

第3章

「Spring Boot」による「Webアプリ開発」

第2章まで準備ができたので、ここから、「Spring Boot」を使った「Webアプリケーション」の開発について説明します。

3.1

この章で作るアプリ

本章では、同じ題材で、以下の2種類のアプリケーションを作ります。

- (a) 「REST」 Web サービス
- (b) 「画面」のある Web アプリ

(a) は、いわゆる「Web API」の開発であり、「HTTP」を介してクライアント（「JavaScript」など）とやりとりするためのエンド・ポイントとなるサービスです。

最近の Web 開発では、「フロントエンド」を「HTML+JavaScript」で開発し、「バックエンド」を「REST Web サービス」にするスタイルが増えてきています。

一方、(b) は、伝統的な「画面遷移 Web アプリケーション」であり、「HTTP リクエスト」のたびに画面が更新されるものです。

「Spring Boot」を用いると、どちらも容易に開発できます。

実際には「アクション・ベース」の「Web MVC フレームワーク」である「Spring MVC」^[1] を使った「Web アプリ」プログラミングを行なうことになります。

本章では、前章で作った「CustomerService」「CustomerRepository」クラスを使って、「簡易顧客管理システム」^[2]を作ります。

まずは、この章で作成するアプリの雛形プロジェクトを「Spring Initializr」から生成します。

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-rest	Web, JPA, H2, Lombok [*]

* 1項目ずつ、入力してクリック

[1] <http://docs.spring.io/spring-framework/docs/4.3.2.RELEASE/spring-framework-reference/html/mvc.html>

[2] 「簡易 CRUD アプリケーション」のことを「顧客管理システム」と呼ぶのは憚れますが、ご容赦ください。



「Generate Project」をクリックし、ダウンロードした「hajiboot-rest.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-web」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-rest</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hajiboot-rest</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 --&gt;
  &lt;/parent&gt;

  &lt;properties&gt;
    &lt;project.build.sourceEncoding&gt;UTF-8&lt;/project.build.sourceEncoding&gt;
    &lt;project.reporting.outputEncoding&gt;UTF-8&lt;/project.reporting.outputEncoding&gt;
    &lt;java.version&gt;1.8&lt;/java.version&gt;
  &lt;/properties&gt;

  &lt;dependencies&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-data-jpa&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.projectlombok&lt;/groupId&gt;</pre>

```

```
<artifactId>lombok</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.lazyLuke</groupId>
  <artifactId>log4jdb-remix</artifactId>
  <version>0.2.7</version>
</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>
```

「hajiboot-rest」プロジェクトを「STS」にインポートした後、[2.3.3] で作った

- ・「Customer」クラス
- ・「CustomerRepository」クラス
- ・application.properties
- ・data.sql

を、この「プロジェクト」にコピーしてください。

*

また、[2.1.6] で中途半端に実装していた「CustomerService」クラスを、このプロジェクトにコピーし、残りの「CRUD 处理」を埋めましょう。

```
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 org.springframework.transaction.annotation.Transactional;

import java.util.List;
```

```

@Service
@Transactional
public class CustomerService {
    @Autowired
    CustomerRepository customerRepository;

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

    public Customer findone(Integer id) {
        return customerRepository.findone(id);
    }

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

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

    public void delete(Integer id) {
        customerRepository.delete(id);
    }
}

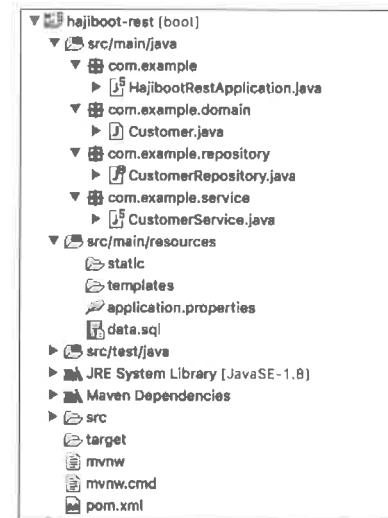
```

ここでは、説明を簡単にするために、単純に「CustomerRepository」のメソッドを呼び出すだけにしています。

本来はここに「ID 存在チェック」などロジックを含めるべきですが、今回は割愛します。

*

ファイルコピー後の「プロジェクト」は、次のような構造になっていることを確認してください。



「REST Web サービス」を作る準備すみの「プロジェクト構成」

3.2

「REST Web サービス」の開発

まずは、「顧客管理システム」を「REST Web サービス」化しましょう。

「REST」とは、「REpresentational State Transfer」の略であり、「クライアント」と「サーバ」間で「データ」をやり取りするための「ソフトウェア・アーキテクチャスタイル」の一つです。

「REST Web サービス」では、「サーバ・サイド」で管理している情報の中から「クライアント」に提供すべき情報を「リソース」として抽出し、「リソース」に対する「CRUD 操作」を「HTTP メソッド」（「POST」「GET」「PUT」「DELETE」など）を使って「Web API」としてクライアントに公開します。

この「Web API」のことを、「REST API」とも呼びます。

今回は「顧客 (Customer) リソース」に対する「CRUD 操作」の「Web API」を公開し、「HTTP」を経由して「顧客情報」を操作できるようにします。

公開する Web API

API 名	HTTP メソッド	リソース・パス	正常時 HTTP レスポンス・ステータス
顧客全件取得	GET	/api/customers	200 OK
顧客 1 件取得	GET	/api/customers/{id}	200 OK
顧客新規作成	POST	/api/customers	201 CREATED
顧客 1 件更新	PUT	/api/customers/{id}	200 OK
顧客 1 件削除	DELETE	/api/customers/{id}	204 NO CONTENT

[3.2.1]

「顧客全件取得」「顧客 1 件取得」API の実装

まずは、「顧客全件取得」「顧客 1 件取得」のための API を実装しましょう。

*

「Customer リソース API」の「エンド・ポイント」として、「CustomerRestController」クラスを作ります。

「CustomerRestController」クラスに実装する「メソッド」と「API」の対応を、以下にまとめます。

「CustomerRestController」クラスの「メソッド」と「API」の対応

API 名	メソッド名	返り値の型
顧客全件取得	getCustomers	List<Customer>
顧客 1 件取得	getCustomer	Customer

顧客新規作成	postCustomers	Customer
顧客1件更新	putCustomer	Customer
顧客1件削除	deleteCustomer	void

それでは「顧客全件取得」API用の「getCustomers」と、「顧客1件取得」API用の「getCustomer」を実装しましょう。

「CustomerRestController」クラス

```
package com.example.api;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController // (1)
@RequestMapping("api/customers") // (2)
public class CustomerRestController {
    @Autowired // (3)
    CustomerService customerService;

    // 顧客全件取得
    @GetMapping // (4)
    List<Customer> getCustomers() {
        List<Customer> customers = customerService.findAll();
        return customers; // (5)
    }

    // 顧客1件取得
    @GetMapping(path = "{id}") // (6)
    Customer getCustomer(@PathVariable Integer id) { // (6)
        Customer customer = customerService.findOne(id);
        return customer;
    }
}
```

プログラム解説

項目番号	説明
(1)	「REST」Webサービスの「エンド・ポイント」となる「Controller」クラスには、「@RestController」アノテーションを付ける。
(2)	この「REST」Webサービスにアクセスするための「パスのルート」を、「@RequestMapping」アノテーションに設定。 ここでは「api/customers」を設定したため、このクラス内の「メソッド単位」でパスを設定した場合は、「api/customers」からの「相対パス」になる。
(3)	作成済みの「CustomerService」を「DI」する。

- (4) 「getCustomers」メソッドに対して「@GetMapping」アノテーションを付与し、「HTTP メソッド」の「GET」を割り当てる。
この「メソッド」では、特にパスの設定はしないため、(2) の設定で、「/api/customers」に「GET」でアクセスすると、「getCustomers」メソッドが実行される。
- (5) 「CustomerService」の全件取得用メソッド「findAll」の結果を「メソッドの返り値」として返す。
「@GetMapping」アノテーションを付けた「メソッドの返り値」は、シリアル化されて、「HTTP レスポンス」の「ボディ」に設定される。
デフォルトでは「Java」オブジェクトは、「JSON 形式」でシリアル化される。
- (6) 「getCustomer」メソッドに対しても、「@GetMapping」アノテーションを付与し、「HTTP メソッド」の「GET」を割り当てる。
「path 属性」に「相対パス」を設定。
ここでは「相対パス」として「プレース・ホルダ」を含む形で{id}を指定した。
したがって、「/api/customers/{IDに相当する整数}」に「GET」でアクセスすると、「getCustomer」メソッドが実行される。
パス中の「プレース・ホルダ」に指定したパラメータは、メソッドの引数で取得できる。
対象の引数に「@PathVariable」アノテーションを付ける。
(「引数名」と「プレース・ホルダ」の値を一致させる必要がある)。

実行

このアプリケーションを実行してみましょう。

```
$ ./mvnw spring-boot:run
(途中略)
2016-08-10 03:26:56.435 INFO 11857 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{"path": "/api/customers/{id}", "method": "GET"}]" onto com.example.domain.Customer com.example.api.CustomerRestController.getCustomer(java.lang.Integer)
2016-08-10 03:26:56.437 INFO 11857 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{"path": "/api/customers", "method": "GET"}]" onto java.util.List<com.example.domain.Customer> com.example.api.CustomerRestController.getCustomers()
(途中略)
2016-08-10 03:26:56.903 INFO 11857 --- [           main] com.example.HajibootRestApplication      : Started HajibootRestApplication in 5.188 seconds (JVM running for 8.692)
```

どういう「パス」に対して、どういう「HTTP メソッド」「パラメータ」「ヘッダ」などでアクセスすると、どの「コントローラ」のどの「メソッド」にマッピングされるか、の情報が出力されます。

もしこの「ログ」が表示されなければ、「設定ミス」である可能性が高いです。

*

さっそく、「HTTP クライアント」の「curl」を利用して、「顧客リソース」の「REST API」にアクセスしてみましょう。

*

「顧客全件取得」の API を実行した結果は、以下の通りです。

[ターミナル] 「顧客全件取得」の API を実行

```
$ curl http://localhost:8080/api/customers -i -XGET
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 09 Aug 2016 18:35:31 GMT
[{"id":1,"firstName":"Nobita","lastName":"Nobi"}, {"id":4,"firstName":
"Shizuka","lastName":"Minamoto"}, {"id":3,"firstName":"Suneo","lastName":
"Honekawa"}, {"id":2,"firstName":"Takeshi","lastName":"Goda"}]
```

全件の顧客データが「JSON 形式」で返却されたことが分かります。

*

次に、「顧客 1 件取得」の API を実行します。

[ターミナル] 「顧客 1 件取得」の API を実行

```
$ curl http://localhost:8080/api/customers/1 -i -XGET
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 09 Aug 2016 18:36:16 GMT
{"id":1,"firstName":"Nobita","lastName":"Nobi"}
```

1 件ぶんのデータが、「JSON」で返却されたことが分かります。

[3.2.2] 「顧客新規作成」「顧客 1 件更新」「顧客 1 件削除」API の実装

残りの API を実装しましょう。

「CustomerRestController」クラス

```
package com.example.api;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("api/customers")
public class CustomerRestController {
    @Autowired
    CustomerService customerService;

    @GetMapping
    List<Customer> getCustomers() {
        List<Customer> customers = customerService.findAll();
        return customers;
    }
```

```
}
```

```
@GetMapping(path = "{id}")
Customer getCustomer(@PathVariable Integer id) {
    Customer customer = customerService.findOne(id);
    return customer;
}

@PostMapping // (1)
@ResponseStatus(HttpStatus.CREATED) // (2)
Customer postCustomer(@RequestBody Customer customer /* (3) */ {
    return customerService.create(customer);
}

@PutMapping(path = "{id}") // (4)
Customer putCustomer(@PathVariable Integer id, @RequestBody Customer customer) {
    customer.setId(id);
    return customerService.update(customer);
}

@DeleteMapping(path = "{id}") // (5)
@ResponseStatus(HttpStatus.NO_CONTENT) // (6)
void deleteCustomer(@PathVariable Integer id) {
    customerService.delete(id);
}
```

プログラム解説

項番	説明
(1)	「postCustomers」メソッドに対して、「@PostMapping」アノテーションを付与し、「HTTP メソッド」の「POST」を割り当てる。このメソッドでは特にパスの設定はしないため、「/api/customers」に「POST」でアクセスすると「postCustomers」メソッドが実行される。
(2)	「@ResponseStatus」アノテーションで API の正常時の HTTP レスポンスを設定できる。「HttpStatus.CREATED」を指定することで、「201 Created」が返る。設定しない場合は「200 OK」。
(3)	「HTTP リクエスト」のボディを「Customer」オブジェクトにマッピングするために、「@RequestBody」アノテーションを設定。
(4)	「putCustomer」メソッドに対して、「@PutMapping」アノテーションを付与し、「HTTP メソッド」の「PUT」を割り当てる。
(5)	「deleteCustomer」メソッドに対して、「@DeleteMapping」アノテーションを付与し、「HTTP メソッド」の「DELETE」を割り当てる。
(6)	「@ResponseStatus」アノテーションに「HttpStatus.NO_CONTENT」を指定することで、「204 No Content」が返る。

実行

アプリケーションを起動すると、以下のように「POST」「PUT」「DELETE」に関するマッピング情報が出力されます。

【ターミナル】起動時のログ

```
2016-08-11 01:11:32.074 INFO 35687 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/api/customers],methods=[POST]}" onto com.example.domain.Customer com.example.api.CustomerRestController.postCustomer(com.example.domain.Customer)
2016-08-11 01:11:32.074 INFO 35687 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/api/customers/{id}],methodS=[PUT]}" onto com.example.domain.Customer com.example.api.CustomerRestController.putCustomer(java.lang.Integer,com.example.domain.Customer)
2016-08-11 01:11:32.074 INFO 35687 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/api/customers/{id}],methodS=[DELETE]}" onto void com.example.api.CustomerRestController.deleteCustomer(java.lang.Integer)
```

*

「curl」を用いて、これらのAPIを実行しましょう。

まずは、「顧客新規作成」APIを実行します。

「JSON」を送信するために、「-d」オプションで「リクエスト・ボディ」に顧客情報の「JSON」を設定し、「-H」オプションで「HTTPヘッダ」に「Content-Type: application/json」を設定します。

【ターミナル】「顧客新規作成」APIを実行

```
$ curl http://localhost:8080/api/customers -i -XPOST -H "Content-Type: application/json" -d "{\"firstName\":\"Tamako\",\"lastName\":\"Nobi\"}"
HTTP/1.1 201
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 10 Aug 2016 16:14:25 GMT

{"id":5,"firstName":"Tamako","lastName":"Nobi"}
```

「HTTPステータス」の「201 Created」が返り、「ID」が採番された「顧客情報」が「レスポンス・ボディ」として返りました。

*

次に、「顧客1件更新」APIを実行します。

「顧客新規作成」とほぼ同じですが、「HTTPメソッド」を「PUT」に変え、「URL」に「更新対象の顧客ID」を含めてください。

【ターミナル】「顧客1件更新」APIを実行

```
$ curl http://localhost:8080/api/customers/1 -i -XPUT -H "Content-Type: application/json" -d "{\"firstName\":\"Nobio\",\"lastName\":\"Nobi\"}"
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: wed, 10 Aug 2016 16:22:05 GMT

{"id":1,"firstName":"Nobio","lastName":"Nobi"}j
```

「更新した顧客情報」が返却されました。

*

最後に、「顧客1件削除」APIを実行します。

「顧客1件取得」と、ほぼ同じです。

【ターミナル】「顧客1件削除」APIを実行

```
$ curl http://localhost:8080/api/customers/1 -i -XDELETE
HTTP/1.1 204
Date: wed, 10 Aug 2016 16:24:55 GMT
```

「HTTPステータス」の「204 No Content」が返り、「レスポンス・ボディ」は「空」です。

ノート 「REST」Webサービスでは、「POST」によって「新規作成」したリソースにアクセスするための「URI」を、「HTTPレスポンス」の「Locationヘッダ」に設定することが一般的です。

「Spring MVC」で「Locationヘッダ」に「リソースのURI」を設定するには、以下のようにソース・コードを修正します。

```
package com.example.api;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;
import java.util.List;
@RestController
@RequestMapping("api/customers")
public class CustomerRestController {
    // (略)

    @PostMapping
```

```

    ResponseEntity<Customer> postCustomers(@RequestBody Customer customer, UriComponentsBuilder uriBuilder /* (1) */ {
        Customer created = customerService.create(customer);
        URI location = uriBuilder.path("api/customers/{id}")
            .buildAndExpand(created.getId()).toUri(); // (2)
        return ResponseEntity.created(location).body(created); // (3)
    }
    // (略)
}

```

プログラム解説

項番	説明
(1)	コンテキスト・パス相対の URI を構築するのに便利な「UriComponentsBuilder」を「コントローラ」の「メソッド引数」にとる。
(2)	「UriComponentsBuilder」と、作った「Customer」オブジェクトの「id」を用いて、「リソースの URI」を作る。 「path」メソッド内の「{id}」は「プレース・ホルダ」で、「buildAndExpand」メソッドに渡す値で置換される。
(3)	「HTTP レスポンス・ヘッダ」を設定したい場合は、メソッドは「Customer」オブジェクトを返却するのではなく、「ResponseEntity」オブジェクトを返却する。 「ResponseEntity」オブジェクトには「レスポンス・ボディ」である「Customer」オブジェクト、「レスポンス・ヘッダ」である「HttpHeaders」オブジェクト、そして「ステータス・コード」である「HttpStatus」を設定する。 今回は「Location」ヘッダを設定し、「ステータス・コード」として「201 Created」を設定するためのファクトリ・メソッドである「created」メソッドを使っている。

実行

アプリケーションを再起動して、「顧客新規作成」API を実行すると、以下のように「レスポンス・ヘッダ」の「Location」に、「新規作成された顧客」の「URI」が出力されます。

【ターミナル】「顧客新規作成」API を実行

```

$ curl http://localhost:8080/api/customers -i -XPOST -H "Content-Type: application/json" -d "{\"firstName\":\"Tamako\", \"lastName\":\"Nobi\"}"
HTTP/1.1 201
Location: http://localhost:8080/api/customers/5
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 10 Aug 2016 16:49:23 GMT
>
{"id":5, "firstName":"Tamako", "lastName":"Nobi"}

```

* ここまで「REST API」の簡単な「CRUD 操作」の「実装方法」を学びました。
基本的には、ここで説明した内容で「REST API」を構築できます。

ノート 本節では、「入力チェック」の説明を省略しました。

「入力チェック」の実装については、画面のある Web アプリで扱います。「[3.3.2] 「顧客情報」の新規作成」を参照してください。

[3.2.3]

「ページ・ネーション」の実装

「Spring Boot」を使うと、2章で紹介した「Spring Data」の「Pageable」クラスを「Controller」の引数として受け取れる設定があらかじめ行なわれます。

*

「顧客全件取得 API」の仕様を少し変えて、「ページ数」「1 ページあたりの件数」を指定できるように変更してみましょう。

CustomerRestController クラス

```

package com.example.api;
// (略)
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
// (略)
@RestController
@RequestMapping("api/customers")
public class CustomerRestController {
    // (略)

    @GetMapping
    Page<Customer> getCustomers(@PageableDefault Pageable pageable /* (1) */ {
        Page<Customer> customers = customerService.findAll(pageable); // (2)
        return customers;
    }
    // (略)
}

```

プログラム解説

項番	説明
(1)	「引数」に「Pageable」オブジェクトを追加することで、「ページネーション」の情報を取得できる。 「リクエスト・パラメータ」に設定した「page」「size」がこの「Pageable」オブジェクトにマッピングされる。 パラメータが指定されなかった場合のデフォルト値を設定するため、「@PageableDefault」アノテーションを付ける。

「@PageableDefault」アノテーションに「page」や「size」のデフォルト値を指定できるが、何も指定しなかった場合は、「page=0」「size=20」が設定される。なお、「page」パラメータは「0」始まりである点に注意。

- (2) 「ページング」情報を「CustomerService」に渡し、検索する。
結果が「Page」オブジェクトとして返るようにする。
(この「findAll」メソッドは、次に実装する)。

*

「CustomerService」クラスも「ページング処理」に対応できるように、メソッドを追加しましょう。

CustomerService クラス

```
package com.example.service;

// (略)
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
// (略)

@Service
@Transactional
public class CustomerService {
    // (略)

    public Page<Customer> findAll(Pageable pageable) {
        return customerRepository.findAllOrderByName(pageable);
    }
    // (略)
}
```

実行

アプリケーションを起動して、「GET リクエスト」を送ってみましょう。

【ターミナル】GET リクエスト送信

```
$ curl -i -XGET "http://localhost:8080/api/customers"
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 10 Aug 2016 17:02:32 GMT

{"content": [{"id": 1, "firstName": "Nobita", "lastName": "Nobi"}, {"id": 4, "firstName": "Shizuka", "lastName": "Minamoto"}, {"id": 3, "firstName": "Suneo", "lastName": "Honekawa"}], "last": true, "totalPages": 1, "totalElements": 4, "size": 10, "number": 0, "sort": null, "first": true, "numberOfElements": 4}
```

「全件取得 API」の結果が「List<Customer>」から「Page<Customer>」に代わったので、「顧客情報」は「content」プロパティに含まれ、「ページ情報」に関するプロパティも含まれていることが分かります。

デフォルトは「page=0」「size=20」相当であるため、データが「4 件」の場合は全件返却されます。

*

次に、「page=0」「size=3」のページング情報を加えてリクエストを送りましょう。

【ターミナル】ページング情報を指定

```
$ curl -i -XGET "http://localhost:8080/api/customers?page=0&size=3"
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 10 Aug 2016 17:08:48 GMT
```

```
{"content": [{"id": 1, "firstName": "Nobita", "lastName": "Nobi"}, {"id": 4, "firstName": "Shizuka", "lastName": "Minamoto"}, {"id": 3, "firstName": "Suneo", "lastName": "Honekawa"}], "last": false, "totalPages": 2, "totalElements": 4, "size": 3, "number": 0, "sort": null, "first": true, "numberOfElements": 3}
```

4 件中 3 件のデータが返却されました。

*

次に、「page=1」に変更しましょう。

「page=0」「size=2」を指定することで、「先頭から 2 件」だけが返却されます。

【ターミナル】「page=1」に変更

```
$ curl -i -XGET "http://localhost:8080/api/customers?page=1&size=3"
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 10 Aug 2016 17:08:53 GMT
```

```
{"content": [{"id": 2, "firstName": "Takeshi", "lastName": "Goda"}], "last": true, "totalPages": 2, "totalElements": 4, "size": 3, "number": 1, "sort": null, "first": false, "numberOfElements": 1}
```

「page=1」「size=2」を指定することで、「3 ~ 4 件目」が返却されます。

この「ページング処理」を使って、「ソート条件」の指定もできます。「Spring Data のドキュメント」^[3]を参考に、試してみてください。

3.3 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

ここまででは「JSON」をやり取りするだけの、「画面のないアプリケーション」を開発してきました。

本節では、一般的な「画面遷移のある Web アプリ」を作ります。

*

「Thymeleaf」^[4] は、素の HTML に「th:***」属性（または「data-th-***」属性）を付けることで、動的な画面が作れる「テンプレート・エンジン」です。

「テンプレート」を「ブラウザ」や「オーサリング・ツール」でそのまま見ることができます。

「デザイナ」が作った「HTML」に少しの「属性」を追加するだけで、「サーバ・サイド」で使えるようになりますし、修正したテンプレートをそのまま「デザイナ」が変更することもでき、「デザイナ・フレンドリー」です。

「プログラマ」と「デザイナ」間での「HTML」のやり取りにおける変換コストが少なく、協業に向いた「テンプレート・エンジン」だと言われています。

*

これまで「Spring MVC」を使った「Web アプリ開発」では「JSP」（Java Server Pages）を用いることが多かったです。

しかし、「Spring Boot」では「Thymeleaf」が画面作成の第一候補になります^[5]。

*

本節では、これまで作ってきた「簡易顧客管理システム」を「Thymeleaf」を使った画面のあるアプリにします。

*

以下のような画面遷移を実装しましょう。

【コマンド・プロンプト】「簡易顧客管理システム」の処理一覧

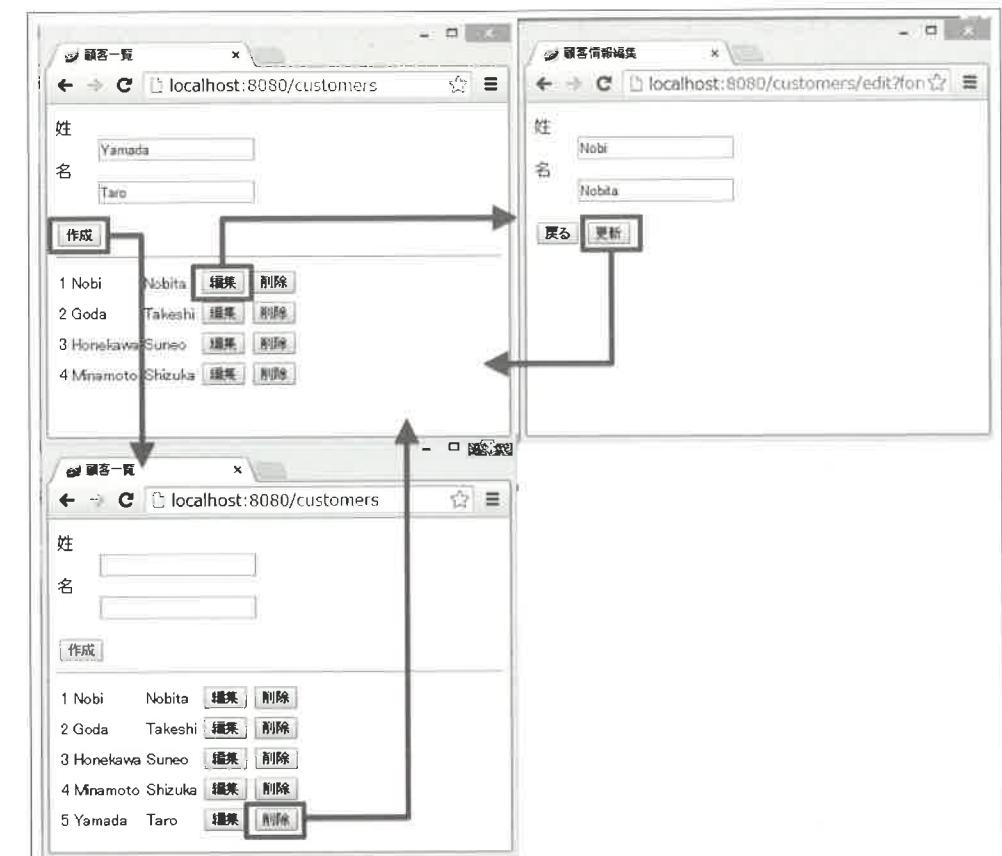
処理名	HTTP メソッド	リソース・パス	画面名
「顧客情報」一覧表示処理	GET	/customers	customers/list
「顧客情報」新規作成処理	POST	/customers/create	「顧客情報」一覧表示処理 ヘリダイレクト
「顧客情報」編集フォーム表示処理	GET	/customers/edit?form&id={id}	customers/edit
「顧客情報」編集処理	POST	/customers/edit&id={id}	「顧客情報」一覧表示処理 ヘリダイレクト
「顧客情報」削除処理	POST	/customers/delete?id={id}	「顧客情報」一覧表示処理 ヘリダイレクト

画面遷移のイメージを以下に示します。

[4] <http://www.thymeleaf.org/>

[5] 逆に「JSP」は、「Spring Boot」を利用する上で制限が多く、推奨されません。

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発



画面遷移のイメージ

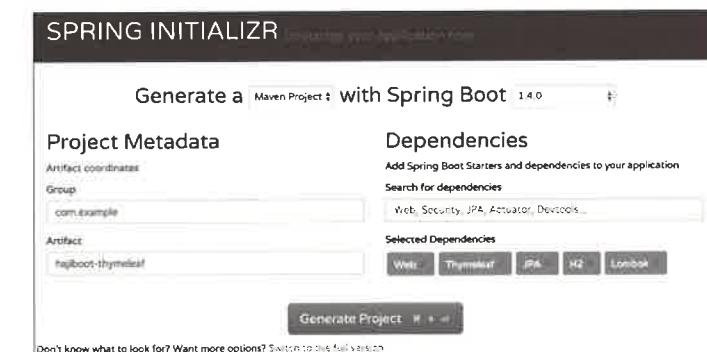
まずは、この章で作るアプリの雛形プロジェクトを「Spring Initializr」から生成します。

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-thymeleaf	Web, Thymeleaf, 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-rest.zip」を開いてください。

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-thymeleaf」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-thymeleaf</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>hajiboot-thymeleaf</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 --&gt;
  &lt;/parent&gt;

  &lt;properties&gt;
    &lt;project.build.sourceEncoding&gt;UTF-8&lt;/project.build.sourceEncoding&gt;
    &lt;project.reporting.outputEncoding&gt;UTF-8&lt;/project.reporting.outputEncoding&gt;
    &lt;java.version&gt;1.8&lt;/java.version&gt;
  &lt;/properties&gt;

  &lt;dependencies&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-data-jpa&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.projectlombok&lt;/groupId&gt;
      &lt;artifactId&gt;lombok&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-thymeleaf&lt;/artifactId&gt;
    &lt;/dependency&gt;
    &lt;dependency&gt;
      &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
      &lt;artifactId&gt;spring-boot-starter-web&lt;/artifactId&gt;
    &lt;/dependency&gt;

    &lt;dependency&gt;
      &lt;groupId&gt;com.h2database&lt;/groupId&gt;
      &lt;artifactId&gt;h2&lt;/artifactId&gt;
      &lt;scope&gt;runtime&lt;/scope&gt;
    &lt;/dependency&gt;</code>
```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

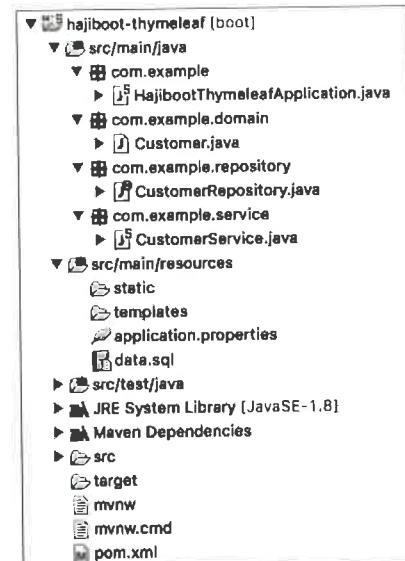
```
<dependency>
  <groupId>org.lazyLuke</groupId>
  <artifactId>log4jdb-remix</artifactId>
  <version>0.2.7</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>

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

「hajiboot-thymeleaf」プロジェクトを「STS」へインポートした後、[3.2.2] で作った

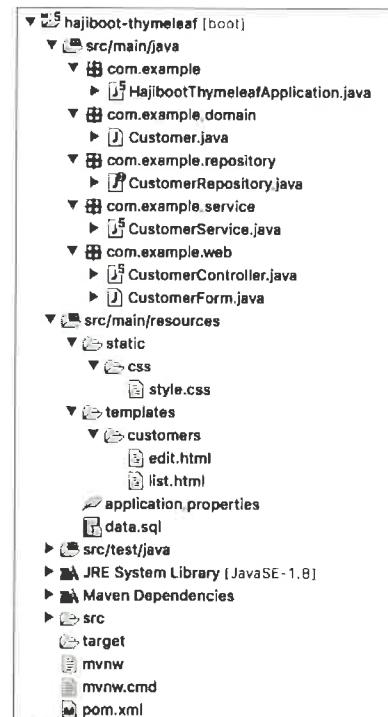
- ・「Customer」クラス
- ・「CustomerRepository」クラス
- ・「CustomerService」クラス
- ・application.properties
- ・data.sql

を、このプロジェクトにコピーしてください。



準備済みのプロジェクト構成

[3.3] でアプリケーション完成後は、以下のようなプロジェクト構成になります。



完成後のプロジェクト構成

[3.3.1]

「顧客情報」の一覧画面表示

「顧客管理アプリケーション」の画面遷移は「CustomerController」に実装します。
「パッケージ名」は「REST API」のときは分けて、「com.example.web」にします。

*

まずは、顧客情報を一覧表示する処理を、以下のように実装します。

CustomerController クラス

```
package com.example.web;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller // (1)
@RequestMapping("customers") // (2)
public class CustomerController {
    @Autowired
    CustomerService customerService;

    @GetMapping // (3)
```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

```
string list(Model model /* (4) */ {
    List<Customer> customers = customerService.findAll();
    model.addAttribute("customers", customers); // (5)
    return "customers/list"; // (6)
}
```

プログラム解説

項番	説明
(1)	「REST API」ではなく、「画面遷移用のコントローラ」には「@Controller」アノテーションを付ける。
(2)	「@RequestMapping」アノテーションで「CustomerController」クラスがマッピングする「URL」の接頭辞（ここでは「customers」）を設定する。
(3)	「list」メソッドが「URL」の「/customers」にマッピングされるように、「@GetMapping」アノテーションを付ける。
(4)	「Spring MVC」では画面に値を渡すために、「Model」オブジェクトを使用。コントローラの引数で「Model」を受け取り、「Model#addAttribute」メソッドを用いて、画面に渡す属性を設定。
(5)	「CustomerService#findAll」の結果を「Model」に設定。属性名を「"customers"」にする。画面からは、この「"customers"」を用いてアクセスできる。
(6)	「@Controller」を付けたコントローラのリクエスト処理メソッドの返り値は、「ビューア」、すなわち「遷移する画面の名前」になる。
	「Spring Boot」ではデフォルトで、「classpath:templates/+ “ビューア” + .html」が「画面のパス」になる。 この場合は、「classpath:templates/customers/list.html」を表示。

*

次は、「src/main/resources/templates/customers/list.html」に、「一覧表示」に対応する以下の HTML を作ります。

[src/main/resources/templates/customers/list.html] 一覧表示

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"><!-- (1) -->
<head>
    <meta charset="UTF-8"/>
    <title>顧客一覧 </title>
</head>
<body>
<table>
    <tr th:each="customer : ${customers}"><!-- (2) -->
        <td th:text="${customer.id}">100</td><!-- (3) -->
        <td th:text="${customer.lastName}">山田 </td>
        <td th:text="${customer.firstName}">太郎 </td>
        <td>
            <form th:action="@{/customers/edit}" method="get"><!-- (4) -->
                <input type="submit" name="form" value="編集" />
                <input type="hidden" name="id" th:value="${customer.id}" />
            </form>
        <!-- (5) -->
        </td>
    </tr>
</table>
```

```

</td>
<td>
    <form th:action="@{/customers/delete}" method="post">
        <input type="submit" value="削除" />
        <input type="hidden" name="id" th:value="${customer.id}" />
    </form>
</td>
</tr>
</table>
</body>
</html>

```

プログラム解説

項番	説明
(1)	「Thymeleaf」の「th:***」属性を使うための「名前空間」を指定。指定しなくとも動作はするが、設定しないと「IDE」に警告される。
(2)	「th:each」属性を使うことで、「List<Customer>」の内容を1件ずつループしてアクセスできる。 「th:each="(繰り返し要素内で使用する属性名) : \${(繰り返し対象のオブジェクトの属性名)}"」という記述をする。
(3)	「th:text」属性の値で、対象の「HTML タグ」内の「文字列」を置換できる。この「HTML テンプレート」を「ブラウザ」で見たときは、「100」が表示されるが、「サーバ・サイド」で「レンダリング」する際に「customer.id」の値で置換される。置換する値は、デフォルトで「HTML エスケープ」されるため、「クロス・サイト・スクリプティング攻撃」を防げる。
(4)	「th:action」属性の値で、「form タグ」の「action」属性の内容を置換できる。「@{***}」という記法を使うことで、「コンテキスト・パス相対のパス」を「絶対パス」に置換できる。
(5)	「th:value」属性の値で、「input タグ」の「value」属性の内容を置換できる。

この HTML をブラウザで直接開いてみましょう。



「src/main/resources/templates/customers/list.html」を直接開く

「th:***」属性を無視した状態で、“普通の”「HTML の結果」が表示されます。サーバでレンダリングすることなく、デザインを確認できるのが、「Thymeleaf」のメリットです。

*

次はアプリケーションを起動して、「http://localhost:8080/customers」にアクセスしましょう。

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発



「http://localhost:8080/customers」にアクセス

「th:***」属性が評価され、DB から取得した顧客情報が一覧で表示されました。

*

「Spring Boot」ではデフォルトで「Thymeleaf」のテンプレート評価結果がキャッシュされ、何度もテンプレートが評価されて性能が悪くならないように設定されています。

これは、「運用時」には有用ですが、「開発時」には何度もアプリケーションを再起動する必要があり、不便です。

この「キャッシング機能」は、「application.properties」に、以下の設定を行なうことで、無効にできます。

[application.properties] 「テンプレート・キャッシング」の無効化の設定

spring.thymeleaf.cache=false

アプリケーション運用時にはキャッシングを有効にすべきなので、開発中にはこの設定をしておき、「実行可能 jar」を作つて実行する際に、「--spring.thymeleaf.cache=true」を指定するといいです。

ただし、この方法は、少し手間がかかりてしまいます。

この方法の代わりに、[1.3.6] で紹介した「Spring Dev Tools」を使うと、「spring.thymeleaf.cache」の設定を明示的に変更することなく、①開発中はキャッシングを無効化し、②「java -jar」で「実行可能 jar」を実行する際には、キャッシングを有効にすることができます。

「spring.thymeleaf.cache」の設定を変えるのではなく、「pom.xml」に、以下を追加して「Spring Dev Tools」を有効にするといいでしょう。

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>

```

[3.3.2]

「顧客情報」の新規作成

次は、「顧客情報」の「新規作成」処理を実装しましょう。

*

まずは「更新用フォーム」に対応する「CustomerForm」クラスを作ります。

このクラスに HTML の <form> から送るパラメータをマッピングさせます。

入力チェックに「Bean Validation」を使い、入力ルールに対応するアノテーションを「CustomerForm」クラスのフィールドに付けます。

「Bean Validation」は Java 標準の「入力チェック・フレームワーク」です。「Bean Validation」を使うことによって、「Java Bean」にアノテーションで「入力チェック・ルール」を指定できます。

「Spring Boot」は「Bean Validation」の実装ライブラリとして、「Hibernate Validator」を使っています。

CustomerForm クラス

```
package com.example.web;
import lombok.Data;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
@Data // (1)
public class CustomerForm {
    @NotNull // (2)
    @Size(min = 1, max = 127) // (3)
    private String firstName;
    @NotNull
    @Size(min = 1, max = 127)
    private String lastName;
}
```

プログラム解説

項番	説明
(1)	ここでも「Lombok」の「@Data」アノテーションを使い、「setter/getter」の実装などを省略する。
(2)	「入力チェック」のためのアノテーションを付ける。 「firstName」というパラメータが送られてこない場合にエラーにするため、「@NotNull」アノテーションを付ける。
(3)	「firstName」の「文字列長」が「1 文字以上、127 文字以下」に制限するため、「@Size」アノテーションを付ける。

*

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

この「CustomerForm」クラスを用いて「CustomerController」に「新規作成処理用」のメソッド「create」を作ります。

CustomerController クラス

```
package com.example.web;
import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@Controller
@RequestMapping("customers")
public class CustomerController {
    @Autowired
    CustomerService customerService;
    @ModelAttribute // (1)
    CustomerForm setUpForm() {
        return new CustomerForm();
    }
    @GetMapping
    String list(Model model) {
        List<Customer> customers = customerService.findAll();
        model.addAttribute("customers", customers);
        return "customers/list";
    }
    @PostMapping(path = "create")
    String create(@Validated /* (2) */ CustomerForm form, BindingResult result, Model model) {
        if (result.hasErrors()) { // (3)
            return list(model);
        }
        Customer customer = new Customer();
        BeanUtils.copyProperties(form, customer); // (4)
        customerService.create(customer);
        return "redirect:/customers"; // (5)
    }
}
```

プログラム解説

項番	説明
(1)	「@ModelAttribute」アノテーションを付けたメソッド内で「CustomerForm」を初期化。

「@ModelAttribute」アノテーションを付けたメソッドはコントローラ内の「@RequestMapping」でマッピングされたメソッドの前に実行され、「返り値」は自動で

	「Model」に追加される。
	この例だと、「list」や「create」メソッドが呼ばれる前に、「model.addAttribute(new CustomerForm())」相当の処理が行なわれる。 (なお、「addAttribute」で「属性名」を省略した場合は、「属性値」の「クラス名の先頭」を「小文字」にした「文字列」が入る。この場合は、「customerForm」)。
(2)	送信されたフォームの情報の入力チェックを行なうために「@Validated」アノテーションを付ける。 これによって、「CustomerForm」に設定した「Bean Validation」のアノテーションが評価され、結果が隣の引数の「BindingResult」に格納される。
(3)	入力チェックの結果を確認し、エラーがある場合は一覧画面表示に戻る。
(4)	「CustomerForm」を「Customer」にコピーする。 ここでは簡易な「org.springframework.beans.BeanUtils」を使ったが、「フィールドの名前」と「型」が同じ場合にしか使えない。 (より柔軟な「Bean 変換」をする場合は、「Dozer」 ^[6] や「ModelMapper」 ^[7] を利用することをお勧め)。
(5)	「新規作成処理」が正常に終了した場合は一覧画面表示にリダイレクトする ^[8] 。 リダイレクトする場合は、ビュー名を「redirect:遷移先パス」にする。

ノート 「CustomerForm」を作らず、「Customer」をそのまま使うこともできます。
ただし、「画面から送信するフォームの項目数」が「DB に格納する情報」より多い場合がよくあります。

たとえば、「パスワード」と「確認用パスワード」をフォームから送信する場合、通常、「確認用パスワード」は「DB」には保存されません。そのため、「Customer」クラスに、通常、「確認用パスワード・フィールド」は不要です。

このような場合など、「Customer」クラスが画面の要件によって汚染されないように、画面の情報は「CustomerForm」で表現し、コントローラで変換することをオススメします。

少し煩わしいように思えますが、最終的にはアプリケーションの保守性は高まります。
また、「Lombok」を使えば、「フォーム用のクラスを作るコスト」は低くなり、「Dozer」や「ModelMapper」などを用いれば、「変換処理を実装するコスト」も小さくなります。

*

次に、「新規作成フォーム」を「list.html」に追加しましょう。

[list.html] 新規作成フォームの追加

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>顧客一覧 </title>
</head>
<body>
    <div>
        <form th:action="@{/customers/create}" th:object="${customerForm}" method="post"><!-- (1) -->
            <dl>
                <dt><label for="lastName">姓 </label></dt>
                <dd>
                    <input type="text" id="lastName" name="lastName" th:id="*{lastName}" th:errorclass="error-input"
                           value="山田"/><!-- (2) -->
                    <span th:if="${#fields.hasErrors('lastName')}" th:errors="*{lastName}"
                           class="error-messages">error!</span><!-- (3) -->
                </dd>
                <dt><label for="firstName">名 </label></dt>
                <dd>
                    <input type="text" id="firstName" name="firstName" th:id="*{firstName}" th:errorclass="error-input"
                           value="太郎"/>
                    <span th:if="${#fields.hasErrors('firstName')}" th:errors="*{firstName}"
                           class="error-messages">error!</span>
                </dd>
            </dl>
            <input type="submit" value="作成" />
        </form>
    </div>
    <hr/>
    <table>
        <tr th:each="customer : ${customers}">
            <td th:text="${customer.id}">100</td>
            <td th:text="${customer.lastName}">山田 </td>
            <td th:text="${customer.firstName}">太郎 </td>
            <td>
                <form th:action="@{/customers/edit}" method="get">
                    <input type="submit" name="form" value="編集" />
                    <input type="hidden" name="id" th:value="${customer.id}"/>
                </form>
            </td>
            <td>
                <form th:action="@{/customers/delete}" method="post">
                    <input type="submit" value="削除" />
                    <input type="hidden" name="id" th:value="${customer.id}"/>
                </form>
            </td>
        </tr>
    </table>
</body>
</html>
```

[6] http://dozer.sourceforge.net/

[7] http://modelmapper.org/

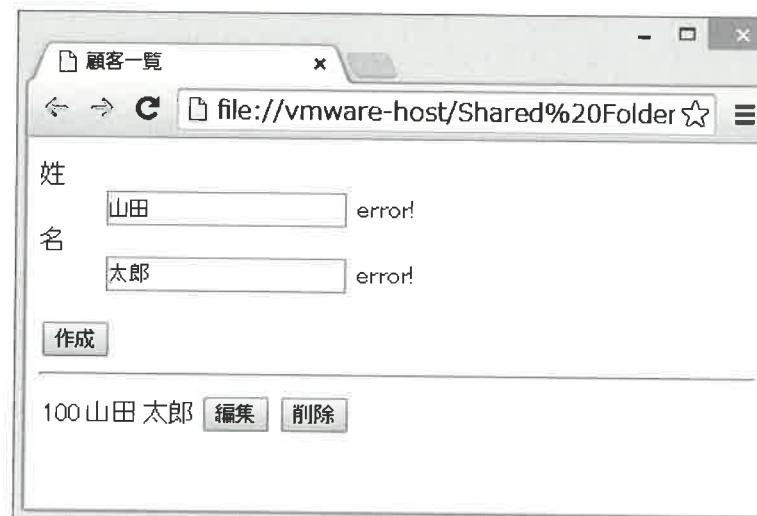
[8] POST 処理の後にリダイレクトを行ない、次の画面を GET するパターンを「PRG パターン」と言います。「PRG パターン」を利用することで、POST 後の画面をリロードして、再 POST されてしまうことを防ぎます。

プログラム解説

項番	説明
(1)	「th:action」属性の値で、「form タグ」の「action」属性の内容を置換できる。 「th:object」属性の値に「Model」の「属性名」を指定することで、この「タグ内」で「*{ フィールド名 }」記法を利用できる。
(2)	「th:field="*{ フィールド名 }"」を設定することで、この「HTML フィールド」と「フォーム・オブジェクト」(ここでは「CustomerForm」)のフィールドをバインディングできる。 この「HTML フィールド」の値が「フォーム・オブジェクト」の対応したフィールドに設定される。 逆に、「Model」上の「フォーム・オブジェクト」のフィールド値が対応した HTML のフィールドの値に設定される。 「入力チェックエラー」時には、「th:errorclass」属性の値が「class」属性に設定される。
(3)	対象のフィールドに「エラー」がある場合のみ表示したいタグには、「th:if="\${#fields.hasErrors('フィールド名')}"」を設定する。 「th:errors="*{ フィールド名 }"」を設定することで、対象のフィールドに関する「エラー・メッセージ」で「タグ内」の「文字列」を置換できる。

実行

まずは、この「list.html」をブラウザで直接開いてみましょう。



「list.html」を直接開く

「thymeleaf」に関する属性は無視されて、「普通の HTML」として表示されます。

*

こんどはアプリケーションを実行して、「http://localhost:8080/customers」にアクセスしましょう。

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発



「http://localhost:8080/customers」にアクセス

「thymeleaf」に関する属性が評価され、「list.html」に設定していたダミー値は消えていることが分かります。

「新規作成フォーム」に情報を入力し、「作成ボタン」を押し、「一覧画面」に作った情報が表示されていることを確認してください。

● 入力チェックエラー時の CSS

次に、「入力チェックエラー時の CSS」を設定しましょう。

「src/main/resources/static/css/style.css」を作り、以下の内容を記述してください。

[src/main/resources/static/css/style.css] 入力チェックエラー時の CSS

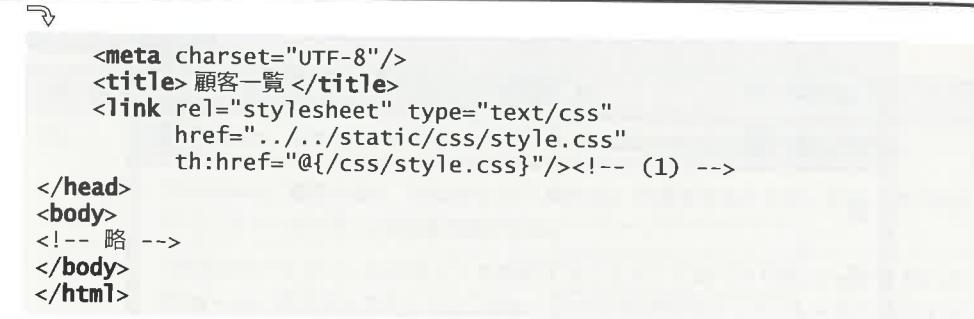
```
.error-input {
    border-color: #b94a48;
    margin-left: 5px;
}

.error-messages {
    color: #b94a48;
}
```

「list.html」には以下のように記述します。

[list.html] 一覧画面

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
```



プログラム解説

項番	説明
(1)	「th:href」属性の値で、「link タグ」の「href」属性の内容を置換できる。 これによって、ブラウザで確認する際には「相対パス」で、アプリケーションの実行時は「コンテキスト・パス相対」で「CSS」を参照できる。 なお、「クラス・パス」直下の「static」フォルダ以下のファイルは、「コンテキスト・パス相対」のパスで「静的リソース」としてアクセスできる。 そのため、「 http://localhost:8080/css/style.css 」で「クラス・パス」直下の「static/css/style.css」にアクセスできる。

実行

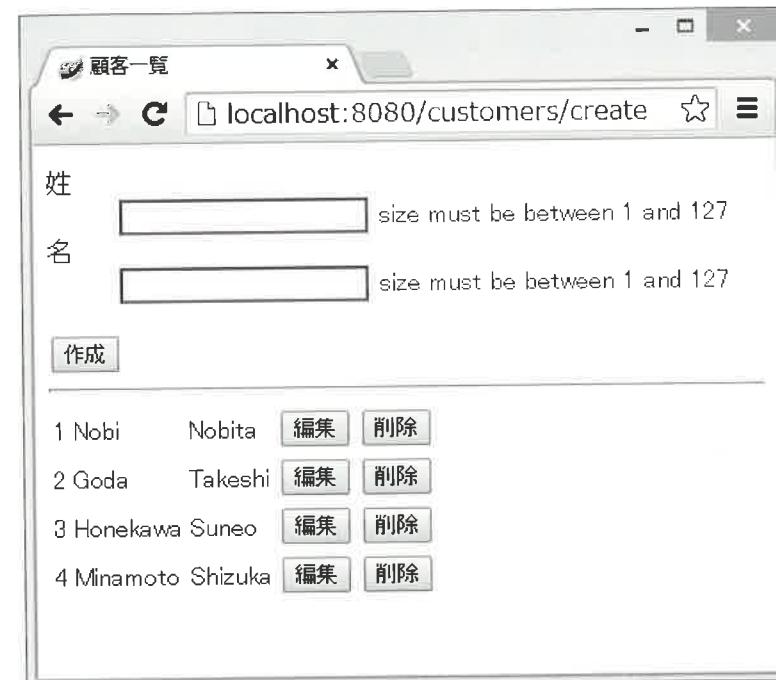
まずはこの「list.html」をブラウザで開くと、以下のように表示されます。



「list.html」を直接開く

「アプリケーション実行後」に「<http://localhost:8080/customers>」にアクセスし、「未入力」のまま「送信ボタン」を押すと、以下のように「エラー・メッセージ」が表示されます。

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

「<http://localhost:8080/customers>」にアクセス

ノート 「REST Web サービス」で「リクエスト・ボディ」の「JSON」の「入力チェック」を行なう場合、以下のように「更新系 API」のメソッド引数の「@RequestBody」が付いている引数に対して、「@Validated」アノテーションを付けるだけでいいです。「BindingResult」は不要です。

```

@PostMapping
 ResponseEntity<Customer> postCustomers(@Validated /* 追加 */ @RequestBody Customer customer, UriComponentsBuilder uriBuilder) {
    // (略)
}

```

```

@PostMapping(path = "{id}")
 Customer putCustomer(@PathVariable Integer id, @Validated /* 追加 */ @RequestBody Customer customer) {
    // (略)
}

```

この例の場合、「Customer」クラスのフィールドに、「@NotNull」や「@Size」など、「Bean Validation」のアノテーションを付ける必要があります。

[3.3.3]

「顧客情報」の編集

「顧客情報」の「編集処理」を追加しましょう。

*

まずは「CustomerController」にメソッドを追加します。

(a) 「/customers/edit?form&id={id}」に「GET」でアクセスして「特定の顧客」の「更新フォームを表示」する「editForm」メソッドと、(b) 「/customers/edit&id={id}」に「POST」でアクセスして「特定の顧客」の「更新処理を実施」する「edit」メソッドを追加しましょう。

CustomerController クラス

```
package com.example.web;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequestMapping("customers")
public class CustomerController {
    @Autowired
    CustomerService customerService;

    @ModelAttribute
    CustomerForm setUpForm() {
        return new CustomerForm();
    }

    @GetMapping
    String list(Model model) {
        List<Customer> customers = customerService.findAll();
        model.addAttribute("customers", customers);
        return "customers/list";
    }

    @PostMapping(path = "create")
    String create(@Validated CustomerForm form, BindingResult result,
    Model model) {
        if (result.hasErrors()) {
            return list(model);
        }
        Customer customer = new Customer();
        BeanUtils.copyProperties(form, customer);
        customerService.create(customer);
        return "redirect:/customers";
    }
}
```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

```
return "redirect:/customers";
}

@GetMapping(path = "edit", params = "form")
String editForm(@RequestParam /* (1) */ Integer id, CustomerForm form) {
    Customer customer = customerService.findOne(id);
    BeanUtils.copyProperties(customer, form); // (2)
    return "customers/edit";
}

// (3)
@PostMapping(path = "edit")
String edit(@RequestParam Integer id, @validated CustomerForm
form, BindingResult result) {
    if (result.hasErrors()) {
        return editForm(id, form);
    }
    Customer customer = new Customer();
    BeanUtils.copyProperties(form, customer);
    customer.setId(id);
    customerService.update(customer);
    return "redirect:/customers";
}

// (4)
@GetMapping(path = "edit", params = "goToTop")
String goToTop() {
    return "redirect:/customers";
}
}
```

プログラム解説

項番	説明
(1)	引数に「@RequestParam」アノテーションを付けることで、特定の「リクエスト・パラメータ」をマッピングできる。ここでは「リクエスト・パラメータ」の「id」を「Integerの引数」にマッピングする（「パラメータ名」と「引数名」を一致させる必要がある）。
(2)	「顧客編集フォーム」に「現在の顧客情報」を表示させるため、「CustomerService #findOne」で取得した「顧客情報」を「CustomerForm」にコピーする。
(3)	「edit」メソッドも同様に実装できる。 送信された「CustomerForm」を「Customer」にコピーし、「CustomerService #update」で更新処理を実施。 処理が完了したら、「一覧表示画面」にリダイレクトする。
(4)	「顧客編集」フォームから「一覧表示画面」に戻るため、「リクエスト・パラメータ」に「goToTop」を含む場合は、「一覧表示画面」にリダイレクトするメソッドを用意する。

*

「顧客編集フォーム画面」を「src/main/resources/templates/customers/edit.html」に作ります。

基本的には「新規作成」フォームと同じです。

【src/main/resources/templates/customers/edit.html】編集フォーム画面

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>顧客情報編集 </title>
    <link rel="stylesheet" type="text/css"
        href="../../static/css/style.css"
        th:href="@{/css/style.css}"/>
</head>
<body>
<form th:action="@{/customers/edit}" th:object="${customerForm}" method="post">

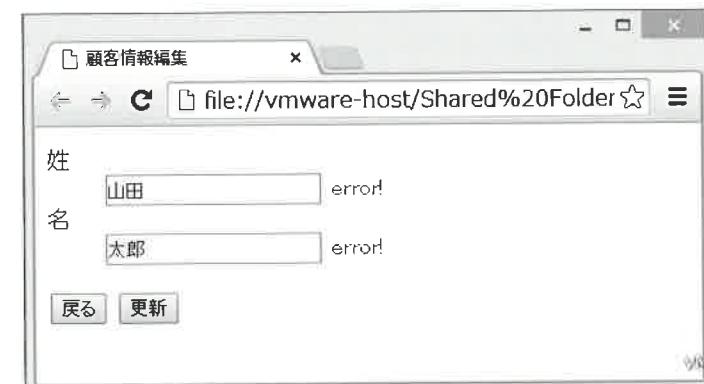
    <dl>
        <dt><label for="lastName">姓 </label></dt>
        <dd>
            <input type="text" id="lastName" name="lastName" th:field="*{lastName}" th:errorclass="error-input"
                value="山田" />
            <span th:if="#{fields.hasErrors('lastName')}" th:errors="*{lastName}"
                class="error-messages">error!</span>
        </dd>
        <dt><label for="firstName">名 </label></dt>
        <dd>
            <input type="text" id="firstName" name="firstName" th:field="*{firstName}" th:errorclass="error-input"
                value="太郎" />
            <span th:if="#{fields.hasErrors('firstName')}" th:errors="*{firstName}"
                class="error-messages">error!</span>
        </dd>
    </dl>
    <input type="submit" name="goToTop" value="戻る" /><!-- (1) -->
    <input type="hidden" name="id" th:value="${param.id[0]}" /><!-- (2) -->
    <input type="submit" value="更新" />
</form>
</body>
</html>
```

プログラム解説

項番	説明
(1)	1つの「フォーム」に複数の「送信ボタン」を置く場合は、「name」属性の値で「コントローラ」のメソッドを振り分ける。 「name="goToTop"」を設定することで、この「ボタン」を押すと「goToTop」メソッドが呼ばれる。
(2)	「顧客 ID」を「type="hidden"」の「input タグ」から送信。 「param.(パラメータ名)」で「リクエスト・パラメータ」にアクセスできる。 (アクセスしたパラメータは、「String[]」である点に注意してください)。

実行

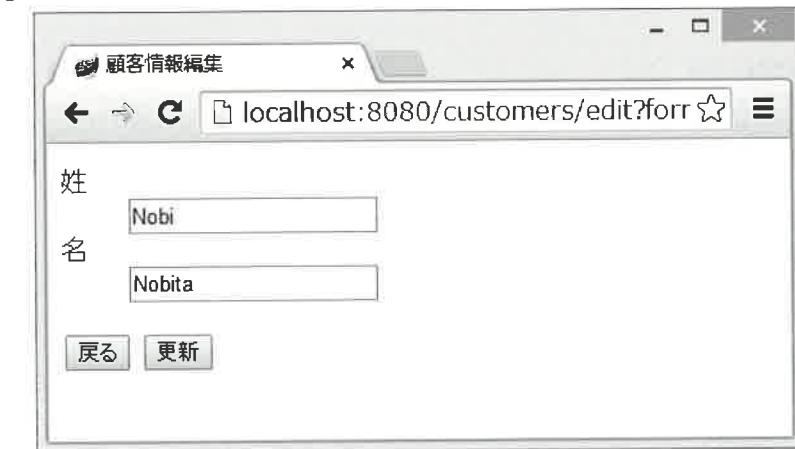
「edit.html」をブラウザで開いてみましょう。



「edit.html」を直接開く

*

次に、アプリケーションを実行して、「<http://localhost:8080/customers/edit?form&id=1>」にアクセスしましょう。



「<http://localhost:8080/customers/edit?form&id=1>」にアクセス

「id=1」である「顧客情報」が表示されています。

フォームの情報を編集して、「更新ボタン」を押し、「一覧画面」の情報が更新されていることを確認してください。

[3.3.4]

「顧客情報」の削除

同様に「削除処理」も実装します。

「/customers/delete?id={id}」で「特定の顧客」の削除処理をする「delete」メソッドを「CustomerController」に追加し、「CustomerService」の「delete」メソッドを呼び出します。

基本的には「更新処理」と同じです。

```

CustomerController クラス
package com.example.web;

import com.example.domain.Customer;
import com.example.service.CustomerService;
import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@Controller
@RequestMapping("customers")
public class CustomerController {

    @Autowired
    CustomerService customerService;

    @ModelAttribute
    CustomerForm setUpForm() {
        return new CustomerForm();
    }

    @GetMapping
    String list(Model model) {
        List<Customer> customers = customerService.findAll();
        model.addAttribute("customers", customers);
        return "customers/list";
    }

    @PostMapping(path = "create")
    String create(@Validated CustomerForm form, BindingResult result,
    Model model) {
        if (result.hasErrors()) {
            return list(model);
        }
        Customer customer = new Customer();
        BeanUtils.copyProperties(form, customer);
        customerService.create(customer);
        return "redirect:/customers";
    }

    @PostMapping(path = "edit" , params = "form")
    String editForm(@RequestParam Integer id, CustomerForm form) {
        Customer customer = customerService.findOne(id);
        BeanUtils.copyProperties(customer, form);
        return "customers/edit";
    }

    @PostMapping(path = "edit")
    String edit(@RequestParam Integer id, @Validated CustomerForm
    form, BindingResult result) {
        if (result.hasErrors()) {
            return editForm(id, form);
        }
        Customer customer = new Customer();
        BeanUtils.copyProperties(form, customer);
    }
}

```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

```

        customer.setId(id);
        customerService.update(customer);
        return "redirect:/customers";
    }

    @RequestMapping(path = "edit" , params = "goToTop")
    String goToTop() {
        return "redirect:/customers";
    }

    @PostMapping(path = "delete")
    String delete(@RequestParam Integer id) {
        customerService.delete(id);
        return "redirect:/customers";
    }
}

```

アプリケーションを再起動し、「削除」ボタンを押すことで、対象の「顧客情報」を「削除」できることを確認してください。

[3.3.5]

「CSS フレームワーク」の利用

ここで画面の「見栄え」をよくするために、「CSS フレームワーク」の「Bootstrap」[9]を使いましょう。

ここでは「WebJars」という「JavaScript」や「CSS」のライブラリを「jar 形式」にパッケージングした仕組みを使います。

「WebJars」を使うことで、「Maven」による「依存性解決」ができます。

*

「pom.xml」に以下の設定をしてください。

[pom.xml] 「Bootstrap」の依存関係の追加

```

<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.7</version>
</dependency>

```

「Spring Boot」では特別な設定をすることなく、以下のように「HTML」から「Web Jars」に梱包された「CSS」や「JS」にアクセスできます。

```

<link rel="stylesheet" type="text/css"
      th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}"/>

```

[3.3.2] の「list.html」を、「Bootstrap」を使って書き換えてみましょう。

[list.html] 「一覧画面」の編集

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>顧客一覧</title>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}"/>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap-theme/3.3.7/css/bootstrap-theme.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css}"/>
</head>
<body>
<div class="container">
    <h1>顧客管理システム</h1>
    <div class="col-sm-12">
        <form th:action="@{/customers/create}" th:object="${customerForm}" class="form-horizontal" method="post">
            <fieldset>
                <legend>顧客情報作成</legend>
                <div class="form-group" th:classappend="${#fields.hasErrors('lastName')}? 'has-error has-feedback'>
                    <label for="lastName" class="col-sm-2 control-label">姓 </label>
                    <div class="col-sm-10">
                        <input type="text" id="lastName" name="lastName" th:field="*{lastName}" class="form-control" value="山田"/>
                        <span th:if="${#fields.hasErrors('lastName')}" class="help-block">error!</span>
                    </div>
                </div>
                <div class="form-group" th:classappend="${#fields.hasErrors('firstName')}? 'has-error has-feedback'>
                    <label for="firstName" class="col-sm-2 control-label">名 </label>
                    <div class="col-sm-10">
                        <input type="text" id="firstName" name="firstName" th:field="*{firstName}" class="form-control" value="太郎"/>
                        <span th:if="${#fields.hasErrors('firstName')}" class="help-block">error!</span>
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-sm-offset-2 col-sm-10">
                        <button type="submit" class="btn btn-primary">作成 </button>
                    </div>
                </div>
            </fieldset>
        </form>
        <hr/>
        <table class="table table-striped table-bordered table-condensed">
```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

```
<tr>
    <th>ID</th>
    <th>姓</th>
    <th>名</th>
    <th colspan="2" style="text-align: center;">編集 </th>
</tr>
<tr th:each="customer : ${customers}">
    <td>${customer.id}</td>
    <td>${customer.lastName}</td>
    <td>${customer.firstName}</td>
    <td><form th:action="@{/customers/edit}" method="get">
        <input class="btn btn-default" type="submit" value="編集" />
        <input type="hidden" name="id" th:value="${customer.id}"/>
    </form></td>
    <td><form th:action="@{/customers/delete}" method="post">
        <input class="btn btn-danger" type="submit" value="削除" />
        <input type="hidden" name="id" th:value="${customer.id}"/>
    </form></td>
</tr>
</table>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"
       th:src="@{/webjars/jquery/1.11.1/jquery.min.js}"/></script>
<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
       th:src="@{/webjars/bootstrap/3.3.7/js/bootstrap.min.js}"/></script>
</body>
</html>
```

実行

ブラウザで確認してみましょう。



list.html をブラウザで直接開く

*

同様に「edit.html」にも「Bootstrap」を適用しましょう。

【edit.html】「編集フォーム画面」の編集

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>顧客情報編集</title>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}"/>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap-theme.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css}"/>
</head>
<body>
<div class="container">
    <h1>顧客管理システム</h1>
    <div class="col-sm-12">
        <form th:action="@{/customers/edit}" th:object="${customerForm}" class="form-horizontal" method="post">
            <fieldset>
                <legend>顧客情報編集</legend>
                <div class="form-group" th:classappend="#{#fields.hasErrors('lastName')}? 'has-error has-feedback'>
                    <label for="lastName" class="col-sm-2 control-label">姓</label>
                    <div class="col-sm-10">
                        <input type="text" id="lastName" name="lastName" th:field="*{lastName}"
                               class="form-control" value="山田"/>
                        <span th:if="#{#fields.hasErrors('lastName')}" th:errors="*{lastName}">
                            class="help-block">error!</span>
                    </div>
                </div>
                <div class="form-group" th:classappend="#{#fields.hasErrors('firstName')}? 'has-error has-feedback'>
                    <label for="firstName" class="col-sm-2 control-label">名</label>
                    <div class="col-sm-10">
                        <input type="text" id="firstName" name="firstName" th:field="*{firstName}"
                               class="form-control" value="太郎"/>
                        <span th:if="#{#fields.hasErrors('firstName')}" th:errors="*{firstName}">
                            class="help-block">error!</span>
                    </div>
                </div>
            </fieldset>
            <div class="form-group">
                <div class="col-sm-offset-2 col-sm-10">
                    <input type="submit" class="btn btn-default" name="goToTop" value="戻る"/>
                    <input type="hidden" name="id" th:value="${param.id[0]}"/>
                    <input type="submit" class="btn btn-primary" value="更新"/>
                </div>
            </div>
        </form>
    </div>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<script th:src="@{/webjars/jquery/1.11.1/jquery.min.js}"></script>
<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
       th:src="@{/webjars/bootstrap/3.3.7/js/bootstrap.min.js}"></script>
</body>
</html>
```

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

```
</div>
</div>
<div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
        <input type="submit" class="btn btn-default" name="goToTop" value="戻る"/>
        <input type="hidden" name="id" th:value="${param.id[0]}"/>
        <input type="submit" class="btn btn-primary" value="更新"/>
    </div>
</div>
</div>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<script th:src="@{/webjars/jquery/1.11.1/jquery.min.js}"></script>
<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
       th:src="@{/webjars/bootstrap/3.3.7/js/bootstrap.min.js}"></script>
</body>
</html>
```

実行

こちらも HTML をそのままブラウザで確認してみましょう。



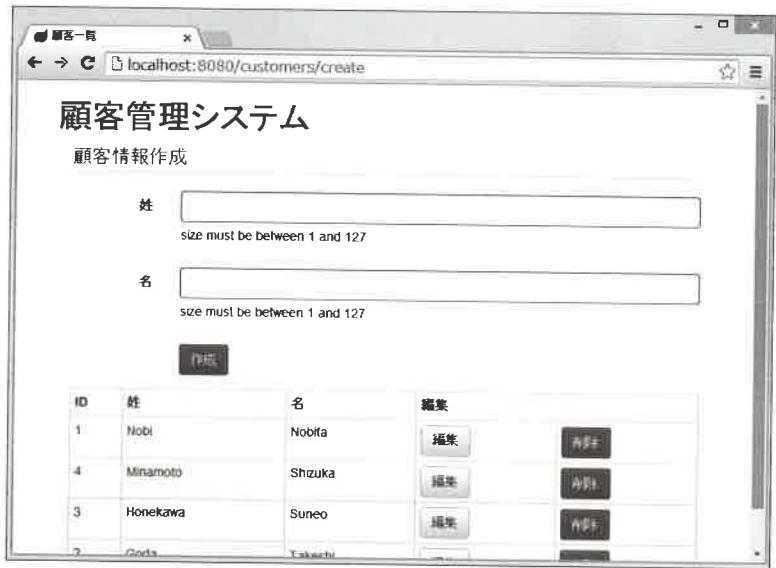
「edit.html」をブラウザで直接開く

次に、アプリケーションを起動した状態で確認しましょう。



「<http://localhost:8080/customers>」にアクセス

入力エラーがある場合は、以下のようにになります。



「入力エラー」がある場合の表示

簡単な作業で、アプリケーションの見栄えが良くなりましたね。

本書では「Bootstrap」について詳しくは扱いません。詳細を知りたい場合は、公式サイトのドキュメントが充実しているので、そちらを参照してください。

日本語の情報が欲しい場合は、拙著「はじめての Bootstrap（工学社）」^[10]をお勧めします。

[3.3] 「Thymeleaf」を使った、「画面のある Web アプリ」の開発

ノート 「list.html」と「edit.html」をそれぞれ1から作りましたが、この2つのHTMLは、「<title>タグ」を除いて、「<div class="col-sm-12">タグ」の中身が同じです。

「Thymeleaf Layout Dialect」という、「ThymeLeaf」の「レイアウト拡張機能」を使うと、「共通領域」と「個別領域」をそれぞれ作って、無駄なくテンプレートを管理できます。

以下のように、「src/main/resources/templates/layout.html」に、「共通領域」となるレイアウトを記述します。

[src/main/resources/templates/layout.html] レイアウト

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/
      layout"><!-- (1) -->
<head>
    <meta charset="UTF-8"/>
    <title>Layout</title><!-- (2) -->
    <link rel="stylesheet"
          href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}"/>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/boo
      tstrap/3.3.7/css/bootstrap-theme.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css}"/>
</head>
<body>
<div class="container">
    <h1>顧客管理システム </h1>
    <div layout:fragment="content"><!-- (3) -->
        Fake content
    </div>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jq
      uery.min.js"
          th:src="@{/webjars/jquery/1.11.1/jquery.min.js}"></script>
<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/boot
      strap.min.js"
          th:src="@{/webjars/bootstrap/3.3.7/js/bootstrap.min.js}"></script>
</body>
</html>
```

プログラム解説

項番	説明
(1)	「レイアウト拡張機能」を使うための「名前空間定義」をする。 このレイアウト拡張機能は「spring-boot-starter-thymeleaf」に含まれているため、追加で依存関係を定義する必要はない。
(2)	レイアウトのHTMLでは「<title>」に「ダミー値」を設定する。 このページで上書き可能。
(3)	コンテンツ部分を「フラグメント」(部分的HTML)で穴埋めできるように、「layout:fragment」属性に「フラグメント名」を指定する。

[10] <http://www.kohgakusha.co.jp/books/detail/978-4-7775-1799-2>

*

このレイアウトを埋める形で、「list.html」を以下のように修正します。

【list.html】「一覧画面」の「フラグメント」

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
      layout:decorator="layout"><!-- (1) -->
<head>
    <title>顧客一覧 </title>
</head>
<body>

<div layout:fragment="content" class="col-sm-12"><!-- (2) -->
    <form th:action="@{/customers/create}" th:object="${customerForm}" class="form-horizontal" method="post">
        <!-- (略) -->
    </form>
    <hr/>
    <table class="table table-striped table-bordered table-condensed">
        <!-- (略) -->
    </table>
</div>
</body>
</html>
```

プログラム解説

項番	説明
(1)	使うレイアウトの「ビューネ名」を、「layout:decorator」属性に指定。
(2)	「<head>」内に「<title>」を記述することで、レイアウトの「<title>」を置換できる。
(3)	レイアウトの「layout:fragment="content"」を埋める「フラグメント」となる領域に、同じく「layout:fragment="content"」を指定。この内容で、「layout.html」の「<div layout:fragment="content">...</div>」が置換される。

*

同様に、「edit.html」を以下のように「フラグメント化」します。

【edit.html】「編集フォーム画面」の「フラグメント」

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
      layout:decorator="layout">
<head>
    <title>顧客情報編集 </title>
</head>
<body>

<div layout:fragment="content" class="col-sm-12">
    <form th:action="@{/customers/edit}" th:object="${customerForm}" class="form-horizontal" method="post">
        <!-- (略) -->
    </form>
</div>
</body>
</html>
```

*

このように、「Thymeleaf Layout Dialect」を使うことによって、複数画面で共通する部分を重複せずに効率よく作ることができます。

ただし、テンプレートを「フラグメント化」することで、「一画面で完結したテンプレートを Web ブラウザで確認できる」という、本来の「Thymeleaf」のメリットが損なわれてしまう点に注意してください。

ノート 本節では「WebJars」で「jar」内の「Bootstrap」を利用しましたが、場合によっては、「CDN」(Contents Delivery Network)^[11]からリソースを取得したほうがいいでしょう。

3.4

「Flyway」で「DBマイグレーション」

次に「Flyway」という「DBマイグレーション・ライブラリ」を利用します。

*

「Flyway」はデータベースに「メタ情報」として、DBに適用した「SQLスクリプト」のバージョンを管理しており、新しいバージョンの「SQLスクリプト」を用意すると、現在適用中のバージョンに応じた差分のスクリプトのみを適用します。

これによって、

- 新規にアプリを使う場合は、1からSQLスクリプトが適用される
- 過去のバージョンのアプリを使っている場合は、差分のSQLスクリプトだけ適用される

という運用を簡単に行ない、アプリケーションをアップデートしていくことが可能になります。

「Flyway」で「DBマイグレーション」を行なうには、

- コマンドライン・ツール
- Maven/Gradle/Ant プラグイン
- Java API

のいずれかを用いることになります。

「Spring Boot」はアプリケーションの起動時に「Flyway」の「Java API」を自動で実行する仕組みをもちます。特定のフォルダに「SQLスクリプト」を置くだけで、アプリケーションを実行すると「DBマイグレーション」が行なわれます。

「[3.3.5]「CSSフレームワーク」の利用」で作ったアプリケーションに対して、「Flyway」を導入してみましょう。

[11] <http://cdn.jsdelivr.net/webjars/bootstrap/3.3.7/js/bootstrap.min.js>

「pom.xml」に以下の「依存関係」を追加してください。

「Flyway」のバージョンは「spring-boot-starter-parent」によって管理されているため、「<version>」の指定は不要です。

[pom.xml] 「Flyway」の依存関係の追加

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

これで「Flyway」を使う準備ができました。

*

「コマンドライン・ツール」や「Maven プラグイン」でマイグレーションを行なう場合は、「公式サイト」のドキュメント^[12]を参照してください。

[3.4.1]

SQLスクリプトの準備

「Spring Boot」ではデフォルトで「src/main/resources/db/migration」フォルダ以下の SQL スクリプトを読み込みます。

*

「src/main/resources/data.sql」は削除し、以下の 2 つの SQL ファイルを作りましょう。

①「src/main/resources/db/migration/V1_create-schema.sql」

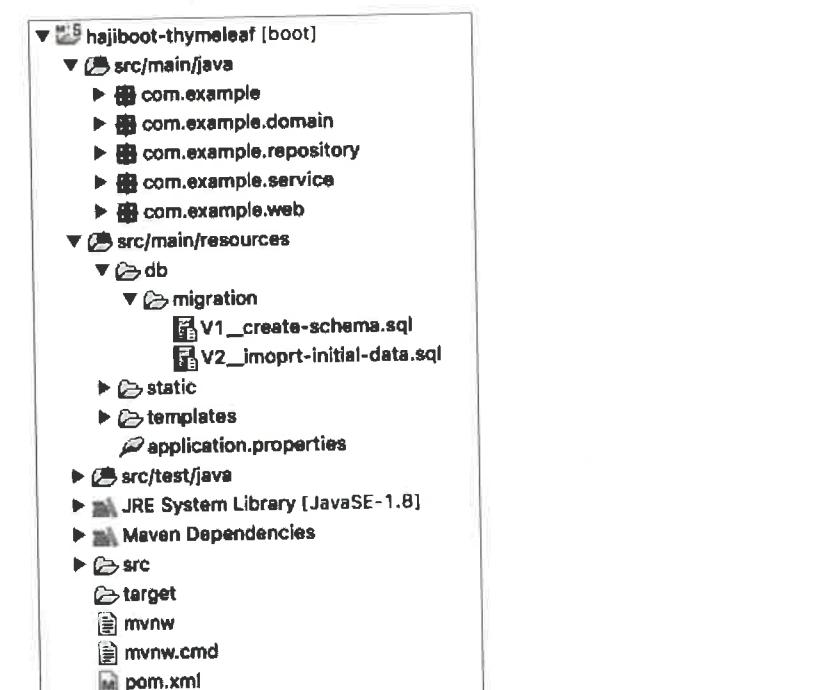
```
CREATE TABLE customers (id INT PRIMARY KEY AUTO_INCREMENT, first_name
VARCHAR(30), last_name VARCHAR(30));
```

②「src/main/resources/db/migration/V2_import-initial-data.sql」

```
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');
```

「Flyway」で管理する「SQL スクリプト」の「ファイル名」は、「V(バージョン番号)_(スクリプトの説明).sql」という形式である必要があります。

プロジェクト構成は、以下のようにになります。



プロジェクト構成

[3.4.2]

「application.properties」の変更

次に、「Flyway」に利用に合わせて、「application.properties」を、以下のように変更します。

【application.properties】「DB 設定」の変更

```
spring.datasource.driver-class-name=net.sf.log4jdb.DriverSpy
spring.datasource.url=jdbc:log4jdb:h2:file:./target/db/customer # (1)
logging.level.jdbc=OFF
logging.level.jdbc.sqltiming=DEBUG
spring.jpa.hibernate.ddl-auto=validate # (2)
```

プログラム解説

項目番	説明
(1)	H2 データベースの「ファイル・データベース」を使い、データを永続化。
(2)	これまで「DDL」の実行を「JPA」に任せていたが、今回は「Flyway」で「DDL」を実行するため、「JPA」にさせるのはスキーマの検証だけ。

[3.4.3]

DBマイグレーションの実行

セットアップが完了したら、「HajibootThymeleafApplication.java」を実行します。

起動時に、以下のようなログが表示されます。

[12] <http://flywaydb.org/>

[ターミナル] 出力されたログ

```

2016-08-12 02:48:21.771 INFO 61996 --- [           main] o.f.c.i.dbsupport.DbSupportFactory      : Database: jdbc:h2:file:///target/db/customer (H2 1.4)
2016-08-12 02:48:21.841 INFO 61996 --- [           main] o.f.core.internal.command.DbValidate      : validated 2 migrations (execution time 00:00.022s)
// (略)
2016-08-12 02:48:21.867 INFO 61996 --- [           main] o.f.c.i.metadata.DataTableMetaTableImpl    : Creating Metadata table: "PUBLIC"."schema_version"
// (略)
2016-08-12 02:48:21.910 INFO 61996 --- [           main] o.f.core.internal.command.DbMigrate        : Current version of schema "PUBLIC": << Empty Schema >>
2016-08-12 02:48:21.910 INFO 61996 --- [           main] o.f.core.internal.command.DbMigrate        : Migrating schema "PUBLIC" to version 1 - create-schema
2016-08-12 02:48:21.917 DEBUG 61996 --- [          main] jdbc.sqltiming
: org.flywaydb.core.internal.dbsupport.JdbcTemplate.executeStatement(JdbcTemplate.java:238)
9. CREATE TABLE customers (id INT PRIMARY KEY AUTO_INCREMENT, first_name VARCHAR(30), last_name VARCHAR(30)) {executed in 5 msec}
2016-08-12 02:48:21.920 DEBUG 61996 --- [          main] jdbc.sqltiming
: org.flywaydb.core.internal.dbsupport.JdbcTemplate.queryForStringList(JdbcTemplate.java:124)
10. select "version" from "PUBLIC"."schema_version" {executed in 0 msec}
2016-08-12 02:48:21.923 DEBUG 61996 --- [          main] jdbc.sqltiming
: org.flywaydb.core.internal.dbsupport.JdbcTemplate.update(JdbcTemplate.java:271)
10. UPDATE "PUBLIC"."schema_version" SET "version_rank" = "version_rank" + 1 WHERE "version_rank" >= 1 {executed in 1 msec}
// (略)
2016-08-12 02:48:21.936 INFO 61996 --- [           main] o.f.core.internal.command.DbMigrate        : Migrating schema "PUBLIC" to version 2 - imoprt-initial-data
2016-08-12 02:48:21.941 DEBUG 61996 --- [          main] jdbc.sqltiming
: org.flywaydb.core.internal.dbsupport.JdbcTemplate.executeStatement(JdbcTemplate.java:238)
9. INSERT INTO customers (first_name, last_name) VALUES ('Nobita', 'Nobi') {executed in 3 msec}
// (略)
2016-08-12 02:48:21.945 DEBUG 61996 --- [          main] jdbc.sqltiming
: org.flywaydb.core.internal.dbsupport.JdbcTemplate.update(JdbcTemplate.java:271)
10. UPDATE "PUBLIC"."schema_version" SET "version_rank" = "version_rank" + 1 WHERE "version_rank" >= 2 {executed in 0 msec}
// (略)
2016-08-12 02:48:21.953 INFO 61996 --- [           main] o.f.core.internal.command.DbMigrate        : Successfully applied 2 migrations to schema "PUBLIC" (execution time 00:00.086s).

```

バージョン管理用の「メタデータ・テーブル」が作られ、「バージョン 1」と「バージョン 2」の「SQL スクリプト」が実行されたことが分かります。

*

アプリケーションを再起動すると、

[ターミナル] 出力されたログ

```

2016-08-12 02:56:13.538 INFO 62112 --- [           main] o.f.core.internal.command.DbMigrate      : Current version of schema "PUBLIC": 2
2016-08-12 02:56:13.541 INFO 62112 --- [           main] o.f.core.internal.command.DbMigrate      : Schema "PUBLIC" is up to date. No migration necessary.

```

というログが表示されます。

現在のバージョンが「バージョン 2」であるため、「SQL」が何も適用されていないことが分かります。

これまで「インメモリ DB」を利用して DB を毎回作り直していましたが、今回「Flyway」を導入することで、永続化したデータベースに対して、継続的にスキーマを更新できる仕組みが整備されました。

次節では実際に「スキーマ」を更新して、「バージョン 3」の「SQL スクリプト」を作ります。

3.5

「Spring Security」で「認証」「認可」を追加

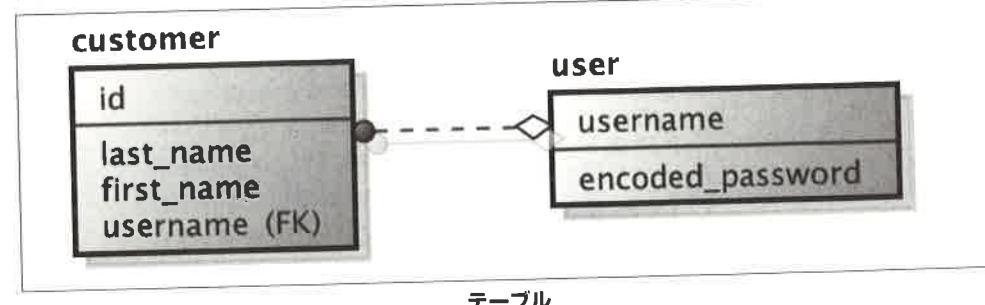
「Spring Security」はアプリケーションのセキュリティを高めるためのフレームワークであり、「認証」や「認可」が主要な機能です。

「Spring Security」は非常に多くの機能をもっています^[13]が、本書では一部の機能だけ使います。

ここまで作ってきた「顧客管理システム」に「ユーザーログイン機能」を付けましょう。

「顧客」(customer)ごとに「担当者」(user)を付け、担当者はログインして自分の顧客を登録できるようにしましょう。

「customer」と「user」が、次のような1対多の関係になるように、テーブルを追加します。



SQL の追加については、後ほど説明します。

まずはこの「データ・モデル」を用いて「Spring Security」を使いましょう。

[13] <http://docs.spring.io/spring-security/site/docs/4.1.1.RELEASE/reference/htmlsingle/>

「pom.xml」に以下の依存関係を追加してください。
おなじみの Starter の「Spring Security」版です。

[pom.xml] 「Spring Security」の Starter の追加

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

これだけで「Spring Security」が有効になり、デフォルトでアプリケーションに対して「Basic認証」が行なわれます。

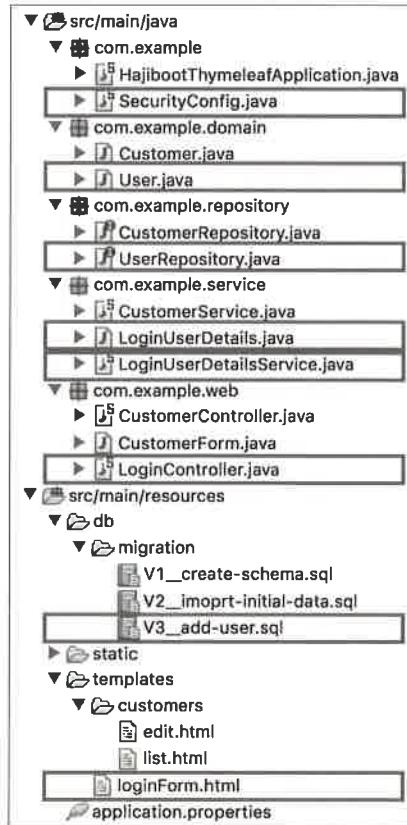
ただし、今回のケースでは、「担当者」(user)がフォームからログインするようになるため、「Basic認証」は不要です。「application.properties」に以下の設定を行ない、「Basic認証」を無効にします。

[application.properties] Basic認証を無効にする

```
security.basic.enabled=false
```

*

本節でのアプリケーション完成後のプロジェクト構成を以下に示します。新規で作るファイルを枠線で囲みました。



プロジェクト構成

[3.5] 「Spring Security」で「認証」「認可」を追加

[3.5.1] 「User」の「エンティティ」と「リポジトリ」の作成

追加した「テーブル」に合わせて、「User クラス」と「UserRepository インターフェイス」を作りましょう。

「Customer」のときと同様に、「User」の「エンティティ・クラス」を作ります。

User クラス

```
package com.example.domain;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
import java.util.List;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
@ToString(exclude = "customers") // (1)
public class User {
  @Id // (2)
  private String username;
  @JsonIgnore // (3)
  private String encodedPassword;
  @JsonIgnore // (3)
  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "user") // (4)
  private List<Customer> customers;
}
```

プログラム解説

項番	説明
(1)	後述のように、「Customer」クラスには、対応する「User」クラスのフィールドを追加。「toString」メソッドで「customers」フィールドを表示すると、「循環参照」によって「StackOverflow エラー」が発生するため、「Lombok」が生成する「toString」メソッドから「customers」フィールドの出力を除く。
(2)	「username」を「主キー」にする。
(3)	「JPA」とは関係ないが、「REST API」で「User」クラスを「JSON」出力する場合に、「パスワード・フィールド」を除外するため、「@JsonIgnore」を付ける。 なお、「パスワード」は「生」の値は使わず、「ハッシュ化」された値を格納する。
(4)	「User」と「Customer」を「1 対多」の関係にするため、「@OneToMany」アンテーションを付ける。 「cascade = CascadeType.ALL」を設定することで、「User」の「永続化操作」や「削除操作」を関連先の「Customer」にも伝播させることができる。 たとえば、「User」を削除すると、関連する「Customer」も削除される。

- (4) 「fetch = FetchType.LAZY」を設定することで、「関連するエンティティ」を「遅延ロード」させることができる（デフォルトで「LAZY」）。
 「User」エンティティを取得した際には関連する「Customer」エンティティは取得されない。
 「customers」フィールドにアクセスしたタイミングで「Customer」エンティティが取得される（「SELECT 文」が発行される）。
- 双向の関連にする場合は「mappedBy」属性に、関連先での「プロパティ名」を指定。

ここでは双向の関連にするため、「Customer」エンティティにも「User」エンティティの関連の定義をします。

Customer クラス

```
package com.example.domain;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "customers")
public class Customer {
    @Id
    @GeneratedValue
    private Integer id;
    private String firstName;
    private String lastName;
    @ManyToOne(fetch = FetchType.LAZY) // (1)
    @JoinColumn(nullable = true, name = "username") // (2)
    private User user;
}
```

プログラム解説

項番	説明
(1)	「User」と「Customer」を「多対1」の関係にするため、「@ManyToOne」アノテーションを付ける。
(2)	「@JoinColumn」アノテーションを使って、「外部キーのカラム名」を指定できる。

「UserRepository」は「Spring Data JPA」を用いて、以下のように簡単に作れます。
 「主キー」の「型」が「String」である点に注意してください。

UserRepository インターフェイス

```
package com.example.repository;
import com.example.domain.User;
import org.springframework.data.jpa.repository.JpaRepository;
public interface UserRepository extends JpaRepository<User, String> { }
```

[3.5.2]

「認証ユーザー」の作成

「Spring Security」で認証する場合、「認証ユーザー」のインターフェイスである「org.springframework.security.core.userdetails.UserDetails」と、その「認証ユーザー」を「ユーザー」名から取得するインターフェイスである「org.springframework.security.core.userdetails.UserDetailsService」を使います。

「Spring Security」には「UserDetails」と「UserDetailsService」の「デフォルト実装」が用意されています。今後の拡張性を考慮して、本書では「User」クラスと「UserRepository」クラスを用いてこれらを実装します。

*

まずは「UserDetails」の「実装クラス」である「LoginUserDetails」を作ります。

LoginUserDetails クラス

```
package com.example.service;

import com.example.domain.User;
import lombok.Data;
import org.springframework.security.core.authority.AuthorityUtils;

@Data
public class LoginUserDetails extends org.springframework.security.core.userdetails.User /* (1) */ {
    private final User user; // (2)

    public LoginUserDetails(User user) {
        super(user.getUsername(), user.getEncodedPassword(), AuthorityUtils.createAuthorityList("ROLE_USER")); // (3)
        this.user = user;
    }
}
```

プログラム解説

項番	説明
(1)	「UserDetails」の「デフォルト実装」である、「org.springframework.security.core.userdetails.User」クラスを拡張する。
(2)	「Spring Security」の「認証ユーザー」から、実際の「User」オブジェクトを取得できるように、「フィールド」を追加する。
(3)	「org.springframework.security.core.userdetails.User」クラスの「コンストラクタ」を使って、「ユーザー名」「パスワード」「認可用のロール」を設定する。 「ロール」の作成には、「AuthorityUtils」を使うと便利。

*
次に、「user テーブル」から取得した情報を用いて、「LoginUserDetails」を作る「LoginUserDetailsService」を作りましょう。

LoginUserDetailsService クラス

```
package com.example.service;

import com.example.domain.User;
import com.example.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service // (1)
public class LoginUserDetailsService implements UserDetailsService {
    @Autowired
    UserRepository userRepository; // (2)

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findOne(username); // (3)
        if (user == null) {
            throw new UsernameNotFoundException("The requested user is not found."); // (4)
        }
        return new LoginUserDetails(user); // (5)
    }
}
```

プログラム解説

項番	説明
(1)	DI コンテナに登録されている「UserDetailsService」が 1 つだけの場合、自動的にその Bean が認証時に使われる。 そのため、今回認証時に使う「LoginUserDetailsService」を DI コンテナに登録するために「@Service」アノテーションを付けて「コンポーネント・スキャン」の対象とする。
(2)	「User」オブジェクトを取得するための「UserRepository」をインジェクション。
(3)	「UserRepository#findOne」で「ユーザー名」から「User」オブジェクトを取得。
(4)	「User」オブジェクトが見つからない場合は、「findOne」メソッドは「null」を返す。この場合は、「UsernameNotFoundException」をスローする。
(5)	「User」オブジェクトが見つかった場合は、「LoginUserDetails」にラップして、返却。

[3.5.3]

「JavaConfig」による「認証」「認可」の設定

次に「Spring Security」を利用するための設定を説明します。

*

これまで通り「JavaConfig」によって設定を行ないます。ここでは「Spring Security」の設定に特化した「SecurityConfig」クラスを作ります。

SecurityConfig クラス

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.crypto.password.Pbkdf2PasswordEncoder;

@EnableWebSecurity // (1)
public class SecurityConfig extends WebSecurityConfigurerAdapter /* (2) */ {

    @Override
    public void configure(WebSecurity web) throws Exception { // (3)
        web.ignoring().antMatchers("/webjars/**", "/css/**"); // (4)
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception { // (5)
        http
            .authorizeRequests() // (6)
                .antMatchers("/loginForm").permitAll()
                .anyRequest().authenticated()
            .and()
            .formLogin() // (7)
                .loginProcessingUrl("/login")
                .loginPage("/loginForm")
                .failureUrl("/loginForm?error")
                .defaultSuccessUrl("/customers", true)
                .usernameParameter("username").passwordParameter("password")
            .and()
            .logout() // (8)
                .logoutSuccessUrl("/loginForm");
    }

    @Bean
    PasswordEncoder passwordEncoder() {
        return new Pbkdf2PasswordEncoder(); // (9)
    }
}
```

プログラム解説

項番	説明
(1)	「JavaConfig」クラスに「@EnableWebSecurity」を付けることで、「Spring Security」の基本的な設定（「認証フィルタ」の設定など）が行なわれる。
(2)	「WebSecurityConfigurerAdapter」を継承することで、「デフォルトの設定」に対して「追加したい箇所」だけ「オーバー・ライド」して設定できる。
(3)	「configure(WebSecurity)」メソッドを「オーバー・ライド」することで、特定のリクエストに対して「セキュリティ設定」を無視する設定など、全体に関わる設定ができる。
(4)	「/webjars」や「/css」といった「静的リソース」に対するアクセスには、「セキュリティ」の設定は無視するようにする。
(5)	「configure(HttpSecurity)」メソッドを「オーバー・ライド」することで、認可の設定や、ログイン・ログアウトに関する設定ができる。
(6)	認可に関する設定を行なう。 ここではログインフォームを表示する「/loginForm」には任意のユーザーがアクセスできるようにする。それ以外のパスには、認証なしでアクセスできないようにする。
(7)	「ログイン」に関する「設定」を行なう。 ここでは「フォーム認証」を有効にし、「認証処理のパス」「ログイン・フォーム表示のパス」「認証失敗時の遷移先」「認証成功時の遷移先」「ユーザー名・パスワードのパラメータ名」を設定する。
(8)	「ログアウト」に関する設定を行なう。 デフォルトでは「/logout」に対して「POST」でアクセスすることでログアウトできるようになる。
(9)	「パスワード」を「ハッシュ化」するための「PasswordEncoder」クラスの定義をする。 「PasswordEncoder」の実装を選ぶことで、「ハッシュ化アルゴリズム」を決める。「パスワードのハッシュ化」において推奨されている「PBKDF2 アルゴリズム」を実装した「Pbkdf2PasswordEncoder」を選択

[3.5] 「Spring Security」で「認証」「認可」を追加

[3.5.4]

「Service」や「Controller」の変更

「Customer」を登録する際に、担当者が「ログイン・ユーザー」の「User」になるように「CustomerService」と「CustomerController」を変更します。

*

「CustomerService」の「create」と「update」メソッドを、以下のように変更します。

CustomerService クラス

```
public Customer create(Customer customer, User user) {
    customer.setUser(user);
    return customerRepository.save(customer);
}

public Customer update(Customer customer, User user) {
    customer.setUser(user);
    return customerRepository.save(customer);
}
```

*

次に、「CustomerController」を修正し、ログイン中のユーザー情報を取得して、「CustomerService」に渡せるように変更します。

CustomerController クラス

```
@PostMapping(path = "create")
String create(@Validated CustomerForm form, BindingResult result, Model model,
             @AuthenticationPrincipal LoginUserDetails userDetails)
/* (1) */
if (result.hasErrors()) {
    return list(model);
}
Customer customer = new Customer();
BeanUtils.copyProperties(form, customer);
customerService.create(customer, userDetails.getUser()); // (2)
return "redirect:/customers";
}

// (略)

@PostMapping(path = "edit")
String edit(@RequestParam Integer id, @Validated CustomerForm form,
            BindingResult result,
            @AuthenticationPrincipal LoginUserDetails userDetails) {
    if (result.hasErrors()) {
        return editForm(id, form);
    }
    Customer customer = new Customer();
    BeanUtils.copyProperties(form, customer);
    customer.setId(id);
    customerService.update(customer, userDetails.getUser());
    return "redirect:/customers";
}
```

プログラム解説

項番	説明
(1)	コントローラの引数に「@AuthenticationPrincipal」を付けることで、ログイン中の「LoginUserDetails」オブジェクトを取得できる。
(2)	ログイン中の「LoginUserDetails」オブジェクトに格納されている「User 情報」を取り出して、「CustomerService」に渡す。

[3.5.5]

画面の追加

「認証」のための「ログイン画面」を作る必要があります。

「ログイン画面」は「SecurityConfig」クラスに設定したように、「/loginForm」でアクセスできます。

また、URL「/login」に対して「ユーザー名」「パスワード」をそれぞれ「"username」「password」を送ることで「認証処理」が行なわれます。

*

まずは、「ログイン画面」を返す「LoginController」を作ります。

LoginController クラス

```
package com.example.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {
    @GetMapping(path = "loginForm")
    String loginForm() {
        return "loginForm";
    }
}
```

*

次に「src/main/resources/templates/loginForm.html」を作ります。

[src/main/resources/templates/loginForm.html] ログイン画面

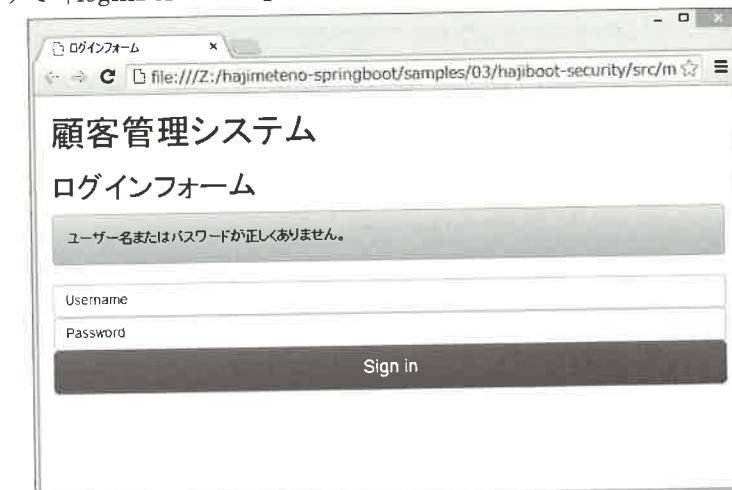
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8"/>
    <title>ログインフォーム</title>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap.min.css}"/>
    <link rel="stylesheet" href="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css"
          th:href="@{/webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css}"/>
</head>
<body>
```

```
<div class="container">
    <h1>顧客管理システム </h1>
    <form class="form-signin" method="post" th:action="@{/login}">
        <h2 class="form-signin-heading">ログインフォーム </h2>
        <div th:if="${param.error}" class="alert alert-danger"><!-- (1) -->
            ユーザー名またはパスワードが正しくありません。
        </div>
        <input type="text" class="form-control" name="username" placeholder="Username" required="required"
               autofocus="autofocus"/>
        <input type="password" class="form-control" name="password" placeholder="Password" required="required"/>
        <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
    </form>
</div>
</body>
</html>
```

プログラム解説

項番	説明
(1)	「認証エラー」が発生した場合に、「loginForm?error」に遷移するように設定したため、「error」パラメータが設定されている場合に、「エラー・メッセージ」を表示するようにする。

ブラウザで「loginForm.html」を確認しましょう。



「loginForm.html」を直接開く

*

「顧客一覧」画面で担当者の「username」が表示されるようにしましょう。

[list.html] 「一覧画面」の編集

```
<table class="table table-striped table-bordered table-condensed">
  <tr>
    <th>ID</th>
    <th>姓</th>
    <th>名</th>
    <th>担当者</th>
    <th colspan="2">編集</th>
  </tr>
  <tr th:each="customer : ${customers}">
    <td th:text="${customer.id}">100</td>
    <td th:text="${customer.lastName}">山田</td>
    <td th:text="${customer.firstName}">太郎</td>
    <td th:text="${customer.user.username}">duke</td><!-- (1) -->
    <td>
      <form th:action="@{/customers/edit}" method="get">
        <input class="btn btn-default" type="submit" name="form" value="編集"/>
        <input type="hidden" name="id" th:value="${customer.id}"/>
      </form>
    </td>
    <td>
      <form th:action="@{/customers/delete}" method="post">
        <input class="btn btn-danger" type="submit" value="削除"/>
        <input type="hidden" name="id" th:value="${customer.id}"/>
      </form>
    </td>
  </tr>
</table>
```

プログラム解説

項目番号	説明
(1)	「customer」オブジェクトがもつ担当者「user」の「username」を表示。

また、

[list.html] 「ログアウト」ボタンの追加

```
<h1>顧客管理システム</h1>
<form th:action="@{/logout}" method="post"><input type="submit" class="btn btn-default btn-xs" value="ログアウト" /></form>
```

ブラウザで確認すると、以下のように表示されます。



ブラウザで直接開く

[3.5.6]

DBスキーマの変更

最後に、既存のアプリケーションに対してDBの変更を加えるために、前節で説明した「Flyway」を利用します。

差分スクリプトを、「src/main/resources/db/migration/V3__add-user.sql」に用意しましょう。

- users テーブルを作る
- users テーブルにデータを投入する
- customers テーブルに username カラムを追加し、外部キーを設定する

を行ないます。

[src/main/resources/migration/V3__add-user.sql] DBスキーマ変更用SQL

```
CREATE TABLE users (username VARCHAR(100) NOT NULL PRIMARY KEY, encoded_password VARCHAR(255));
INSERT INTO users (username, encoded_password) VALUES ('user1', 'ce5f8d0c5790bf82e9b253d362feb51ba02853301ae24149b260bd30acb00f1b2a0d8b18bbff97a9' /*demo*/);
INSERT INTO users (username, encoded_password) VALUES ('user2', 'ce5f8d0c5790bf82e9b253d362feb51ba02853301ae24149b260bd30acb00f1b2a0d8b18bbff97a9' /*demo*/);
ALTER TABLE customers ADD username VARCHAR(100) NOT NULL DEFAULT 'user1';
ALTER TABLE customers ADD CONSTRAINT FK_CUSTOMERS_USERNAME FOREIGN KEY (username) REFERENCES users;
```

「encoded_password」には「PBKDF2」アルゴリズムで「ハッシュ化」された「パスワード」（元の文字列は「demo」）を格納します。

ノート 「PBKDF2」パスワードを生成するには、次のようなプログラムを実行します。「PBKDF2」アルゴリズムは「乱数」を使うため、一意な値にはなりません。

```
import org.springframework.security.crypto.bcrypt.  
BCryptPasswordEncoder;  
  
public class GenPassword {  
    public static void main(String[] args) {  
        System.out.printf(new Pbkdf2PasswordEncoder().  
encode("demo"));  
    }  
}
```

なお、1つのプロジェクトに複数の「main メソッド」があると、「mvn spring-boot:run」コマンドを実行する際に、プラグインがどの「main メソッド」を実行すべきか判断できなくなるため、「pom.xml」に次のように「main メソッド」をもつクラスの「FQCN」を設定する必要があります。

【pom.xml】「Spring Boot」プラグインの設定

```
<plugin>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-maven-plugin</artifactId>  
    <configuration>  
        <mainClass>com.example.HajibootThymeleafApplication</mainClass>  
    </configuration>  
</plugin>
```

これで「Spring Security」で「認証」「認可」をするための準備が完了しました。

実行

アプリケーションを起動しましょう。

【ターミナル】出力されたログ

```
2016-08-14 00:34:35.638 INFO 79544 --- [           main] o.f.core.in  
ternal.command.DbMigrate      : Current version of schema "PUBLIC": 2  
2016-08-14 00:34:35.638 INFO 79544 --- [           main] o.f.core.in  
ternal.command.DbMigrate      : Migrating schema "PUBLIC" to version  
3 - addd-user  
2016-08-14 00:34:35.652 DEBUG 79544 --- [           main] jdbc.sqlti  
ming                          : org.flywaydb.core.internal.dbsupp  
rt.JdbcTemplate.executeStatement(JdbcTemplate.java:238)  
9. CREATE TABLE users (username VARCHAR(100) NOT NULL PRIMARY KEY, en  
coded_password VARCHAR(255))  
{executed in 6 msec}  
(略)  
2016-08-14 00:34:35.800 INFO 79544 --- [           main] o.f.core.in  
ternal.command.DbMigrate      : Successfully applied 1 migration to  
schema "PUBLIC" (execution time 00:00.171s).
```

現在のバージョンが「バージョン 2」であるため、「バージョン 3」の「SQL スク
リプト」が実行されたのが分かります。

【3.5】「Spring Security」で「認証」「認可」を追加

「http://localhost:8080/customers」にアクセスすると、「未認証」状態であるため、「ログイン画面」に遷移します。



「http://localhost:8080/customers」にアクセス

*
「ユーザー名」に「user2」を入力し、「パスワード」に「demo」を入力して、ログ
インしましょう。

「顧客一覧」に「担当者名」が表示されていることを確認してください。



「担当者名」が表示されていることを確認

*
ログインした状態で「新規顧客」を作りましょう。

The screenshot shows a web browser window titled '顧客一覧' (Customer List) with the URL 'localhost:8080/customers'. The page has a header '顧客管理システム' and a sub-header '顧客情報作成'. It features a form with fields for '姓' (Last Name) and '名' (First Name), and a '作成' (Create) button. Below the form is a table listing five customers with columns: ID, 姓 (Last Name), 名 (First Name), 担当者 (Manager), and two buttons (編集, 制除). The manager column shows values like 'user1' or 'user2'. At the bottom is a link '新規顧客の作成'.

作られた顧客の「担当者名」が「ログイン・ユーザー名」になっていますね。

*

「ログアウト」して、「ログイン画面」に戻りましょう。
「ユーザー名」に「user3」を入力してログインしようとすると、「エラー・メッセージ」が表示されます。

The screenshot shows a web browser window titled 'ログインフォーム' with the URL 'localhost:8080/loginForm?error'. The page has a header '顧客管理システム' and a sub-header 'ログインフォーム'. It features a form with 'Username' and 'Password' fields, and a 'Sign in' button. A red error message 'ユーザー名またはパスワードが正しくありません。' is displayed above the form. At the bottom is a link 'エラー・メッセージ'.

ノート 「user2」でログインして、「新規顧客」を作った後に、顧客情報を一覧表示すると、以下のような「SQL ログ」が出力されます。

【コマンド・プロンプト】出力された「SQL ログ」

```
2014-09-22 19:26:13.234 DEBUG 28225 --- [nio-8080-exec-9] jdbc.sqlTiming : org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
8. select customer0_.id as id1_0_, customer0_.first_name as first_name2_0_, customer0_.last_name as last_name3_0_, customer0_.username as username4_0_ from customers
customer0_ order by customer0_.first_name,
customer0_.last_name {executed in 0 msec}
2014-09-22 19:26:13.251 DEBUG 28225 --- [nio-8080-exec-9] jdbc.sqlTiming : org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
8. select user0_.username as username1_1_0_, user0_.encoded_password as encoded_2_1_0_ from users
user0_ where user0_.username='user1' {executed in 0 msec}
2014-09-22 19:26:13.261 DEBUG 28225 --- [nio-8080-exec-9] jdbc.sqlTiming : org.hibernate.engine.jdbc.internal.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
8. select user0_.username as username1_1_0_, user0_.encoded_password as encoded_2_1_0_ from users
user0_ where user0_.username='user2' {executed in 0 msec}
```

はじめに「customers テーブル」を「SELECT」して「顧客情報」を取得し、各顧客に対して「担当者」の情報を、「users テーブル」から「SELECT」して取得しています。

「担当者」が同じ場合は、再度「SELECT」は行なわれませんが、担当者が「N 人」いた場合、全部で発行される「SQL の回数」は「N+1 回」になります。

これは「N+1 SELECT 問題」と呼ばれ、「JPA」が関連するエンティティを遅延ロードする際に起こる問題です。

今回のケースでは、顧客情報を取得後、一覧で表示するタイミングで担当者情報が必要になり、個々の担当情報が遅延ロードされています。

これがアプリケーションの性能上、本当に問題になるかどうかは、「N の大きさ」と「性能要件」によって異なります。

「1 画面」で表示する「N」が数十件ある場合は、通常、レスポンス・タイムに大きく影響が出ます。

この問題への 1 つの対処方法を紹介します。

顧客情報を取得する際に担当者情報も「JOIN」する、「JOIN FETCH」を使う手法です。

「CustomerRepository」に、以下のように、「findAllWithUserOrderByByName」というメソッドを追加し、「JOIN FETCH」を含む「JPQL」を「@Query」に記述します。

CustomerRepository インターフェイス

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
    // (略)
    @Query("SELECT DISTINCT x FROM Customer x JOIN FETCH x.user
ORDER BY x.firstName, x.lastName")
```

```
List<Customer> findAllWithUserOrderByName();
// (略)
}
```

「JOIN FETCH」の後に、「Customer」の取得と同時に取得したい「関連フィールド」を指定します。

次に、「CustomerService#findAll」を修正して、この「findAllWithUserOrderByName」メソッドを呼ぶように変更します。

CustomerService クラス

```
public class CustomerService {
    // (略)
    public List<Customer> findAll() {
        return customerRepository.findAllWithUserOrderByName();
    }
    // (略)
}
```

アプリケーションを再起動して、再び「顧客情報」を一覧表示してください。以下のような「SQL ログ」が output されます。

【コマンド・プロンプト】出力された「SQL ログ」

```
2014-09-22 19:55:48.563 DEBUG 28385 --- [nio-8080-exec-8] jdbc.sqlt
iming          : org.hibernate.engine.jdbc.intern
al.ResultSetReturnImpl.extract(ResultSetReturnImpl.java:80)
6. select distinct customer0_.id as id1_0_0_, user1_.username as us
ername1_1_1_, customer0_.first_name
as first_na2_0_0_, customer0_.last_name as last_nam3_0_0_, custom
er0_.username as username4_0_0_,
user1_.encoded_password as encoded_2_1_1_ from customers customer0_
inner join users user1_
on customer0_.username=user1_.username order by customer0_.first_
name, customer0_.last_name
{executed in 1 msec}
```

「N」(担当者数)が増えても、「SQL 発行回数」は「1 回」です。

頻繁に関連を取得することが分かっている用途に対しては、「JOIN FETCH」のクエリを実行するメソッドをリポジトリに予め用意しておくといいでしよう。

それ以外のケースでは(簡単に「JPA」を使うため)「遅延ロード」を利用し、「SQL」の発行回数増加による性能劣化が見受けられたら、改善の一環でメソッドを追加していく方針でいいと筆者は思います。

本書では「担当者」の「CRUD」の実装はしませんが、アプリケーションとしては必要な機能なので、追加課題としてぜひ実装してみてください。

[3.5.7] 「Thymeleaf」の「Spring Security」対応

「認証ユーザー」の情報を画面に表示したり、「認可情報」によって画面の表示を制御する、といった、「Spring Security」で扱う情報を、「Thymeleaf」の画面上で扱うための連携機能が用意されています。

[3.5] 「Spring Security」で「認証」「認可」を追加

この「連携機能」を使うには、以下のように「pom.xml」に「依存関係」を追加する必要があります。

【pom.xml】「Spring Security」連携機能の依存関係追加

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
```

「Flyway」と同様に、「spring-boot-starter-parent」によって「thymeleaf-extras-springsecurity4」のバージョンが管理されているため、「<version>」の指定は不要です。

*

「顧客一覧」画面に、「ログイン・ユーザー名」を表示してみましょう。

【list.html】「ログイン・ユーザー名表示」の追加

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
    <!-- (1) -->

    <!-- (略) -->
    <body>
        <div class="container">
            <h1>顧客管理システム </h1>
            <p>
                <span sec:authentication="principal.user.username">duke</span>
                さんログイン中。<!-- (2) -->
            </p>
            <form th:action="@{/logout}" method="post"><input type="submit" value="ログアウト" /></form>
        <!-- (略) -->
    </div>
    <!-- (略) -->
    </body>
</html>
```

プログラム解説

項目番号	説明
(1)	「Thymeleaf」の「Spring Security」連携機能の名前空間を設定。設定しなくとも動作するが、設定しないと IDE に警告される。
(2)	「sec:authentication」属性で認証ユーザー情報にアクセスできる。「principal」プロパティで「UserDetails」オブジェクトにアクセスできるので、「principal.user.username」で「ログイン・ユーザー名」が分かる。今回の例では「User」クラスは「username」しかもないが、「氏名」や「所属部署名」などをもたせた場合にも、同様に表示できる。

*

アプリケーションを再起動して、ログインしてみましょう。



「ログイン・ユーザー名」が表示されましたね。

「認可制御」など、ここで挙げた例以外の使い方は、ドキュメント^[14]を参照してください。

ノート 「Spring Boot 1.4」からは「Thymeleaf」の最新版である「バージョン3」がサポートされています。

デフォルトでは「バージョン2」を使っていますが、「pom.xml」のタグ内に、以下の設定を追加することで、「バージョン3」に変更できます。

```
<properties>
  <!--(略)-->
  <thymeleaf.version>3.0.1.RELEASE</thymeleaf.version>
  <thymeleaf-layout-dialect.version>2.0.1</thymeleaf-layout-dialect.version>
  <thymeleaf-extras-springsecurity4.version>3.0.0.RELEASE</thymeleaf-extras-springsecurity4.version>
</properties>
```

「Thymeleaf2」までは、デフォルトでテンプレートの「HTML」を「XHTML」形式にする必要があり、必ず閉じタグを用意しなければなりません。通常の「HTML」に対応するには、別の「ライブラリ」を追加する必要があります。

「Thymeleaf3」からは、追加ライブラリなしで通常の「HTML」を使えるようになりました。また、「HTML」以外にも「プレーンテキスト」「CSS」「JavaScript」のテンプレート・エンジンとしても使用可能になりました。

「Spring Boot」では、「Thymeleaf2」用の設定がデフォルトで使われるため、起動時に次のような WARN ログが出力されます。

```
[THYMELEAF] [restartedMain] Template Mode 'HTML5' is deprecated. Using Template Mode 'HTML' instead.
```

「Thymeleaf3」用の設定を使うために、「application.properties」に、以下のプロパティを追加してください。

```
spring.thymeleaf.mode=HTML
```

「閉じタグ」を作るのが煩わしいと感じる場合は、「Thymeleaf3」を試してみてはいかがでしょうか。

[14] <https://github.com/thymeleaf/thymeleaf-extras-springsecurity4>

PaaS「Cloud Foundry」にデプロイ

これまで作ったアプリケーションを「PaaS」(Platform as a Service)

に「デプロイ」してみましょう。

「デプロイ先」の「PaaS」として、本書では「Cloud Foundry」を利用します。

*

「Cloud Foundry」で Java アプリケーションを実行する場合、「war」を「アプリケーション・サーバ」にデプロイするのではなく、実行可能な形式にしてアプリケーションをプロセスとして起動します。

「Spring Boot」は初めからその形式になっているため、少ない手順で、簡単にデプロイできます。

「Spring Boot」は「Cloud Foundry」と相性がいいと言えるでしょう。

4.1 「PaaS」(Platform as a Service) の重要性

本書では簡単なアプリケーション開発を通じて「Spring Boot」を使ったアプリケーション開発方法を学んできました。

また実行可能な「jar」を作って、スタンドアロンなプロセスとしてアプリケーションを実行できることも学びました。

*

ただし、これまでの方法は、「1 インスタンス」での運用しか考えてられていないことに注意してください。

利用者が少なく、可用性も求められない場合はこれでもいいのですが、実際にサービスを運用する場合は、複数の「インスタンス」にスケール・アウトして、ロード・バランサによってリクエストを振り分けるようになります。

「インスタンス」が増えると、「死活監視」や「ログ・メトリクスの集約」など考えなければいけないことも増えてきます。

上記の要件に加えて、「セキュリティ」面の考慮もふまえたインフラ環境の構築には、高度なスキルと非常に多くの時間を要します。

このような環境をサービスとして提供するのが「PaaS」であり、「PaaS」を使うことで開発者はアプリケーションの開発に集中でき、新しい機能を素早くリリースすることが可能になります。

ソフトがビジネスを左右する時代になり、このリリースに対するスピードの重要度が