

＊

「hajiboot-jdbc」プロジェクトを「STS」にインポートしてください。

コラム　H2データベース

「H2データベース」はJavaで実装されたリレーショナルデータベースシステム（RDBMS）です。
組み込みモードをもつため、特別なインストール作業を行なうことなく利用できます。
本書では、アプリケーションの解説に注力するため、基本的にH2データベースを使います。

[2.2.1] 「JdbcTemplate」を使ったDBアクセス

ここでは「JdbcTemplate」をラップした、より便利な「NamedParameterJdbcTemplate」というクラスを使います。

「NamedParameterJdbcTemplate」では、SQL文中にパラメータを埋め込むため、「:パラメータ名」形式のプレースホルダを利用します。通常の「JDBC API」は、「?」をプレースホルダに使うので不便です。扱いやすいプレースホルダを使うことで、「SQLインジェクション」の対策漏れを防ぎやすくなります。

●「JdbcTemplate」を使った簡単なデータアクセス

まずは簡単な使い方を説明します。

HajibootJdbcApplication クラス

```
package com.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

@SpringBootApplication
public class HajibootJdbcApplication implements CommandLineRunner {

  @Autowired
  NamedParameterJdbcTemplate jdbcTemplate; // (1)

  @Override
  public void run(String... args) throws Exception {
    String sql = "SELECT 1"; // (2)
    SqlParameterSource param = new MapSqlParameterSource(); // (3)
    Integer result = jdbcTemplate.queryForObject(sql, param, Integer.
class); // (4)

    System.out.println("result = " + result);
  }

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

プログラム解説

項 番	説 明
(1)	「DI コンテナ」に登録された「NamedParameterJdbcTemplate」オブジェクトを取得。 「Spring Boot」では「autoconfigure」という仕組み※で「DataSource」や「JdbcTemplate」「NamedParameterJdbcTemplate」が自動で生成され、「DI コンテナ」に登録される。 したがって、「Spring Boot」では、「依存ライブラリ」に「spring-boot-jdbc」と「JDBC ドライバ」を追加するだけで特に設定することなく、「JdbcTemplate」が使用できる。 今回は、「pom.xml」に「JDBC ドライバ」として「H2 データベース」への「依存関係」を定義したので、「H2 データベース」用の「DataSource」が生成される。 「H2 データベース」を使って、データベースの URL などの指定がない場合は、デフォルトで「インメモリの組み込みデータベース」が作られる。 データベースのセットアップの手間がかからないため、動作確認用としては便利。
(2)	簡単な SQL を使う。 この文法ではエラーになる DBMS もあるが、「H2」の場合はそのまま「1」が返却される。
(3)	「SqlParameterSource」クラスで「SQL」に埋め込むパラメータを作る。 今回の例ではパラメータを使わないため、「空オブジェクト」を利用する。 次の例から実際にパラメータを使用。 「SqlParameterSource」の代わりに「Map<String, Object>」を使うこともできるが、「SqlParameterSource」のほうが便利であるため、本書ではこちらを使って説明する。
(4)	「NamedParameterJdbcTemplate」の「queryForObjcet」メソッドを使ってクエリの実行結果をオブジェクトに変換して取得。 「第一引数」に「SQL」、「第二引数」に「パラメータ」、「第三引数」に「返り値となるオブジェクトのクラス」を指定。 このメソッドは、「クエリの返り値」が 1 件でない場合は、「IncorrectResultSize DataAccessException」がスローされる。

※「autoconfigure」の仕組みについては、筆者の発表資料を参照してください。
http://www.slideshare.net/makingx/spring-boot-java-jsug

実 行

【コマンド・プロンプト】実行結果

```
result = 1
```

という結果が出力されたでしょうか。

＊

次に、パラメータを指定してみましょう。

HajibootJdbcApplication クラス

```
package com.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

@SpringBootApplication
public class HajibootJdbcApplication implements CommandLineRunner {
  @Autowired
  NamedParameterJdbcTemplate jdbcTemplate;

  @Override
  public void run(String... strings) throws Exception {
    String sql = "SELECT :a + :b"; // (1)
    SqlParameterSource param = new MapSqlParameterSource()
        .addValue("a", 100)
        .addValue("b", 200); // (2)
```

```
    Integer result = jdbcTemplate.queryForObject(sql, param, Integer.class);

    System.out.println("result = " + result);
  }
  public static void main(String[] args) {
    SpringApplication.run(HajibootJdbcApplication.class, args);
  }
}
```

プログラム解説

項　番	説　明
(1)	SQL 内に「:a」と「:b」という「プレース・ホルダ」を埋め込んだ。
(2)	「MapSqlParameterSource」の「addValue」メソッドで (1) の「a」と「b」というパラメータに対して、値を設定。

実 行

実行すると、

【コマンド・プロンプト】実行結果

```
result = 300
```

という結果が出力されたと思います。

●「JdbcTemplate」を使った「オブジェクト・マッピング」

次に、SQL の実行結果を Java オブジェクトにマッピングする方法を説明します。

先ほどの「Customer」クラスを利用して、SQL の結果を「Customer」オブジェクトにマッピングします。

コードを書き換えます。

クエリの結果をオブジェクトにマッピングするため、「RowMapper」を「匿名クラス」で実装します。

HajibootJdbcApplication クラス

```
package com.example;

import com.example.domain.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

import java.sql.ResultSet;
import java.sql.SQLException;
```



```
@SpringBootApplication
public class HajibootJdbcApplication implements CommandLineRunner{
  @Autowired
  NamedParameterJdbcTemplate jdbcTemplate;

  @Override
  public void run(String... strings) throws Exception {
    String sql = "SELECT id, first_name, last_name FROM customers WHE
RE id = :id"; // (1)
    SqlParameterSource param = new MapSqlParameterSource()
        .addValue("id", 1); // (2)
    Customer result = jdbcTemplate.queryForObject(sql, param, new Row
Mapper<Customer>() {
      @Override
      public Customer mapRow(ResultSet rs, int rowNum) throws SQLExce
ption { // (3)
        return new Customer(rs.getInt("id"), rs.getString("first_name"), rs
.getString("last_name"));
      }
    });

    System.out.println("result = " + result);
  }
  public static void main(String[] args) {
    SpringApplication.run(HajibootJdbcApplication.class, args);
  }
}
```

プログラム解説

項　番	説　明
(1)	「customers」テーブルから「主キー」を指定して情報を取得する SQL を記述。
(2)	パラメータとして取得対象の「主キー」を設定。
(3)	「ResultSet」から「Customer」オブジェクトを生成する「RowMapper<Customer>」を実装。

このままでは DB にテーブルもデータも何も登録されていないため、実行時に「エラー」になってしまいます。

「Spring Boot」では、「クラス・パス」直下に以下の SQL ファイルが存在した場合に、それらを読み込んで実行されます。

- schema-(platform).sql [3]
- schema.sql
- data-(platform).sql
- data.sql

ここでは、「src/main/resources/schema.sql」に今回使う「DDL」を、「src/main/resources/data.sql」に「初期データ」を記述します。

[3] (platform) はデフォルトで「all」です。起動時に「--spring.datasource.platform=xxx」を指定することで変更することができます。環境ごとに実行する SQL を変えたい場合に便利です。

【src/main/resources/schema.sql】今回使う「DDL」

```
CREATE TABLE customers (id INT PRIMARY KEY AUTO_INCREMENT, first_name
VARCHAR(30), last_name VARCHAR(30));
```

【src/main/resources/data.sql】初期データ

```
INSERT INTO customers(first_name, last_name) VALUES('Nobita', 'Nobi');
INSERT INTO customers(first_name, last_name) VALUES('Takeshi', 'Goda');
INSERT INTO customers(first_name, last_name) VALUES('Suneo', 'Honekawa');
INSERT INTO customers(first_name, last_name) VALUES('Shizuka', 'Minamoto');
```

実 行

この2ファイルを作った後、アプリケーションを実行しましょう。

【コマンド・プロンプト】実行結果

```
result = Customer(id=1, firstName=Nobita, lastName=Nobi)
```

という結果が出力されたでしょうか。

*

ところで、「RowMapper<Customer>」を「匿名クラス」で実装するのは、少し煩わしいですね。ここは「Java SE 8」から導入された「ラムダ式」を使うことができます。

HajibootJdbcApplication クラス

```
package com.example;

import com.example.domain.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;

@SpringBootApplication
public class HajibootJdbcApplication implements CommandLineRunner {
  @Autowired
  NamedParameterJdbcTemplate jdbcTemplate;

  @Override
  public void run(String... strings) throws Exception {
    String sql = "SELECT id, first_name, last_name FROM customers WHE
RE id = :id";
    SqlParameterSource param = new MapSqlParameterSource()
        .addValue("id", 1);
    Customer result = jdbcTemplate.queryForObject(sql, param,
        (rs, rowNum) -> new Customer(rs.getInt("id"), rs.getString("fi
rst_name"), rs.getString("last_name"))
    );

    System.out.println("result = " + result);
  }
```



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

プログラム解説

項 番	説 明
(1)	「RowMapper<Customer>」の「匿名クラス」の代わりに、「(引数) -> 返り値」形式の「ラムダ式」を用いた。

「Spring Framework」には、引数に「匿名クラス」をもつ、「○○ Template」というクラスが多く用意されており、「ラムダ」を使うと簡易に記述できます。

ノート SQL ファイルに「日本語」を使いたい場合は、後に説明する「src/main/resources/application.properties」に、以下のようにファイルの文字コードを指定してください。

【src/main/resources/application.properties】ファイルの文字コードを指定

```
spring.datasource.sql-script-encoding=UTF-8
```

[2.2.2] 「データ・ソース」の設定を明示的に変更

ここまで「データ・ソース」の設定(JDBC の設定)は、自動で行なわれるものを使いました。

「H2」データベースの「JDBC」ドライバを依存関係に追加した場合、デフォルトでは「組み込みインメモリ DB」が使われます。そのため、「JDBC」の設定を変更したいときには、設定ファイルに明示的に設定を記述する必要があります。

「Spring Boot」では、「プロパティ」の設定は、「クラス・パス」直下の「application.properties」(Properties ファイル)または「application.yml」(YAML ファイル)に記述します。本書では、Spring Initializr で作られる、「application.properties」を使います。

*

まずは、これまで通り、「組み込みインメモリ DB」を使う場合の、明示的な設定をします。

「src/main/resources/application.properties」に、以下の設定をしてください。

【src/main/resources/application.properties】「組み込みインメモリ DB」を使う設定

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_
ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
```

この設定では、アプリケーションが終了するたびに、データはすべて消えてしまいます。

データを永続化するために、「H2」データベースの「ファイル・データベース」を使うように設定を変更してみましょう。

以下のように「spring.datasource.url」プロパティの値を変更します。

【src/main/resources/application.properties】「H2」データベースを使う設定

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./target/db/testdb
spring.datasource.username=sa
spring.datasource.password=
```

この状態で「HajibootJdbcApplication.java」を実行すると実行フォルダ直下の「target/db/testdb.h2.db」というファイルが作られ、ここにデータベースの情報が永続化されます。

ノート ファイル・データベースを使って、「HajibootJdbcApplication.java」を二回実行すると、「sch ema.sql」の「CREATE table」を実行する際に、すでにテーブルが存在するため、エラーが発生します。
そこで、以下のようにSQLを修正する必要があります。

```
CREATE table IF NOT EXISTS customers(id int primary key auto-increm
ent, first_name varchar(30), last_name varchar(30));
```

「data.sql」は複数回実行される点に注意してください。

ここで扱ったプロパティ以外にも設定可能なプロパティは多数存在します。それについては公式ドキュメント[4]を参照してください。

本書では基本的にこれまで通り、「組み込みインメモリDB」を使います。

次章の「**[3.4]「Flyway」で「DBマイグレーション」**」以降では、ファイルに保存するDBを使います。

[2.2.3] 「Log4JDBC」を使った「SQLログ出力」

「Spring Boot」の話から少し脱線してしまいますが、今後の開発を容易にするために、実行中の「SQLログ」を出力する方法を説明します。

*

「SQLログ」を出力する方法として、『「Log4JDBC」を使って「JDBCドライバ」に「プロキシ」をラップする方法』を説明します。

*

「pom.xml」に以下の設定を追加してください。

[4] http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

●「pom.xml」に「依存関係」を追加

「Log4JDBC」を依存関係に追加するため、「pom.xml」に、以下の設定を追加します。

【pom.xml】「Log4JDBC」の「依存関係」の追加

```
<dependency>
    <groupId>org.lazyluke</groupId>
    <artifactId>log4jdbc-remix</artifactId>
    <version>0.2.7</version>
</dependency>
```

●「Log4JDBC」用の「JDBCドライバ」の設定

「Log4JDBC」を使うために、ドライバの「クラス名」と「url」を変更します。

ドライバの「クラス名」には「net.sf.log4jdbc.DriverSpy」を設定してください。

実際に使われる「JDBCドライバ」(今回の場合はH2)は自動的に認識されて、ラップされます。

ドライバの「url」には、通常設定する値から、「jdbc:」の後に「log4jdbc:」を追加したものを設定します。

*

「src/main/resources/application.properties」に以下のプロパティを設定してください。

```
spring.datasource.driver-class-name=net.sf.log4jdbc.DriverSpy
spring.datasource.url=jdbc:log4jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;
DB_CLOSE_ON_EXIT=FALSE
```

●「ログ・レベル」の設定

「Log4JDBC」による「SQLログ」が出力されるように、「logback」のレベルの設定をします。

ログのレベルも「src/main/resources/application.properties」に「logging.level.<ロガー名>=<ログ・レベル>」というプロパティを設定することで、変更可能です。

デフォルトは「INFOレベル」です。

*

「src/main/resources/application.properties」に、以下のプロパティを追加してください。

```
logging.level.jdbc=OFF # (1)
logging.level.jdbc.sqltiming=DEBUG # (2)
```

プログラム解説

項　番	説　明
(1)	「Log4JDBC」は、「jdbc」から始めるいろいろなログを出力するが、ここでは「SQL」のログだけ出力したいため、それ以外は出力しないように、「OFF」を設定。
(2)	「SQL ログ」は「デバッグ・レベル」で出力されるため、「DEBUG」を設定。「jdbc.sqltiming」のログは、「SQL 文 + 実行時間」をログに出力。

実 行

「HajibootJdbcApplication」を実行してみましょう。

以下のような「SQL ログ」が出力されます。

［ターミナル］出力された「SQL ログ」

```
(略)
2016-08-05 10:40:55.766 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.datasource.
init.ScriptUtils.executeSqlScript(ScriptUtils.java:473)
10. CREATE table customers(id int primary key auto_increment, first_
name varchar(30), last_name
varchar(30))  {executed in 31 msec}
(略)
2016-08-05 10:40:55.773 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.datasource.
init.ScriptUtils.executeSqlScript(ScriptUtils.java:473)
10. INSERT INTO customers(first_name, last_name) VALUES('Nobita',
 'Nobi')  {executed in 3 msec}
2016-08-05 10:40:55.774 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.datasource.
init.ScriptUtils.executeSqlScript(ScriptUtils.java:473)
10. INSERT INTO customers(first_name, last_name) VALUES('Takeshi',
'Goda')  {executed in 0 msec}
2016-08-05 10:40:55.775 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.datasource.
init.ScriptUtils.executeSqlScript(ScriptUtils.java:473)
10. INSERT INTO customers(first_name, last_name) VALUES('Suneo', 'Hon
ekawa')  {executed in 0 msec}
2016-08-05 10:40:55.776 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.datasource.
init.ScriptUtils.executeSqlScript(ScriptUtils.java:473)
10. INSERT INTO customers(first_name, last_name) VALUES('Shizuka',
 'Minamoto')  {executed in 1 msec}
(略)
2016-08-05 10:40:56.131 DEBUG 25721 --- [           main] jdbc.sqltim
ing                          :  org.springframework.jdbc.core.JdbcTe
mplate$1.doInPreparedStatement(JdbcTemplate.java:692)
10. SELECT id, first_name, last_name FROM customers WHERE id = 1  {ex
ecuted in 5 msec}
result = Customer(id=1, firstName=Nobita, lastName=Nobi)
```

「前半」は「SQL スクリプト」の結果で、「後半」は「JdbcTemplate」の実行結果です。

本書では、今後は基本的にこの「SQL ログ」の設定を行なうことを前提とします。

ノート　本書で利用する「log4jdbc-remix」は開発が終了しており、後継のライブラリは「log4jdbc-log4j2」[5] です。

「log4jdbc-log4j2」と「Spring Boot」のデフォルト・ロギング・ライブラリである「Logback」を一緒に使う際に、設定ファイルが 1 つ増えてしまうため、本書では「log4jdbc-remix」を使っています。

[2.2.4]　「JdbcTemplate」を使った「リポジトリ・クラス」の実装

先ほど作った「CustomerRepository」クラスを「NamedParameterJdbcTemplate」を使って書き直します。

「Customer」オブジェクトの格納先は、「メモリ」から「DB」になります。

これまでは「NamedParameterJdbcTemplate」を使って、DB の参照しかしてきませんでしたが、「CustomerRepository」クラスの実装を通じて、DB の更新方法も説明します。

CustomerRepository クラス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParame
terSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Repository
@Transactional // (1)
public class CustomerRepository {
    @Autowired
    NamedParameterJdbcTemplate jdbcTemplate;

    // (2)
    private static final RowMapper<Customer> customerRowMapper = (rs, i) -> {
        Integer id = rs.getInt("id");
        String firstName = rs.getString("first_name");
        String lastName = rs.getString("last_name");
        return new Customer(id, firstName, lastName);
    };

    public List<Customer> findAll() {
        List<Customer> customers = jdbcTemplate.query(
                "SELECT id,first_name,last_name FROM customers ORDER BY id",
                customerRowMapper); // (2)
        return customers;
    }
```

[5] https://github.com/brunorozendo/log4jdbc-log4j2

```
    public Customer findOne(Integer id) {
        SqlParameterSource param = new MapSqlParameterSource().addVal
ue("id", id);
        return jdbcTemplate.queryForObject(
                "SELECT id,first_name,last_name FROM customers WHERE
id=:id",
                param,
                customerRowMapper);
    }

    public Customer save(Customer customer) {
        SqlParameterSource param = new BeanPropertySqlParameterSource
(customer); // (3)
        // (4)
        if (customer.getId() == null) {
            jdbcTemplate.update("INSERT INTO customers(first_name,la
st_name) values(:firstName, :lastName)",
                    param);
        } else {
            jdbcTemplate.update("UPDATE customers SET first_name=:fir
stName, last_name=:lastName WHERE id=:id",
                    param);
        }
        return customer;
    }

    public void delete(Integer id) {
        SqlParameterSource param = new MapSqlParameterSource().addVal
ue("id", id);
        jdbcTemplate.update("DELETE FROM customers WHERE id=:id",
                param); // (5)
    }
}
```

プログラム解説

項　番	説　明
(1)	「@Transactional」アノテーションが「クラス・レベル」に付いたクラスを「DI コンテナ」から取得すると、そのクラスの各「メソッド」が他のクラスから呼ばれた場合に、自動的に「DB トランザクション」の制御が行なわれる。 (a)「メソッドが正常終了した場合」は、「DB トランザクション」がコミットされ、(b)「実行時例外が発生した場合※」は、「DB トランザクション」がロールバックされる。 「DI コンテナ」によって、各メソッドに前後処理が追加されたクラスが、動的に生成される。
(2)	「NamedParameterJdbcTemplate」の「query」メソッドを用いて、SQL の実行結果を「Java オブジェクト」の「リスト」として取得。 「ResultSet」を「Java オブジェクト」に変換するための「RowMapper」は、次のメソッドでも使うため、フィールドに持たせる。
(3)	更新用の「SqlParameterSource」を作る。 「BeanPropertySqlParameterSource」を使うことで、「Java オブジェクト」の「フィールド名」と「値」をマッピングした「SqlParameterSource」が自動的に作成できる。

(4)	更新系の SQL を実行するために「NamedParameterJdbcTemplate」の「update」メソッドを使う。 「CustomerRepository」の「save」では、(a) 引数の「Customer オブジェクト」の「id」が「null」の場合に「INSERT 文」を実行し、(b) そうでない場合は「UPDATE 文」を実行。
(5)	「DELETE 文」を実行する場合も、「update」メソッドを使う。

※チェック例外の場合は、ロールバックされません（重要）。

この「CustomerRepository」を使って、「HajibootJdbcApplication クラス」を修正しましょう。

HajibootJdbcApplication クラス

```
package com.example;

import com.example.domain.Customer;
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HajibootJdbcApplication implements CommandLineRunner{
  @Autowired
  CustomerRepository customerRepository;

  @Override
  public void run(String... strings) throws Exception {
    // データ追加
    Customer created = customerRepository.save(new Customer(null, "Hi
detoshi", "Dekisugi"));
    System.out.println(created + " is created!");
    // データ表示
    customerRepository.findAll()
        .forEach(System.out::println); //(1)
  }

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

プログラム解説

項　番	説　明
(1)	「Java SE 8」から「java.lang.Iterable」インターフェイスに追加された「forEach」メソッドを使った。 リストの各データに対して、「forEach」メソッドの引数に渡された「ラムダ式」を適用する。 「System.out::println」は「メソッド参照」と呼ばれ、「(x) -> {System.out.println(x);}」の省略型。

実 行

実行すると、以下のように出力されます。

【ターミナル】実行結果

```
(略)
Customer(id=null, firstName=Hidetoshi, lastName=Dekisugi) is created!
(略)
Customer(id=1, firstName=Nobita, lastName=Nobi)
Customer(id=2, firstName=Takeshi, lastName=Goda)
Customer(id=3, firstName=Suneo, lastName=Honekawa)
Customer(id=4, firstName=Shizuka, lastName=Minamoto)
Customer(id=5, firstName=Hidetoshi, lastName=Dekisugi)
```

ノート 細かいですが、今の「CustomerRepository」の実装だと、「save」メソッドで「Customer」オブジェクトを新規作成した場合に、「DB 側」で生成した「ID」が設定されません。

「リポジトリ」の機能としては、新規作成後に ID を設定するのが望ましいです。

「JdbcTemplate」でこの機能を実装するには、「SimpleJdbcInsert」を使うと簡単です。

以下のように「CustomerRepository」を修正しましょう。

CustomerRepository クラス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlPara
meterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.PostConstruct;
import java.util.List;

@Repository
@Transactional
public class CustomerRepository {
    @Autowired
    NamedParameterJdbcTemplate jdbcTemplate;

    SimpleJdbcInsert insert;

    @PostConstruct
    public void init() {
        insert = new SimpleJdbcInsert((JdbcTemplate) jdbcTemplate.
getJdbcOperations()) // (1)
                .withTableName("customers") // (2)
                .usingGeneratedKeyColumns("id"); // (3)
    }

    // (略)

    public Customer save(Customer customer) {
        SqlParameterSource param = new BeanPropertySqlParameterSour
ce(customer);
        if (customer.getId() == null) {
            // ここから変更
            Number key = insert.executeAndReturnKey(param); // (4)
            customer.setId(key.intValue());
            // ここまで
        } else {
            // (略)
        }
        return customer;
    }
    // (略)
}
```

プログラム解説

項 番	説 明
(1)	「SimpleJdbcInsert」には「JdbcTemplate」を設定する必要があるため、「NamedJdbcTemplate」に内包されている「JdbcTemplate」オブジェクトを取り出す。
(2)	「SimpleJdbcInsert」は「INSERT」の SQL を自動生成するので、「テーブル名」を明示的に指定。
(3)	自動採番される「主キー」の「カラム名」を指定。
(4)	「executeAndReturnKey」メソッドを実行すると SQL を実行し、自動採番された「ID」が返却される。

「CustomerRepository」を修正後、「HajibootJdbcApplication クラス」を再実行すると「Customer(id=5, firstName=Hidetoshi, lastName=Dekisugi) is created!」と ID が設定された状態で出力されます。

2.3 「Spring Data JPA」による DB アクセス

次は「JPA (Java Persistence API)」による DB アクセスについて説明します。

*

「JPA」は Java 標準の「O/R マッパー」に関する仕様です。
実装ライブラリとして、「Hibernate」や「EclipseLink」が有名です。

*

「JPA」には以下のような特徴があります。

- 「Java オブジェクト」と「データベースに格納されているデータ」とのマッピング機能
- 「データベース」への「CRUD 処理」をカプセル化した「API」
- 「Java オブジェクト」を検索するための「クエリ言語」(JPQL)

データベース製品の差異も、「JPA」によって吸収されます。

「Spring Data JPA」は「Spring Framework」のサブプロジェクトで、「JPA」を用いたプログラミングをより簡単にするために開発されています。

「Spring Boot」では「Spring Data JPA」に関する依存関係を簡単に追加するための「spring-boot-starter-data-jpa」が用意されています。

*

「Spring Initializr」で、再度プロジェクトを生成しましょう。
今回は次の項目を入力してください。

「Spring Initializr」の入力項目

Artifact	Search for dependencies
hajiboot-jpa	JPA, H2, Lombok※

※1項目ずつ、入力してクリック

「Generate Project」をクリックし、ダウンロードした「hajiboot-jpa.zip」を展開してください。

展開されたフォルダに存在する「pom.xml」を確認すると、「spring-boot-starter-data-jpa」が設定されていることが分かります。

[2.2.3]で扱った「Log4JDBC」の設定も、追加してください。

[pom.xml]「依存ライブラリ」の定義

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>hajiboot-jpa</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hajiboot-jpa</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
```



```
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
    </dependency>

    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

実行

「./mvnw dependency:tree」を実行して、どのような「ライブラリ」が追加されているか、確認しましょう。

[ターミナル]「追加されたライブラリ」の確認

```
$ ./mvnw dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building hajiboot-jpa 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @ hajiboot-jpa ---
[INFO] com.example:hajiboot-jpa:jar:0.0.1-SNAPSHOT
[INFO] +- org.springframework.boot:spring-boot-starter-data-jpa:jar:1.4.0.RELEASE:compile
[INFO] |  +- org.springframework.boot:spring-boot-starter:jar:1.4.0.R
ELEASE:compile
```

```
[INFO] |  |  +- org.springframework.boot:spring-boot:jar:1.4.0.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot-autoconfigure:jar:1.4.
0.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot-starter-logging:
jar:1.4.0.RELEASE:compile
[INFO] |  |  |  +- ch.qos.logback:logback-classic:jar:1.1.7:compile
[INFO] |  |  |  |  \- ch.qos.logback:logback-core:jar:1.1.7:compile
[INFO] |  |  |  +- org.slf4j:jul-to-slf4j:jar:1.7.21:compile
[INFO] |  |  |  \- org.slf4j:log4j-over-slf4j:jar:1.7.21:compile
[INFO] |  |  \- org.yaml:snakeyaml:jar:1.17:runtime
[INFO] |  +- org.springframework.boot:spring-boot-starter-aop:jar:1.4.0.
RELEASE:compile
[INFO] |  |  +- org.springframework:spring-aop:jar:4.3.2.RELEASE:compile
[INFO] |  |  \- org.aspectj:aspectjweaver:jar:1.8.9:compile
[INFO] |  +- org.springframework.boot:spring-boot-starter-jdbc:jar:1.4.0.
RELEASE:compile
[INFO] |  |  +- org.apache.tomcat:tomcat-jdbc:jar:8.5.4:compile
[INFO] |  |  |  \- org.apache.tomcat:tomcat-juli:jar:8.5.4:compile
[INFO] |  |  \- org.springframework:spring-jdbc:jar:4.3.2.RELEASE:compile
[INFO] |  +- org.hibernate:hibernate-core:jar:5.0.9.Final:compile
[INFO] |  |  +- org.jboss.logging:jboss-logging:jar:3.3.0.Final:compile
[INFO] |  |  +- org.hibernate.javax.persistence:hibernate-jpa-2.1-
api:jar:1.0.0.Final:compile
[INFO] |  |  +- org.javassist:javassist:jar:3.20.0-GA:compile
[INFO] |  |  +- antlr:antlr:jar:2.7.7:compile
[INFO] |  |  +- org.jboss:jandex:jar:2.0.0.Final:compile
[INFO] |  |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  |  |  \- xml-apis:xml-apis:jar:1.4.01:compile
[INFO] |  |  \- org.hibernate.common:hibernate-commons-annotations:ja
r:5.0.1.Final:compile
[INFO] |  +- org.hibernate:hibernate-entitymanager:jar:5.0.9.Final:co
mpile
[INFO] |  +- javax.transaction:javax.transaction-api:jar:1.2:compile
[INFO] |  +- org.springframework.data:spring-data-jpa:jar:1.10.2.
RELEASE:compile
[INFO] |  |  +- org.springframework.data:spring-data-commons:jar:1.12
.2.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-orm:jar:4.3.2.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-context:jar:4.3.2.RELEASE:
compile
[INFO] |  |  |  \- org.springframework:spring-expression:jar:4.3.2.RE
LEASE:compile
[INFO] |  |  +- org.springframework:spring-tx:jar:4.3.2.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-beans:jar:4.3.2.RELEASE:compile
[INFO] |  |  +- org.slf4j:slf4j-api:jar:1.7.21:compile
[INFO] |  |  \- org.slf4j:jcl-over-slf4j:jar:1.7.21:compile
[INFO] |  \- org.springframework:spring-aspects:jar:4.3.2.RELEASE:compile
[INFO] +- org.projectlombok:lombok:jar:1.16.10:compile
[INFO] +- com.h2database:h2:jar:1.4.192:runtime
[INFO] \- org.springframework.boot:spring-boot-starter-test:jar:1.4.0.
RELEASE:test
[INFO]    +- org.springframework.boot:spring-boot-test:jar:1.4.0.RELEASE:test
[INFO]    +- org.springframework.boot:spring-boot-test-autoconfigure:
jar:1.4.0.RELEASE:test
[INFO]    +- com.jayway.jsonpath:json-path:jar:2.2.0:test
[INFO]    |  \- net.minidev:json-smart:jar:2.2.1:test
[INFO]    |     \- net.minidev:accessors-smart:jar:1.1:test
[INFO]    |        \- org.ow2.asm:asm:jar:5.0.3:test
[INFO]    +- junit:junit:jar:4.12:test
[INFO]    +- org.assertj:assertj-core:jar:2.5.0:test
[INFO]    +- org.mockito:mockito-core:jar:1.10.19:test
[INFO]    |  \- org.objenesis:objenesis:jar:2.1:test
[INFO]    +- org.hamcrest:hamcrest-core:jar:1.3:test
```



```
[INFO]    +- org.hamcrest:hamcrest-library:jar:1.3:test
[INFO]    +- org.skyscreamer:jsonassert:jar:1.3.0:test
[INFO]    |  \- org.json:json:jar:20140107:test
[INFO]    +- org.springframework:spring-core:jar:4.3.2.RELEASE:compile
[INFO]    \- org.springframework:spring-test:jar:4.3.2.RELEASE:test
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.496 s
[INFO] Finished at: 2016-08-07T17:28:12+09:00
[INFO] Final Memory: 21M/309M
[INFO] ------------------------------------------------------------------------
```

*

「Spring Data JPA」と、「JPA」の実装ライブラリとして「Hibernate」が使われていることが分かります。

[2.3.1] 「JPA」の「エンティティ・クラス」の作成

まずは「エンティティ・クラス」を作ります。

*

先ほどの「Customer クラス」を、「JPA」に合わせてカスタマイズします。

Customer クラス

```
package com.example.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity // (1)
@Table(name = "customers") // (2)
@Data
@NoArgsConstructor // (3)
@AllArgsConstructor // (4)
public class Customer {
    @Id // (5)
    @GeneratedValue // (6)
    private Integer id;
    @Column(nullable = false) // (7)
    private String firstName;
    @Column(nullable = false)
    private String lastName;
}
```

プログラム解説

項 番	説 明
(1)	「@Entity」アノテーションを付けて、「JPA」の「エンティティ」であることを示す。
(2)	「@Table」アノテーションを付けて、「エンティティ」に対応するテーブル名を指定。 デフォルトでは「テーブル名＝クラス名」になる。 この例では、「テーブル名」は「クラス名」から変えている。

(3)	「JPA」の仕様で、「エンティティ・クラス」には引数のない「デフォルト・コンストラクタ」を作る必要がある。 「Lombok」で「デフォルト・コンストラクタ」を生成するために「@NoArgsConstructor」アノテーションを付ける。
(4)	「JPA」とは関係ないが、プログラミングの際に便利であるため、「デフォルト・コンストラクタ」以外に、全フィールドを引数にもつコンストラクタも「Lombok」で生成させる。 「@AllArgsConstructor」アノテーションを付けることで生成可能。
(5)	「エンティティ」の「主キー」であるフィールドに、「@Id」アノテーションを付ける。
(6)	「主キー」が「DB」で自動採番されることを、「@GeneratedValue」アノテーションを付けて示す。
(7)	「フィールド」に「@Column」アノテーションを付けて、該当するDBのカラムに対する「名前」や「制約」など設定する。ここでは「not null」制約を設定。

本書では「JPA」に関する詳しい説明はしません。「JPA」に関する学習用には「パーフェクトJava EE（技術評論社）[6]」をお勧めします。

[2.3.2] 「Spring Data JPA」を使った「リポジトリ・クラス」の作成

「Spring Data」は、「データ・ストア」を操作するための汎用的な機能を提供する「サブ・プロジェクト」です。

「データ・ストア」として、「RDBMS」「MongoDB」「Redis」「Neo4J」など、さまざまなプロダクトがサポートされています。

汎用的なリポジトリクラスも提供されています。

今回は、「JPA」でRDBMSを操作する「Spring Data JPA」を利用します。

「Spring Data JPA」が提供する「JpaRepository」を使って、「CustomerRepository」を再作成してみます。

CustomerRepository クラス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.data.jpa.repository.JpaRepository;

public interface CustomerRepository extends JpaRepository<Customer,
Integer> {
}
```

「JpaRepository」には、

- findOne
- findAll
- save
- delete

といった「CRUD」（「Create」「Read」「Update」「Delete」）操作のための基本的

[6] http://gihyo.jp/book/2016/978-4-7741-8316-9

なメソッドが定義されており、「リポジトリ」の作成は「JpaRepository」を継承したインターフェイスを作るだけです。非常に簡単です。

「インターフェイス」されあれば実行時に「実行クラス」が生成されるので、冗長なプログラムを記述する必要がありません。

この「リポジトリ・クラス」を使って「Spring JDBC」の説明で作った「HajibootJdbcApplicationクラス」を実行できます。

ただし、今回作成したクラス名は「HajibootJpaApplication」になっているので、[2.2]で作成した「HajibootJdbcApplication」のファイル名とクラス名を「HajibootJpaApplication」にリネームして今回のプロジェクトにコピーしてください。また「application.properties」もコピーしてください。

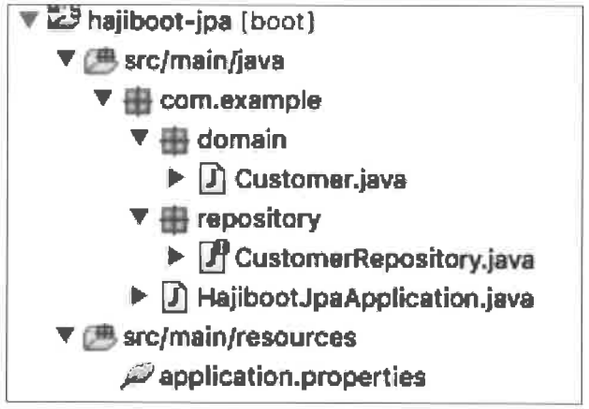
「HajibootJpaApplicationクラス」を再掲します。

HajibootJpaApplication クラス

```
package com.example;

import com.example.domain.Customer;
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HajibootJpaApplication implements CommandLineRunner {
  @Autowired
  CustomerRepository customerRepository;

  @Override
  public void run(String... strings) throws Exception {
    // データ追加
    Customer created = customerRepository.save(new Customer(null, "Hi
detoshi", "Dekisugi"));
    System.out.println(created + " is created!");
    // データ表示
    customerRepository.findAll()
        .forEach(System.out::println);
  }

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

実 行

「HajibootJpaApplication クラス」を実行しましょう。
ただし、この時点では「data.sql」と「schema.sql」は作らないでください。



プロジェクト構成

実行結果は、以下のようになります。

【コマンド・プロンプト】実行結果

```
2016-08-07 18:14:27.014 DEBUG 34652 --- [           main] jdbc.sqltim
ing                             :  org.hibernate.tool.hbm2ddl.DatabaseE
xporter.export(DatabaseExporter.java:47)
10. drop table customers if exists  {executed in 0 msec}
2016-08-07 18:14:27.020 DEBUG 34652 --- [           main] jdbc.sqltim
ing                             :  org.hibernate.tool.hbm2ddl.DatabaseE
xporter.export(DatabaseExporter.java:47)
10. create table customers (id integer generated by default as identi
ty, first_name varchar(255)
not null, last_name varchar(255) not null, primary key (id))  {execut
ed in 5 msec}
2016-08-07 18:14:27.021  INFO 34652 --- [           main] org.hiberna
te.tool.hbm2ddl.SchemaExport  : HHH000230: Schema export complete
2016-08-07 18:14:27.052  INFO 34652 --- [           main] j.LocalCont
ainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory
 for persistence unit 'default'
2016-08-07 18:14:27.431  INFO 34652 --- [           main] o.s.j.e.a.A
nnotationMBeanExporter        : Registering beans for JMX exposure on
startup
2016-08-07 18:14:27.472 DEBUG 34652 --- [           main] jdbc.sqltim
ing                             :  org.hibernate.engine.jdbc.internal.
ResultSetReturnImpl.executeUpdate(ResultSetReturnImpl.java:204)
10. insert into customers (id, first_name, last_name) values (null,
 'Hidetoshi', 'Dekisugi')  {executed in 1 msec}
Customer(id=1, firstName=Hidetoshi, lastName=Dekisugi) is created!
2016-08-07 18:14:27.524  INFO 34652 --- [           main] o.h.h.i.Que
ryTranslatorFactoryInitiator  : HHH000397: Using ASTQueryTranslatorFa
ctory
2016-08-07 18:14:27.653 DEBUG 34652 --- [           main] jdbc.sqltim
ing                             :  org.hibernate.engine.jdbc.internal.
ResultSetReturnImpl.extract(ResultSetReturnImpl.java:70)
10. select customer0_.id as id1_0_, customer0_.first_name as first_
```



```
na2_0_, customer0_.last_name
as last_nam3_0_ from customers customer0_  {executed in 0 msec}
Customer(id=1, firstName=Hidetoshi, lastName=Dekisugi)
2016-08-07 18:14:27.663  INFO 34652 --- [           main] com.examp
le.HajibootJpaApplication     : Started HajibootJpaApplication in
2.794 seconds (JVM running for 7.272)
```

「組み込みデータベース」[7] を使うと、起動時にエンティティに対応したテーブルの「削除」(drop table) や、「作成」(create table) がデフォルトで行なわれます。
「create 文」の中に「@Column」アノテーションで指定した「not null」属性が反映されていることが分かります。

その後、「customerRepository#save」による「insert」と、「customerRepository#findAll」による「select」が自動で発行されているのが分かります。

[2.3.3] 「JPQL」でクエリの定義

「JpaRepository」には定義されていない「検索処理」などをしたい場合は、「継承」したインターフェイスに、対応するメソッドを追加します。「JPQL」(Java Persistence Query Language) によってクエリを記述できます。

「JPQL」は「JPA」の「エンティティ操作」用の「問い合わせ言語」であり、「SQL」に似ています。
「JPQL」は実行時に「SQL」に変換されますが、RDBMS の実装によって、異なる SQL の方言を吸収してくれます。「JPQL」を利用することで、「ベンダー非依存」な「クエリ」を記述できます。

なお、「JPQL」の説明は、本書の範囲を超えるため、割愛します。

例として、「Customer」を「名前の昇順で取得するメソッド」(findAllOrderByName) を「CustomerRepository」に追加します。

CustomerRepository クラス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

public interface CustomerRepository extends JpaRepository<Customer,
Integer> {
    @Query("SELECT x FROM Customer x ORDER BY x.firstName, x.lastName")
// (1)
    List<Customer> findAllOrderByName();
}
```

[7]「組み込みデータベース」は「オンメモリ・モード」で使用した「H2」「HSQL」「Derby」です。

プログラム解説

項　番	説　明
(1)	「@Query」アノテーションを付けて、「JPQL」を記述。

「HajibootJpaApplication クラス」内でこの「メソッド」を実行します。
以下のように修正してください。

HajibootJpaApplication クラス

```
customerRepository.findAllOrderByName()
        .forEach(System.out::println);
```

ここで、2.2.3 で作成した「data.sql を src/main/resources」にコピーしてください。
その後、「HajibootJpaApplication クラス」を実行しましょう。
データの出力順が、以下のように変わりました。

【コマンド・プロンプト】出力されたデータ

```
Customer(id=5, firstName=Hidetoshi, lastName=Dekisugi)
Customer(id=1, firstName=Nobita, lastName=Nobi)
Customer(id=4, firstName=Shizuka, lastName=Minamoto)
Customer(id=3, firstName=Suneo, lastName=Honekawa)
Customer(id=2, firstName=Takeshi, lastName=Goda)
```

また、出力される「SQL ログ」に「order by customer0_.first_name, customer0_.last_name」が追加されました。

【コマンド・プロンプト】出力された「SQL ログ」

```
2014-06-28 18:33:33.047 DEBUG 5844 --- [           main] jdbc.sqltimi
ng                             :  org.hibernate.engine.jdbc.internal.Re
sultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
7. select customer0_.id as id1_0_, customer0_.first_name as first_
na2_0_, customer0_.last_name
as last_nam3_0_ from customers customer0_ order by customer0_.first_
name, customer0_.last_name
 {executed in 2 msec}
```

ノート 「JPA」では「JPQL」以外にもSQLを使うことができます。「Spring Data JPA」では以下のように「@Query」に「nativeQuery = true」を指定することで、「SQL」を使えます。

```
    @Query(value = "SELECT id,first_name,last_name FROM customers OR
DER BY first_name, last_name", nativeQuery = true)
    List<Customer> findAllOrderByName();
```

ただし、「SQL」を利用すると「JPA」のメリットが薄れてしまいます。
基本的には「JPQL」を使い、ベンダー依存の構文を使わないと表現できない複雑なクエリを記述する場合に、「SQL」を使えばいいでしょう。

[2.3.4] 「ページング処理」の実装

「Spring Data」では、データ・アクセスの際に「ページング処理」[8] を簡単に行なうための仕組みが用意されています。
「JpaRepository」にページング処理を行なうためのメソッドが予め用意されています。

HajibootJpaApplication クラス

```
package com.example;

import com.example.domain.Customer;
import com.example.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;

@SpringBootApplication
public class HajibootJpaApplication implements CommandLineRunner {
  @Autowired
  CustomerRepository customerRepository;

  @Override
  public void run(String... strings) throws Exception {
    // データ追加
    Customer created = customerRepository.save(new Customer(null, "Hi
detoshi", "Dekisugi"));
    System.out.println(created + " is created!");
    // ページング処理
    Pageable pageable = new PageRequest(0, 3); // (1)
    Page<Customer> page = customerRepository.findAll(pageable); // (2)
    // (3)
    System.out.println("1ページのデータ数=" + page.getSize());
    System.out.println("現在のページ=" + page.getNumber());
    System.out.println("全ページ数=" + page.getTotalPages());
    System.out.println("全データ数=" + page.getTotalElements());
    page.getContent().forEach(System.out::println); // (4)
  }

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

プログラム解説

項　番	説　明
(1)	「Pageable」インターフェイスで、取得するページング情報を用意。 「実装」クラスとして「PageRequest」クラスがある。 「コンストラクタ」の「第1引数」は「ページ数」で、「第2引数」は「1ページ当たりの件数」。 (ページ数は「0」始まりであることに注意してください)。

[8]「1 ページ」あたり「N 件」のデータを取得するとし、「M ページ目」のデータを取得する処理。

(2)	「findAll」(Pageable) メソッドを実行して、指定したページの「Customer」データを取得。「返り値」は「Page<Customer> 型」。
(3)	「Page#getSize」で「1 ページあたりのデータ数」、「Page#getNumber」で「現在のページ数」(0 始まり)、「Page#getTotalPages」で「全ページ数」、「Page#getTotalElements」で「全データ数」を取得できる。
(4)	「Page#getContent」で該当ページのデータのリストを取得できる。

実行

実行すると以下のように出力されます。

【コマンド・プロンプト】実行結果

```
1 ページあたりのデータ数 =3
現在のページ =0
全ページ数 =2
全データ数 =5
Customer(id=1, firstName=Nobita, lastName=Nobi)
Customer(id=2, firstName=Takeshi, lastName=Goda)
Customer(id=3, firstName=Suneo, lastName=Honekawa)
```

「SQL ログ」を見てみましょう。

(a)「全データ件数」を取得するための「SQL」と、(b)「該当ページのデータ」を取得する「SQL」の 2 つが発行されているのが分かります。

対象の「RDBMS」次第で、実行される SQL は異なります。

【コマンド・プロンプト】SQL ログ

```
2014-06-28 14:44:04.888 DEBUG 5264 --- [           main] jdbc.sqltimi
ng                             :  org.hibernate.engine.jdbc.internal.Re
sultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
7. select count(customer0_.id) as col_0_0_ from customers customer0_
{executed in 0 msec}
2014-06-28 14:44:04.895 DEBUG 5264 --- [           main] jdbc.sqltimi
ng                             :  org.hibernate.engine.jdbc.internal.Re
sultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
7. select customer0_.id as id1_0_, customer0_.first_name as first_
na2_0_, customer0_.last_name
as last_nam3_0_ from customers customer0_ limit 3  {executed in 1
msec}
```

「ページング処理」は、「リポジトリ」に追加したメソッドに対しても、適用できます。

先ほど追加した「findAllOrderByName」メソッドの「返り値の型」と「引数の型」を変更します。

引数に「Pageable」を追加し、「返り値の型」を「Page<Customer>」に変更します。

CustomerRepository インターフェイス

```
package com.example.repository;

import com.example.domain.Customer;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

public interface CustomerRepository extends JpaRepository<Customer,
Integer> {
    @Query("SELECT x FROM Customer x ORDER BY x.firstName, x.lastName")
    Page<Customer> findAllOrderByName(Pageable pageable);
}
```

「HajibootJpaApplication クラス」の「ページング処理」の部分を、以下のように修正します。

HajibootJpaApplication クラス

```
Page<Customer> page = customerRepository.findAllOrderByName(pageable);
```

実行すると、以下のように出力されます。

【コマンド・プロンプト】実行結果

```
1 ページあたりのデータ数 =3
現在のページ =0
全ページ数 =2
全データ数 =5
Customer(id=5, firstName=Hidetoshi, lastName=Dekisugi)
Customer(id=1, firstName=Nobita, lastName=Nobi)
Customer(id=4, firstName=Shizuka, lastName=Minamoto)
```

「SQL ログ」を確認すると、「JPQL」で記述したクエリに対しても、「ページ取得」のための条件が加わっていることが分かります。

【コマンド・プロンプト】SQL ログ

```
2014-06-28 14:59:15.813 DEBUG 5471 --- [           main] jdbc.sqltimi

ng                             :  org.hibernate.engine.jdbc.internal.Re
sultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
7. select count(customer0_.id) as col_0_0_ from customers customer0_
{executed in 0 msec}
2014-06-28 14:59:15.824 DEBUG 5471 --- [           main] jdbc.sqltimi
ng                             :  org.hibernate.engine.jdbc.internal.Re
sultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
8. select customer0_.id as id1_0_, customer0_.first_name as first_
na2_0_, customer0_.last_name
as last_nam3_0_ from customers customer0_ order by customer0_.first_
name, customer0_.last_name
limit 3  {executed in 1 msec}
```

ここでは「Spring Data JPA」の機能の一部しか紹介しませんでした。

詳細は、「公式ドキュメント」[9] を参照してください。

[9] http://docs.spring.io/spring-data/jpa/docs/1.10.2.RELEASE/reference/html/