

付録

ここからは、本編に入りきらなかった4つの内容について説明していきます。本編に入りきらなかったとはいえ、開発現場ではよく使う内容ばかりなので、実際にできるようになる必要があります。

付録Dで取り上げるテストコードの作成は、プロジェクトによって実施されないケースが多いのが現状です。しかし、品質を向上させる意味でも、テストコードの作成を強く推奨します。本書では、実践に必要な最低限の知識について解説します。

付録A ファイル操作（ファイルのアップロード／ダウンロード） ————— p.2

付録B REST ————— p.21

付録C ビルド ————— p.80

付録D テスト（テストコードの作成） ————— p.93

付録A

ファイル操作

付録Aでは、Spring Bootでの**ファイル操作**について学習します。Webアプリケーション開発では、ファイルのアップロード／ダウンロードの機能を作ることがあります。それらの機能の作り方について学びます。

A-1

ファイルアップロード

まずは、**ファイルアップロード**の機能を作ります（図A-1）。なお、ファイルは一度に複数アップロードできるようにします。



図 A-1 ファイル一覧画面



サンプルアプリケーションの作成

サンプル ChA_File/A-1

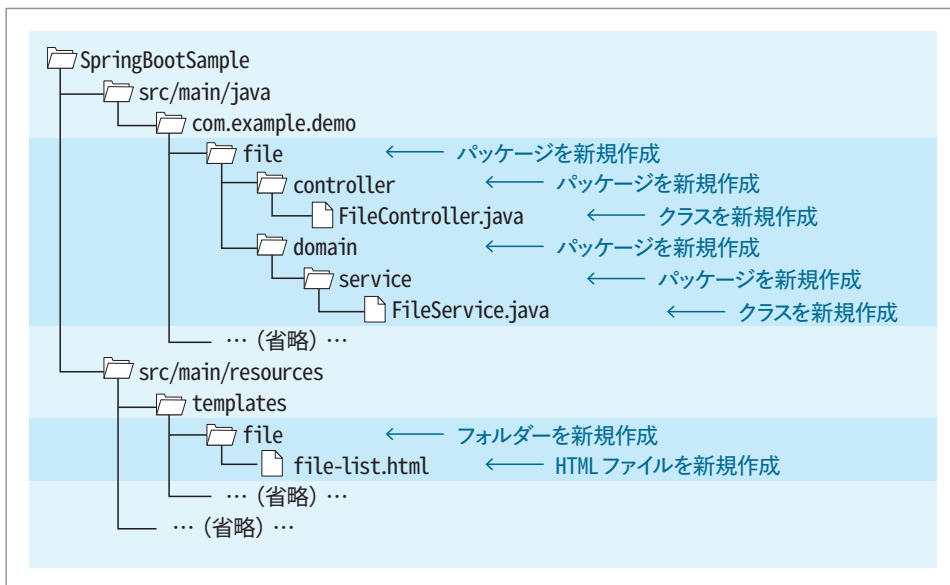
以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② ファイル操作サービスの作成

- ③ ファイル一覧画面の作成
- ④ アップロードファイルサイズの設定
- ⑤ ファイル一覧画面へのリンク作成

① ファイルなどの作成

図A-2の構成となるように、ファイルなどを作成してください。



図A-2 パッケージ・エクスプローラーの構成

◆ アップロードファイルの保存先のフォルダー

次に、アップロードファイルの保存先のフォルダーを作成します。使用しているOSによって、以下のフォルダーを作成してください。

- Windowsの場合 ➡ C: ¥work ¥spring ¥file-sample
- Macの場合 ➡ /var/spring/file-sample

以下、OSごとにフォルダーを作成する方法を説明します。

◆ ファルダーの作成——Windowsの場合

Cドライブ直下にworkフォルダーを作成します。そして、workフォルダー内にspringフォルダー、その中にfile-sampleフォルダーを作成します (図A-3)。

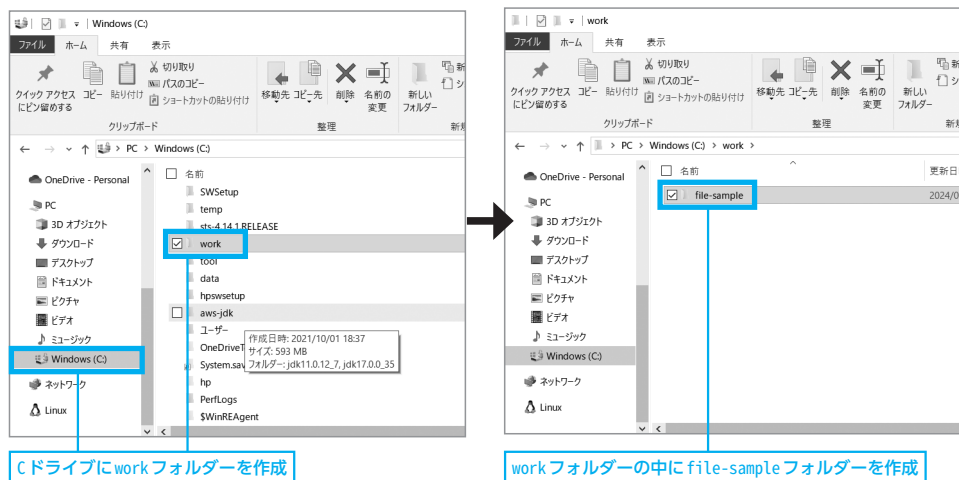


図 A-3 ファイルアップロード先フォルダーを作成する (Windows)

◆ フォルダの作成——Macの場合

Finderで`/var`を表示するためには、Finder上で`Command` + `Shift` + `G`キーを押してください。パスを入力できるため、`/var`と入力します。そして、`/var`配下に`spring`フォルダー、その中に`file-sample`フォルダーを作成してください (図 A-4) [※1]。



図 A-4 ファイルアップロード先フォルダーを作成する (Mac)

② ファイル操作サービスの作成

次に、ファイル関連の操作をするサービスを作成します。まずはファイルを保存するメソッドを作成します (リスト A-1)。

[※1] `/var`配下にフォルダーを作成したりする際に、パスワード入力などの認証が求められることがあります。

リスト A-1 FileService.java

```

package com.example.demo.file.domain.service;

import java.io.IOException;
import java.io.OutputStream;

import org.springframework.core.io.Resource;
import org.springframework.core.io.WritableResource;
import org.springframework.stereotype.Service;

import lombok.extern.slf4j.Slf4j;

@Service
@Slf4j
public class FileService {

    /** ファイル保存 */
    public void saveFile(Resource resource, byte[] bytes) throws IOException {
        WritableResource writableResource = (WritableResource) resource;
        // ファイル保存
        try (OutputStream outputStream = writableResource.getOutputStream();) {
            outputStream.write(bytes);
        } catch (IOException e) {
            log.error("FileUploadError fileName={}", resource.getFilename(), e);
            throw e;
        }
    }
}

```

ポイント ① Resource

Springでリソースを表わすのが**Resource**インターフェースです。リソースには様々な意味があります。ここでは、アプリケーションが利用する外部のデータやファイルのことです。

Resourceインターフェースを使用すると、リソースの読み書きができます。**Resource**インターフェースの実装クラスである**WritableResource**インターフェースを使用すれば、ファイルへの書き込みができます。**WritableResource**では、Javaの**OutputStream**を使用します。

③ ファイル一覧画面の作成

次に、ファイル一覧画面を作成します。

◆ コントローラーの作成

まずはコントローラーを作成します ([リストA-2](#))。このコントローラーでは、ファイル一覧画面の表示とファイルアップロードのリクエストを受け付けます。ファイルアップロードのリクエストが来た場合、先ほど作成したファイルを保存するメソッドを呼び出します。

リストA-2 FileController.java

```
package com.example.demo.file.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.Resource;
import org.springframework.core.io.ResourceLoader;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;

import com.example.demo.file.domain.service.FileService;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@Controller
@RequiredArgsConstructor
@Slf4j
@RequestMapping("/file")
public class FileController {

    private final ResourceLoader resourceLoader; — ①

    private final FileService service;

    @Value("${file.save.path}") — ②
    private String fileSavePath;
```

```

/** ファイル一覧画面表示 */
@GetMapping("/list")
public String getList() {
    return "file/file-list";
}

/** ファイルアップロード */
@PostMapping("/upload")
public String upload(@RequestParam("uploadFile") List<MultipartFile> uploadFileList) — ③
    throws Exception {
    try {
        for(MultipartFile uploadFile: uploadFileList) {
            // 保存先パス
            String resourcePath = fileSavePath + uploadFile.getOriginalFilename();
            // Resource作成
            Resource resource = this.resourceLoader.getResource(resourcePath);
            // ファイル保存
            service.saveFile(resource, uploadFile.getBytes());
        }
    } catch (Exception e) {
        log.error("FileのUploadに失敗しました");
        throw e;
    }
    return "redirect:/file/list";
}
}

```

ポイント ① ResourceLoader

ResourceLoader は、様々なリソースからファイルを取得するためのインターフェースです。通常、リソースを取得する場合、リソースの種類に応じて様々なプログラムを作る必要があります。しかし、**ResourceLoader** を使えば、その手間を省くことができます。たとえば、

- クラスパス (src/main/resource 配下のファイル)
- ファイルシステム (/var/sample 配下のファイル)
- URL (HTTP リクエストで取得したファイル)

などのリソースからファイルのデータを取得できます。

リソースを取得するためには、**getResource()** メソッドを呼び出します。このメソッドの引数には、以下のような文字列を渡します。

getResource() メソッドの書式

```
getResource("リソース文字列+ファイルパス")
```

リソース文字列は、どのリソースからファイルを取得するのかを表す文字列です。指定したリソースのどのファイルを取得するかを設定します。

たとえば、クラスパス (src/main/resource) 内のリソースを取得する場合は、次のようにします。

src/main/resources/config/application-local.ymlの取得例

```
getResource("classpath:config/application-local.yml")
```

この場合、リソース文字列として **classpath:** を付けます。classpath: が src/main/resources/ に置き換わるイメージです。

また、ファイルシステムから任意のファイルを取得する場合は、次のようにします。

/var/sample/hoge.txtの取得例

```
getResource("file:/var/sample/hoge.txt")
```

この場合、リソース文字列として **file:** を付けます。ファイルシステムでは相対パス、絶対パスのどちらでも指定できますが、上記の例では絶対パスを使用しています。

ポイント② @Value

本書のサンプルコードでは、**@Value** アノテーションを使い、ファイルの保存先パスを設定ファイルから取得しています。

FileController.java (一部抜粋)

```
@Value("${file.save.path}") ← ファイル保存先は設定ファイルから取得する
private String fileSavePath;

... (省略) ...

// 保存先パス
String resourcePath = fileSavePath + uploadFile.getOriginalFilename();
```

ポイント③ MultipartFile

HTMLでファイルをブラウザに送信するときには、どの形式のファイルを送信するかを **MIMEタイプ**で指定する必要があります。たとえば、テキストファイルを送信する場合、MIMEタイプに **text/plain**と指定します。MIMEタイプはHTMLやJavaScriptで指定し

ます。しかし、ユーザーがどのような形式のファイルを送信するか決まっていない場合もあります。そのようなとき、MIMEタイプに**マルチパート**を指定します。そうすれば、様々な形式のファイルをサーバーに送信できます。

Spring Bootでマルチパートのファイルを表わすのが**MultipartFile**インターフェースです。つまり、**MultipartFile**とは、アップロードされてきたファイルを表わすインターフェースです。コントローラーでファイルを受け取る場合、**MultipartFile**を引数に設定します。これで、送信されたファイルを受け取ることができます。

なお、複数のファイルがアップロードされる場合は、引数を**List<MultipartFile>**型にします。

◆ 画面の作成

次に、ファイル一覧画面を作成します（[リストA-3](#)）。ファイルをアップロードするリクエストを送れるようにします。

リストA-3 file-list.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorate="~{layout/layout}">
<head>
  <meta charset="UTF-8">
  <title>FileList</title>
</head>
<body>
  <div layout:fragment="content">
    <div class="container mt-5">
      <div class="border-bottom mb-3">
        <h1 class="h2">FileList</h1>
      </div>
      <!-- ファイルアップロード -->
      <div>
        <form method="POST" th:action="@{/file/upload}" enctype="multipart/form-data"> — A
          <input type="file" name="uploadFile" multiple> — B
          <input type="submit" value="アップロード">
        </form>
      </div>
    </div>
  </div>
</body>
</html>
```



▶ `enctype="multipart/form-data"`

様々な形式のファイルを送信する場合、MIMEタイプにマルチパートを指定する必要があります。[A](#)のように、`form`タグの`enctype`属性に`"multipart/form-data"`を設定すると、マルチパート形式でファイルを送信できます。マルチパート形式でファイルを送信する場合、HTMLのメソッドをPOSTにする必要があります。



▶ `multiple`属性

[B](#)のように、`input`タグの`type`属性に`"file"`を設定すると、ファイルをサーバーに送信できます。そして、`multiple`属性を付けると、一度に複数のファイルをアップロードできるようになります。

④ アップロードファイルサイズの設定

次に、アップロードファイルサイズの設定を行ないます ([リストA-4](#))。

リストA-4 `application.yml`

```
spring:
  ... (省略) ...
  # JPA
  jpa:
    hibernate:
      ddl-auto: none
  # アップロードファイルサイズ制限 — ①
  servlet:
    multipart:
      max-file-size: 10KB
      max-request-size: 100KB
  ... (省略) ...
```

ポイント① アップロードファイルサイズ制限

アップロードされるファイルサイズを制限するためには、`spring.servlet.multipart`配下のプロパティを設定します ([表A-1](#))。

これらの上限値を超えたサイズのファイルをアップロードすると、`MaxUploadSizeExceededException`が発生します。なお、どちらのプロパティもデフォルト値が設定されているため、とても大きいサイズのファイルを送られても問題ありません。

表A-1 spring.servlet.multipart配下のプロパティ

プロパティ	デフォルト値	説明
spring.servlet.multipart.max-file-size	1MB	アップロードできるファイルサイズの上限值。KB、MB単位で指定できる
spring.servlet.multipart.max-request-size	10MB	1回のリクエストで送信できるファイルの上限值。つまり、1回のリクエストで複数のファイルをアップロードする場合、複数のファイルサイズの合計値。KB、MB単位で指定できる

◆ アップロードファイルの保存先（local環境／dev環境）

第17章（設定の外部化）でも説明した通り、環境によってアップロードファイルの保存先も異なってきます。そのため、ここでは以下のような環境を想定して設定値を追加します。

- local環境 = Windows
- dev環境 = Mac (Linux)

ではまず、local環境のファイル保存先の設定値を追加します（[リストA-5](#)）。

リストA-5 application-local.yml

```
... (省略) ...

log:
  file:
    path: ${LOG_FILE_PATH:./sample-local.log}
    pattern: ${log.file.path}-%d{yyyy-MM-dd}.log.zip

file:
  save:
    path: file:C:/work/spring/file-sample/
```

同様に、dev環境の設定値を追加します（[リストA-6](#)）。

リストA-6 application-dev.yml

```
... (省略) ...

log:
  file:
    path: ${LOG_FILE_PATH:./sample-dev.log}
    pattern: ${log.file.path}-%d{yyyy-MM-dd}.log.zip

file:
  save:
    path: file:/var/spring/file-sample/
```

⑤ ファイラー一覧画面へのリンク作成

最後に、ファイラー一覧画面へのリンクをメニューに追加します（リストA-7）。

リストA-7 menu.html

```
...（省略）...
<body>
  <ul class="nav nav-pills nav-stacked" layout:fragment="menu-contents">
    <li role="presentation">
      <a class="nav-link" th:href="@{'/user/list'}">ユーザー一覧</a>
    </li>
    <li role="presentation" sec:authorize="hasRole('ADMIN')">
      <a class="nav-link" th:href="@{'/admin'}">アドミン専用</a>
    </li>
    <li role="presentation">
      <a class="nav-link" th:href="@{'/file/list'}">ファイラー一覧</a>
    </li>
  </ul>
</body>
</html>
```

実行

Spring Bootを実行して、ブラウザからログインしてください。そして、メニューからファイラー一覧画面にアクセスします（図A-5）。



図A-5 ファイラー一覧画面（図A-1再掲）

「ファイル選択」ボタンをクリックして、アップロードするファイルを選択します（図A-6）。複数ファイルを同時に選択できます。



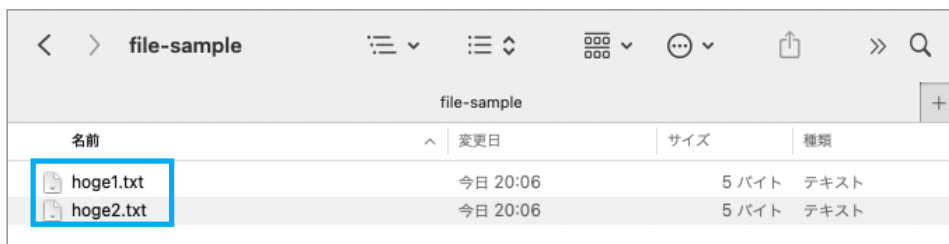
図A-6 ファイル選択

ファイル一覧画面で、ファイルを選択して「アップロード」ボタンをクリックします（図A-7）。



図A-7 ファイルアップロード

ファイルアップロードに成功した後、作成したフォルダー配下にアップロードしたファイルが保存されていることを確認できます（図A-8）。



図A-8 アップロード確認

これでファイルのアップロードができました。

A-2

ファイルダウンロード

続いて、**ファイルダウンロード**の機能を作ります。A-1 節 [p.2](#) で作成した `file-sample` フォルダーにあるファイルを一覧表示します。また、そのファイルをダウンロードする機能を追加します (図 A-9)。

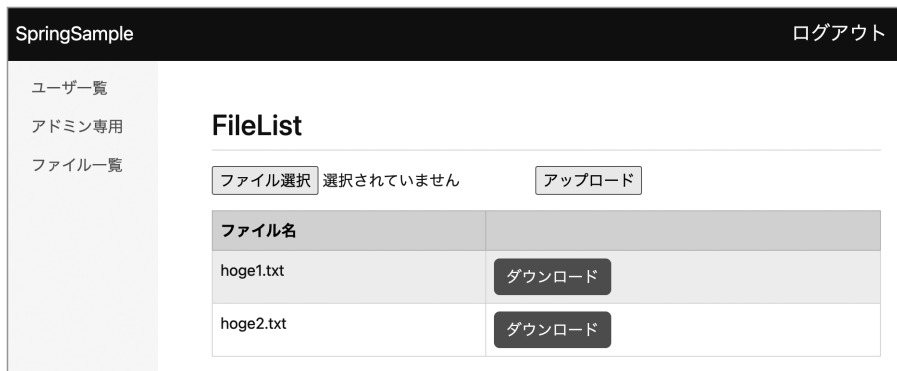


図 A-9 ファイル一覧画面——ダウンロード機能を追加後



サンプルアプリケーションの修正

サンプル ChA_File/A-2

以下の手順で、前節で作成したサンプルアプリケーションを修正します。

- 1 ファイル操作サービスの修正
- 2 ファイル一覧画面の修正

1 ファイル操作サービスの修正

まずはファイル操作サービスを修正します。以下のメソッドを追加します (リスト A-8)。

- **Resource**の配列を取得するメソッド
- 指定されたファイルの **byte** 配列を取得するメソッド 【※ 2】

【※ 2】 **byte** 配列とは、バイナリーデータの配列であり、ファイルのデータのことです。

リストA-8 FileService.java

```

... (省略) ...

import java.io.InputStream;
import org.springframework.core.io.ResourceLoader;
import org.springframework.core.io.support.ResourcePatternResolver;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
@Slf4j
public class FileService {

    private final ResourceLoader resourceLoader;

    private final ResourcePatternResolver resourcePatternResolver; — ①

    /** ファイル保存 */
    public void saveFile(Resource resource, byte[] bytes) throws IOException {
        ... (省略) ...
    }

    /** ファイル一覧取得 */
    public Resource[] getFileList(String path) throws IOException {
        // sample配下の全ファイルを取得
        Resource[] resourceArray = resourcePatternResolver.getResources(path + "**.*"); — ①
        return resourceArray;
    }

    /** ファイル取得 */
    public byte[] getFile(String filePath) throws IOException {
        // Resource取得
        Resource resource = this.resourceLoader.getResource(filePath); — A

        // byte[]取得
        try (InputStream is = resource.getInputStream()) { — B
            return is.readAllBytes();
        } catch (IOException e) {
            throw e;
        }
    }
}

```

ポイント① ResourcePatternResolver

複数のリソース（ファイル）を取得する場合、**ResourcePatternResolver**を使用します。ResourcePatternResolverの**getResources()**メソッドで、複数のリソースを取得できます。このメソッドの引数にもリソース名+ファイル名を指定します。ファイル名に正規表現を使用することで、複数のResourceを取得できます。

本書のサンプルコードでは、file-sample フォルダ配下のファイルをすべて取得しています。

このサンプルコードでは、**A** **B**の部分でResourceとbyte配列を取得しています。

Aでは、ResourceLoaderを使って読み込むファイル（Resource）を取得します。ファイルを読み込むには、ファイルのパスを指定する必要があります。

Resourceはファイルなどのリソースを表わすインターフェースであり、InputStreamを取得できます。そのため、**B**のようにInputStreamのメソッドを使ってbyte配列を取得できます。

② ファイル一覧画面の修正

続いて、ファイル一覧画面を修正していきます。

◆ コントローラーの修正

まずはコントローラーを以下のように修正します（[リストA-9](#)）。

- 画面表示のリクエスト時にファイル一覧を取得する
- ファイルダウンロードのリクエストを受け付ける

リストA-9 FileController.java

…（省略）…

```
import java.io.IOException;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.ResponseBody;
```

```
@Controller
@RequiredArgsConstructor
@Slf4j
```

```

@RequestMapping("/file")
public class FileController {

    ... (省略) ...

    /** ファイル一覧画面表示 */
    @GetMapping("/list")
    public String getList(Model model) {
        Resource[] resourceArray = null;
        try {
            // ファイル一覧取得
            resourceArray = service.getFileList(fileSavePath);
        } catch (IOException e) {
            log.error("getFileList Error", e);
        }
        // ファイル一覧をModelに登録
        model.addAttribute("fileList", resourceArray);

        return "file/file-list";
    }

    /** ファイルアップロード */
    @PostMapping("/upload")
    public String upload(@RequestParam("uploadFile") List<MultipartFile> uploadFileList)
        throws Exception {
        ... (省略)
    }

    /** ファイルダウンロード */
    @GetMapping("/download/{fileName}")
    @ResponseBody ①
    public ResponseEntity<byte[]> downloadFile(@PathVariable("fileName") String fileName) { ②

        // ダウンロードファイル取得
        byte[] bytes = null;
        try {
            String fullPath = fileSavePath + fileName;
            bytes = service.getFile(fullPath);
        } catch (IOException e) {
            log.error("getFile Error: {}", fileName, e);
            return ResponseEntity.internalServerError().body(null); ②
        }
    }
}

```

```
// HTTPヘッダーの設定
HttpHeaders header = new HttpHeaders(); —③
header.setContentDispositionFormData("filename", fileName);

return new ResponseEntity<>(bytes, header, HttpStatus.OK); —②
}
}
```

ポイント① @ResponseBody

@GetMappingや@PostMappingをメソッドに付けると、そのメソッドの戻り値を文字列（HTMLテンプレートのパス）にしなければいけません。しかし、**@ResponseBody** アノテーションをメソッドに付けると、メソッドの戻り値を任意のデータ型に変更できます。

たとえば、本書のサンプルコードのように **ResponseEntity<byte[]>** を戻り値にすることで、ファイルをダウンロードできるようになります。

ポイント② ResponseEntity

ResponseEntity は、HTTP のレスポンスを表わすクラスです。このクラスを使用することで、HTTP レスポンスを任意にカスタマイズできます。

HTTP ボディ、HTTP ヘッダー、HTTP ステータスのデータを使用して、**ResponseEntity** のインスタンスを生成します。HTTP ボディにファイルの **byte** 配列を設定することで、ファイルをダウンロードできます。

また、**internalServerError()** メソッドで、HTTP ステータス **500** を表わすインスタンスを生成できます。HTTP ステータス **500** は、サーバー側のエラーを表わすステータスです。

ポイント③ HttpHeaders

HttpHeaders は、HTTP のヘッダー情報を表わすクラスです。**add()** メソッドや **setter** で HTTP ヘッダーの項目を追加できます。HTTP ヘッダーにどのような項目があるのかは、HTTP の仕様を確認してください。

本書のサンプルコードでは、ダウンロードファイルのファイル名を設定しています。

◆ 画面の修正

最後に、ファイル一覧を表示するように画面を修正します（[リスト A-10](#)）。ファイル一覧では、ファイル名とダウンロードのリンクを表示します。

```

... (省略) ...
<body>
  <div layout:fragment="content">
    <div class="container mt-5">
      <div class="border-bottom mb-3">
        <h1 class="h2">FileList</h1>
      </div>
      <!-- ファイルアップロード -->
      <div>
        <form method="POST" th:action="@{/file/upload}" enctype="multipart/form-data">
          <input type="file" name="uploadFile" multiple/>
          <input type="submit" value="アップロード"/>
        </form>
      </div>
      <!-- 一覧表示 -->
      <table class="table table-striped table-bordered table-hover mt-3">
        <thead>
          <tr class="table-primary">
            <th>ファイル名</th>
            <th></th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="item: ${fileList}">
            <td th:text="${item.getFilename()}"></td>
            <td>
              <a class="btn btn-primary" th:href="@{'/file/download/' + ${item.getFilename()}}">
                ダウンロード
              </a>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</body>
</html>

```



Spring Bootを実行して、ブラウザからファイル一覧画面にアクセスしてください (☒ A-10)。

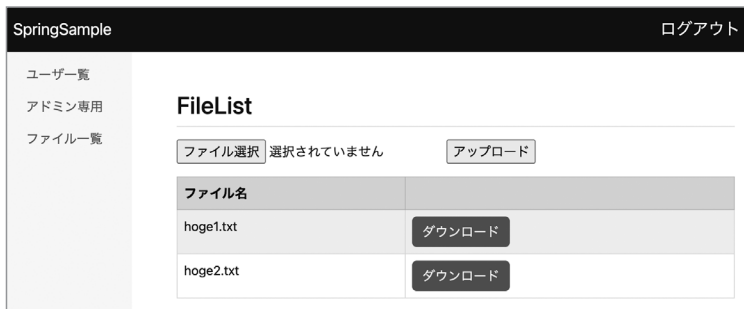


図 A-10 ファイル一覧画面——ダウンロード機能（図 A-9 再掲）

file-sample フォルダにあるファイルの一覧が表示されます。[ダウンロード] ボタンを押して、ファイルのダウンロードができることも確認してください。

これでファイル操作の学習は終了です。

A-3 まとめ

この付録で学んだ内容をまとめておきましょう。

→ ファイルアップロード

- ✓ `ResourceLoader` は、様々なリソースからファイルを取得するためのインターフェース。通常、リソースの種類に応じて色々なプログラムを作る必要があるが、`ResourceLoader` だけで様々な種類のリソースを取得できる。ここでのリソースとはアプリケーションが利用する外部のデータやファイルのことを指す。
- ✓ HTML でファイルをブラウザに送信するときには、どの形式のファイルを送信するかを MIME タイプで指定する必要がある。ただし、ユーザーがどのような形式のファイルを送信するか決まっていない場合もあり、そのようなときは MIME タイプにマルチパートを指定する。`MultipartFile` 型を使うことで、マルチパートのリクエストを処理できる。
- ✓ `Resource` はファイルを読み書きするインターフェース。ファイルへ書き込みする場合は、`WritableResource` インターフェースを使用する必要がある、`WritableResource` を使用すれば、Java の `OutputStream` を使用してファイルの書き込みができる。
- ✓ アップロードされるファイルサイズを制限するためには、`spring.servlet.multipart` 配下のプロパティを設定する。

→ ファイルダウンロード

- ✓ 複数のリソースを取得する場合、`ResourcePatternResolver` を使用する。`ResourcePatternResolver` の `getResources()` メソッドで、複数のリソース（ファイル）を取得できる。
- ✓ `@ResponseBody` アノテーションを付けると、メソッドの戻り値を任意のデータ型に変更できる。
- ✓ `ResponseEntity` クラスを使用することで、HTTP レスポンスを任意にカスタマイズできる。
- ✓ `HttpHeaders` クラスを使用することで、HTTP ヘッダーを任意にカスタマイズできる。

付録B

REST

付録Bでは、**REST**について学習します。現在、多くのWebアプリケーションでRESTが採用されているため、本付録の内容は**非常に重要**です。

まずはRESTの概要と、そのメリット・デメリットについて解説します。その後、これまでの章で作成してきた処理をREST形式へと書き換えて（REST化して）いきます。

B-1

REST とは？

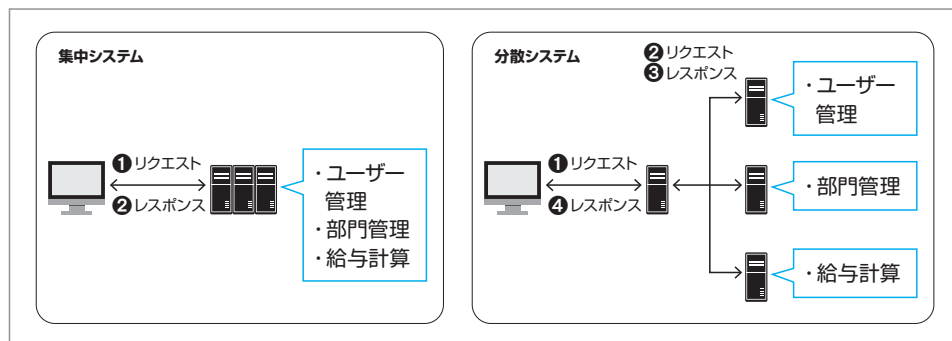
REST (Representational State Transfer) とは、シンプルで効率的な分散システムを構築するためのアーキテクチャスタイルです。特に、Webサービスに適した設計手法として広く普及しています。第3章でも説明した通り、**アーキテクチャスタイル**とは、設計のパターンやガイドラインのことです。たとえば、**MVCモデル**もアーキテクチャスタイルの1つです。

RESTの詳細に入る前に、まずは前提知識となる「分散システム」と、RESTにおいて最も重要な考え方である「リソース」について解説します。



分散システムとは？

分散システムとは、複数のコンピュータやシステムがネットワークを介して連携し、利用者からはあたかも「1つのシステム」であるかのように動作する仕組みのことです（図B-1）。対する概念として、すべての処理を1つのサーバーで完結させる**集中システム**があります。



図B-1 集中システムと分散システム

様々な観点から評価すると、どちらのシステムにも一長一短があります。開発、テスト、保守の観点では、一般的に表B-1のように評価されます。

表B-1 開発視点からの評価

項目	集中システム	分散システム
開発	○ 1つのシステムにすべての機能を組み込むため、開発しやすい	△ ●システム単位での開発になるため、シンプルな機能になり、開発しやすい ●システム間連携の開発が必要
テスト	○ 1つのシステムにすべての機能を組み込むため、テストしやすい	△ ●シンプルな機能なのでテストがしやすい ●システム間連携のテストが必要
保守	× システムが巨大であればあるほど、修正やリリースが大変になる	○ ●システム単位での保守が可能 ●機能の追加、リリースが容易

これは、それぞれのシステムの主な傾向を簡略化して比較したのですが、初期開発のしやすさという点では集中システムに軍配が上がります。しかし、システムが大規模になるほど、集中システムの保守は極めて困難になります。一部の機能を修正するだけでも、他のすべての機能に影響がないかを調査・テストしなければならないからです。

一方、分散システムは、システム間の連携に関する開発やテストが必要なため、初期段階では複雑な側面があります。しかし、一度構築してしまえばその後の保守は容易になります。機能を修正しても他の部分への影響範囲が限定的だからです。また、新しい機能の追加も比較的スムーズに行なえます。

小規模なシステムであれば集中システムが適していますが、中規模以上のシステムであれば分散システムでの構築が現実的でしょう。現在は、ビジネスの競争力を高めるために、機能の修正や追加を迅速かつ頻繁に行なうことが求められています。これこそが分散システムを採用する（つまり、RESTを活用する）最大のメリットと言えます。

そのため、近年では「マイクロサービス」に代表される、分散システムによる構築が主流となっています。



RESTにおけるリソースとは？

RESTでは、**リソース**に対して操作を行いません。ここで言うリソースとは、Webサービスが取り扱う「情報」や「対象」のことです。たとえば、ユーザー管理機能であれば、個々の「ユーザー」がリソースになります。



RESTの原則

分散システムとリソースについて理解できたところで、RESTの設計原則について説明します。RESTにはいくつかの原則がありますが、その中でも代表的なものを以下に挙げます。

◆ アドレス可能性 (Addressability)

すべてのリソースがURI (Uniform Resource Identifier) を持ち、一意に指し示せることです。たとえば、`http://example.com/user/1`というURIにアクセスすれば、ID=1のユーザー情報へ確実に到達できることを意味します。

◆ ステートレス性 (Stateless)

サーバー側でクライアントの状態（セッションなど）を保持せず、1つのリクエストに必要な情報をすべて含めることです。これにより、各リクエストの独立性が高まります。また、サーバーがセッション管理から解放されるため、大量のリクエストを効率的に処理できるようになります。

◆ 統一インターフェース (Uniform Interface)

リソース操作のための統一されたインターフェースを提供することです。具体例には、以下のようにHTTPメソッドを使い分けます。

- GET → リソースの取得
- POST → 新しいリソースの作成
- PUT → 既存リソースの更新
- DELETE → リソースの削除

上記以外にも、クライアントとサーバーの分離、キャッシュ、階層化システム、コードオンデマンド（オプション）という原則もあります。

これらの原則に従うと、シンプルで柔軟性の高い分散システムを構築できるようになります。これらの原則に従ったシステムのことを**RESTful**（レストフル）と言います。なお、RESTの原則にすべて従う必要はなく、システムの特성에応じて最適な原則を選択して適用します。

狭義の意味のREST

RESTには、**広義**と**狭義**の2つの意味があります。

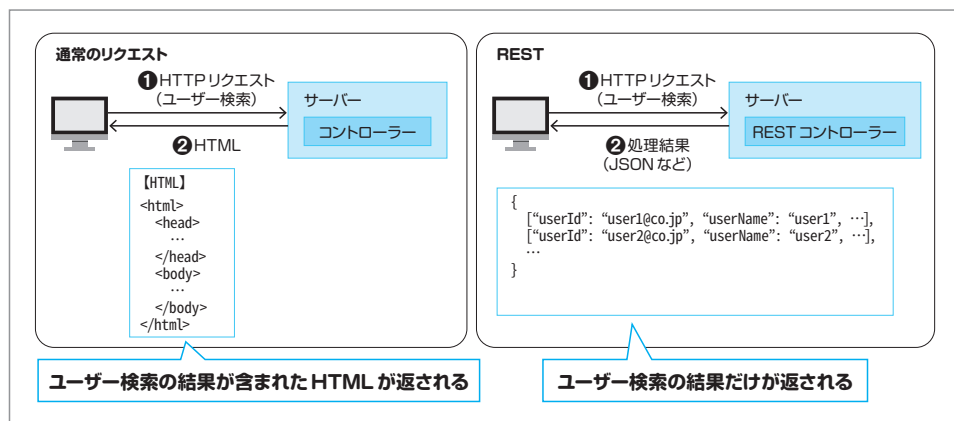
- 広義の意味でのREST → 先ほど説明した、分散システムのためのアーキテクチャスタイル（設計指針）そのもの
- 狭義の意味でのREST → HTTPプロトコルを用いたWeb APIのこと

広義のRESTは設計のガイドラインであるため、理論上はHTTP以外のプロトコルでも成り立ちますが、実際にはHTTPを使ったWeb APIでリソースを操作するのが一般的です。そのため、開発現場で「REST」と言う場合は、この狭義の意味（つまりWeb APIそのもの）を指すことが多いのが実状です。

Web APIとは？

通常、HTTPリクエストを送ると、**HTML**が返ってきます。これまで作ってきたWebアプリケーションではボタンを押すと、HTTPリクエストがサーバーに送られ、コントローラーが処理結果を含むHTMLを返していました。そのHTMLをブラウザが解釈して、画面として表示してくれます。

一方、Web APIでは、HTTPリクエストを送ると、**データ（情報や処理結果）**そのものが返ってきます。これらのデータは、一般的にJSONやXMLなどの形式でやり取りされます（図B-2）。



図B-2 Web APIの仕組み



JSON

JSON（JavaScript Object Notation）とは、JavaScriptでオブジェクトを記述するためのデータ形式です。JSONは軽量で読み書きが容易なため、プログラミング言語を問わず広く利用されています。特に、Webアプリケーションでは、サーバーとクライアント間でデータをやり取りするための標準的な形式としてJSONが使用されています。

Web APIを使う場合、HTTPリクエストを送信するプログラムをJavaScriptで実装するのが一般的です。また、返ってきたJSONデータを解析して画面に反映させる処理もJavaScriptで記述します。

つまり、Web APIを採用すると、従来の開発手法に比べてフロントエンド（JavaScript）側の開発工数が増えることになります。わざわざWeb APIを作らなくてもアプリケーションは構築できますが、そのコストを差し引いても余りある大きなメリットがWeb APIにはあります。



Web APIのメリット

Web API（狭義の意味のREST）には、様々なメリットがあります。代表的なメリットは以下の通りです。

- ① 再利用性の向上
- ② 非同期通信による操作性（ユーザビリティ）の向上

① 再利用性の向上

Web APIを利用しない場合、サーバーからのレスポンスは**HTML**となるため、その内容は特定の画面表示にしか使えません。しかし、Web APIであれば、**データ（処理結果）**だけが返ってくるため、以下のようなケースでWeb APIを再利用可能です。

- **他システムからの利用** ➡ 既存のシステムから特定の機能（Web API）を呼び出すことが可能になる
- **マルチデバイス対応** ➡ PCのブラウザだけでなく、スマートフォンのネイティブアプリケーションからも同じWeb APIを利用できる

たとえば、ユーザー管理の機能を複数のシステムで使う必要があるとします。Web APIを使用していない場合、それぞれのシステムでユーザー管理機能を作らなければいけません。

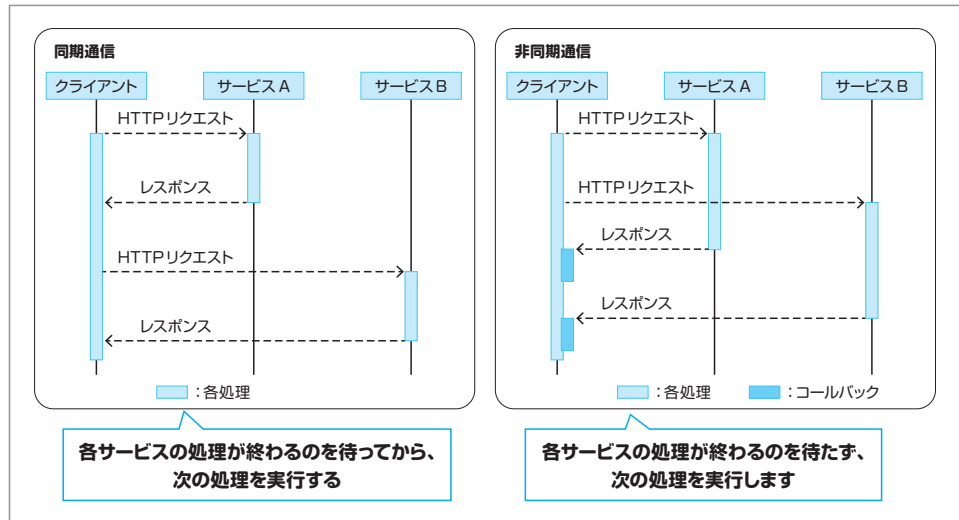
一方、Web APIでユーザー管理機能を用意していれば、HTTPのリクエストを送るだけで、その機能を利用できるようになります。また、ユーザー管理機能を一元管理できるため、開発や保守の効率が大幅に向上します。

② 非同期通信による操作性（ユーザビリティ）の向上

まずは、同期通信と非同期通信について説明します。

- **同期通信** ➡ リクエストを送ってから処理が終わり、結果が返ってくるまで、次の処理へ進まずに待つ通信方法
- **非同期通信** ➡ リクエストを送ったあとに処理の完了を待たず、すぐに次の処理へ進むことができる通信方法

たとえば、クライアントから1つのメソッド内で2つのHTTPリクエストをサーバーに送る場合、図B-3のような動きになります。



図B-3 同期通信と非同期通信

◆ 同期通信の特徴

同期通信は、仕組みがシンプルなため実装が容易です。しかし、時間のかかる処理にリクエストを送ってしまうと、処理が完了するまで待たなければならず、そのあいだは次の処理に進むことができません。つまり、同期通信が完了するまで、ユーザーは画面を操作できなくなります。

同期通信の例としては、本書でこれまで作成してきた「コントローラーへのリクエスト（画面遷移）」があります。リクエストを送信し、サーバー側で処理がすべて完了してから、新しい画面が表示される仕組みです。

◆ 非同期通信の特徴

一方、**非同期通信**は時間のかかる処理にリクエストを送っても、その完了を待ちません。そのため、すぐに次の処理を行なうことができ、サーバーからの結果を待っているあいだも、ユーザーは画面を自由に操作することが可能です。ただし、処理結果が返ってきた際の挙動を別途プログラミングする必要があるため、開発量は少し増えます。

この「結果が返ってきたときに行なう処理」のことを**コールバック**と言います。相手に電話をかけた際、相手が不在で出られない場合に折り返し電話（コールバック）をもらう仕組みになぞらえて、このように呼ばれるようになりました。

非同期通信の身近な例は、Google Mapなどの地図アプリケーションです。地図をドラッグしてスライドさせると、裏側で表示したい場所の情報を次々と取得し、準備ができた部分から順に地図が表示されていきます。情報を取得している最中も、ユーザーは地図を動かし続けることができます。

つまり、Web APIを活用して非同期通信を取り入れることで、アプリケーションのユーザービリティを大きく向上させることができるのです。

RESTのまとめ

説明が長くなったため、ここでRESTの要点を整理しておきましょう。

広義の意味のREST

シンプルで柔軟な分散システムを構築するためのアーキテクチャスタイル（設計指針）です。その原則に従ったシステムをRESTfulと言います。RESTfulなシステムには、以下のようなメリット・デメリットがあります。

- **メリット**
 - システム間連携がスムーズになり、シンプルで柔軟な分散システムを構築しやすい。
 - 分散システムでは機能の修正・追加がしやすいため、ビジネスの競争力が向上する。
- **デメリット**
 - システム間連携の機能を開発・テストしなければならない。

狭義の意味のREST

狭義の意味のREST（つまりWeb API）を使うと、以下のようなメリット・デメリットがあります。

● メリット

- 共通する機能を複数のシステムで作る必要がなくなるため、機能の開発・保守が容易になる。
- 非同期通信を行なうことで、ユーザーの操作性（ユーザビリティ）が上がる。

● デメリット

- リクエストの送信や、レスポンスを画面に反映するJavaScriptの開発が必要になる。

実際、Webアプリケーションでは画面表示以外のリクエスト（登録・更新・削除）はすべてRESTで行なうことが多いです。また、画面表示においてもRESTを使うことがあります。どのようにRESTを使うかは、これから実際に作ってみましょう。

B-2

RESTのHTTPメソッドとリクエストの送信

ここからは、実際にREST（Web API）を作っていきます。これ以降、単に「REST」と書く場合は、すべて狭義のRESTであるWeb APIを指すものとします。

RESTは、HTTPリクエストのメソッド（GETやPOSTなど）によって、どのような処理を行なうかが定義されています（表B-2）。

表B-2 HTTPリクエストのメソッドと処理内容

HTTPリクエストのメソッド	処理の内容
GET	リソースの取得（読み取り）
POST	新しいリソースの作成（登録）
PUT	既存リソースの更新
DELETE	リソースの削除

これ以外にもHTTPリクエストのメソッドはありますが、実務では表B-2の4つが最も一般的に使われます。

画面からRESTのリクエストを送るには、JavaScriptを使用します。JavaScriptを使うことで、画面全体を書き換える（画面遷移する）ことなく、HTMLの一部だけを書き換えたり、ボタンが押されたタイミングで裏側から任意のプログラムを実行したりできます。

ただし、JavaScriptをそのまま記述するのは手間がかかるため、通常はライブラリやフレームワークを利用します。かつてはjQueryを使うのが主流でしたが、現在では様々なモダンなフレームワークが広く利用されています。本書はJavaScriptの専門書ではないため、扱いがシンプルで学習コストの低いjQueryを採用して進めていきます。



JavaScriptのフレームワーク

最近ではVue.jsやReactといったフレームワークが主流となっています。しかし、jQueryも依然として現役で使われています。特に、企業内で利用されるバックエンドシステム（業務システム）などでは、Vue.jsなどが登場する以前から運用されているケースが多く、現在でもjQueryが広く利用されています。

B-3

RESTの実装（1）—— GETメソッド

まずは、RESTを用いてGETの処理を作成します。具体的には、サーバーに対してGETリクエストを送ると、データベースからすべての部署データを取得して返すREST（Web API）を実装します。

このREST APIを利用して、ユーザー登録画面で部署一覧を動的に取得し、セレクトボックス（プルダウン）から部署を選択できるように修正します（図B-4）。

ユーザー登録

ユーザーID

パスワード

ユーザー名

誕生日
YYYY/MM/DD

年齢

性別 ☐ 男性 ☐ 女性

部署

システム管理部
営業部

ログインはこちら

図B-4 ユーザー登録画面



サンプルアプリケーションの作成

サンプル ChB_REST/B-3

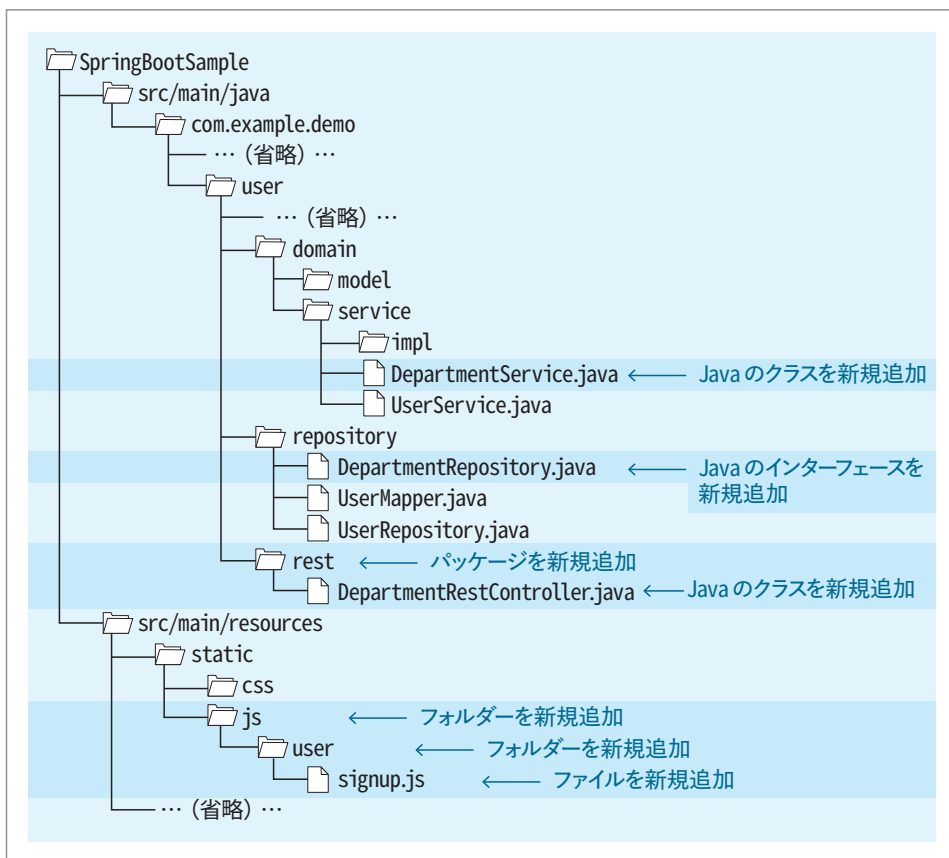
以下の手順で、部署一覧を取得するREST APIを組み込んだサンプルアプリケーションを作成・修正していきます。

① ファイルなどの作成

- ② jQueryのダウンロード
- ③ 部署一覧を取得するREST APIの作成
- ④ セキュリティ設定
- ⑤ ユーザー登録画面（HTML）の修正
- ⑥ jQueryによるHTTPリクエストの実装

① ファイルなどの作成

図B-5の構成となるように、ファイルなどを作成してください。



図B-5 パッケージ・エクスプローラーの構成

② jQueryのダウンロード

まずはjQueryのライブラリをダウンロードします。pom.xmlのdependenciesタグ内にリストB-1のコードを追加してください。

リストB-1 pom.xml

```

<dependencies>
  ...
  <!-- Thymeleaf拡張ライブラリ (セキュリティ) -->
  <dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
  </dependency>
  <!-- jQuery -->
  <dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.7.1</version>
  </dependency>
  ...
</dependencies>

```

③ 部署一覧を取得するREST APIの作成

ここから、サーバー側のREST処理を作成します。REST APIの作成といっても、途中までは通常の処理を作成する場合と同じです。まずは、部署のリポジトリを作成します。JPAを使って、SQLを自動生成します (リストB-2)。

リストB-2 DepartmentRepository.java

```

package com.example.demo.user.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.user.domain.model.Department;

public interface DepartmentRepository extends JpaRepository<Department, Integer> {

}

```

次に、サービスを作成します (リストB-3)。部署のリポジトリを使用して、部署の一覧(List) を取得します。

リストB-3 DepartmentService.java

```

package com.example.demo.user.domain.service;

import java.util.List;

import org.springframework.stereotype.Service;

```

```

import com.example.demo.user.domain.model.Department;
import com.example.demo.user.repository.DepartmentRepository;

import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class DepartmentService {

    private final DepartmentRepository repository;

    /** 部署一覧の取得 */
    public List<Department> getDepartmentList() {
        return repository.findAll();
    }
}

```

続いて、コントローラーを作成します（[リストB-4](#)）。REST APIを作成するためには、REST用のコントローラーを作成します。

リストB-4 DepartmentRestController.java

```

package com.example.demo.user.rest;

import java.util.List;

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

import com.example.demo.user.domain.model.Department;
import com.example.demo.user.domain.service.DepartmentService;

import lombok.RequiredArgsConstructor;

@RestController —①
@RequiredArgsConstructor
@RequestMapping("/department/rest")
public class DepartmentRestController {

    private final DepartmentService service;

    @GetMapping("/list")
    public List<Department> getDepartmentList() {
        return service.getDepartmentList();
    }
}

```

ポイント ① @RestController

@RestController アノテーションをクラスに付けると、**@GetMapping**などのアノテーションが付いている全メソッドの戻り値が、そのままHTTPレスポンスのボディとして返されるようになります。付録A（A-2節）で使用した**@ResponseBody**アノテーションをすべてのメソッドに付けるようなものです。

なお、デフォルトではJSON形式のデータが戻り値になります。上記のサンプルコードの場合、**List<Department>**をJSON形式に変換したデータが戻り値になります。

@RestControllerでは**@Controller**アノテーションと同様に、**@RequestMapping**や**@GetMapping**などのアノテーションで、HTTPリクエストのURLや処理内容を設定できます。

4 セキュリティ設定

次に、セキュリティの設定を行ないます。ユーザー登録画面は、ログインしなくても誰でもアクセスできる画面に設定していました。そのため、登録画面から呼び出される「部署一覧を取得するREST」のリクエスト先のURLも、同様に誰でもアクセスできるように設定を追加します（リストB-5）。

リストB-5 SecurityConfig.java

```
... (省略) ...

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    ... (省略) ...

    /** セキュリティの対象外を設定 */
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        // セキュリティ対象外の設定
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                .requestMatchers("/login").permitAll()
                .requestMatchers("/user/signup").permitAll()
                .requestMatchers("/error").permitAll()
                .requestMatchers("/department/**").permitAll()
                .requestMatchers("/h2-console/**").permitAll()
```

```

        .requestMatchers("/admin").hasAuthority("ROLE_ADMIN")
        .anyRequest().authenticated()
    ).formLogin(login -> login
        ... (省略) ...
    ).logout(logout -> logout
        ... (省略) ...
    ).rememberMe(remember -> remember
        ... (省略) ...
    );

    ... (省略) ...
}

... (省略) ...
}

```

⑤ ユーザー登録画面 (HTML) の修正

次に、ユーザー登録画面をリストB-6のように修正します。

リストB-6 signup.html

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- CSS読み込み -->
    <link rel="stylesheet" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}">
    <link rel="stylesheet" th:href="@{/css/signup.css}">
    <!-- JS読み込み -->
    <script th:src="@{/webjars/bootstrap/js/bootstrap.bundle.min.js}" defer></script>
    <script th:src="@{/webjars/jquery/jquery.min.js}" defer></script>
    <script th:src="@{/js/user/signup.js}" defer></script>
    <title th:text="#{user.signup.title}"></title>
</head>
<body class="bg-light">
    <form id="signup-form" method="post" action="/user/signup" class="form-signup mt-5" ➡
th:object="${signupForm}">
        <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
        <h2 class="text-center" th:text="#{user.signup.title}"></h2>
        <!-- ユーザーID -->
        <div class="form-group">
            ... (省略) ...

```

```

</div>

... (省略) ...

<!-- 性別 -->
<div class="form-group mt-2">
  ... (省略) ...
</div>

<!-- 部署 -->
<div class="form-group"> — B
  <label for="departmentId">部署</label>
  <select id="departmentId" name="departmentId" class="form-select">
    <option></option>
  </select>
</div>

<!-- エラーメッセージ -->
<p th:if="${errorMessage != null}" class="text-danger mt-3" th:text="${errorMessage}"></p>
<!-- 登録ボタン -->
... (省略) ...
</form>
</body>
</html>

```

Aのように、ユーザー登録画面でjQueryとユーザー登録画面用のJavaScript (signup.js) を読み込みます。必ずjQueryを先に読み込むように設定してください。そうでないと、signup.js内でjQueryの機能を使用できないからです。

Bでは、部署を選択するためのセレクトボックスを追加しています。この時点では中身は空ですが、のちほどJavaScriptを使ってREST APIから取得したデータをここに設定 (追加) します。

⑥ jQueryによるHTTPリクエストの実装

最後にjQueryを使用して、部署一覧を取得するHTTPリクエストを送信します ([リストB-7](#))。

リストB-7 signup.js

```

'use strict';

/** 画面ロード時の処理. */
$(function() { — ①
  // 部署一覧取得
  getDepartmentList();
});

```

```

/** 部署一覧取得 */
function getDepartmentList() {
    // Ajax通信
    $.ajax({ ②
        type : 'GET',
        url : '/department/rest/list',
        cache : false,
        timeout : 5000,
    }).done(function(data) {
        // Ajax成功時の処理
        console.log(data);
        // JSONからselectタグ内のoptionを生成
        for(var i in data) {
            var departmentId = data[i].departmentId;
            var departmentName = data[i].departmentName;
            // selectタグに追加するHTML
            var optionString = '<option value="' + departmentId + '>' + departmentName + '</option>';
            // id=departmentのタグ内にoptionStringを追加
            $('#departmentId').append(optionString);
        }
    }).fail(function(jqXHR, textStatus, errorThrown) {
        // Ajax失敗時の処理
        alert('部署一覧の取得に失敗しました');
    }).always(function() {
        // 常に実行する処理（特になし）
    });
}

```

ポイント① 画面ロード時の処理

JavaScriptでは、HTMLの画面がロードされたあとに処理を実行することができます。
jQueryでは、以下のように記述します。

```

$(function(){
    ... (画面ロード時に行ないたい処理) ...
})

```


ポイント② Ajax

Ajax（エイジャックス）とは、JavaScriptで非同期通信を行なうための仕組みのことです。jQueryでは、`$.ajax()`メソッドを呼び出すことで、非同期通信を行なうことができます。

非同期通信では、処理結果を受け取ったあとに実行する処理（コールバック関数）を定義する必要があります。そのため、`done()`や`fail()`などのメソッドを使って、処理内容を記述します。`$.ajax()`メソッドは大きく以下のような構造になっています。

```
$.ajax({  
    ...ajax()メソッドのプロパティ（HTTPリクエストの設定）...  
}).done(function(data) {  
    ...リクエスト成功した場合の処理...  
}).fail(function(jqXHR, textStatus, errorThrown) {  
    ...リクエスト失敗した場合の処理...  
}).always(function() {  
    ...リクエストが成功しても失敗しても行なう処理...  
});
```

`ajax()`メソッドのプロパティでは、HTTPリクエストの内容を設定します。サンプルコードの`type : 'GET'`などの部分のことです。`ajax()`メソッドでは、表B-3のプロパティがよく使われます。

表B-3 `ajax()`メソッドのプロパティ

<code>ajax()</code> メソッドのプロパティ	説明
<code>type</code>	HTTPリクエストのメソッド（GET、POST）などを設定する
<code>url</code>	リクエスト先のURLを設定する
<code>data</code>	HTTPボディで送るデータを設定する
<code>cache</code>	レスポンスをキャッシュするかどうかを設定する。HTTPリクエストのメソッドがHEAD、GETの場合のみ使用できる（HEADはレスポンスのボディを取得せず、ヘッダー情報のみを取得するためのメソッド）
<code>timeout</code>	リクエストタイムアウトをミリ秒単位で設定する
<code>headers</code>	HTTPヘッダーを設定する

`data`プロパティと`headers`プロパティについては、このあとで使う際に解説します。他にもプロパティが多数あるため、jQueryの公式サイトなどで確認してください。

- **jQuery API Documentation**

<https://api.jquery.com/jquery.ajax/>

なお、本書のサンプルコードでは、HTTPリクエストが成功した場合、部署一覧のセレクトボックスの選択肢を追加しています。

Spring Bootを実行して、ブラウザからユーザー登録画面にアクセスしてください。

<http://localhost:8080/user/signup>

ユーザー登録画面で部署を選択できるようになります（図B-6）。

部署一覧をRESTで取得できると、色々な画面や他のアプリケーションからも部署の一覧を取得できるようになります。なお、部署の選択ができるだけで、まだ選択した部署の登録はできません。このあとで選択した部署を登録できるように修正します。



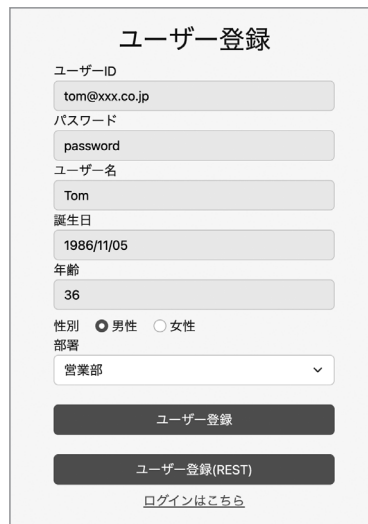
図B-6 ユーザー登録画面（図B-4再掲）

B-4

RESTの実装（2）—— POSTメソッド

次は、ユーザー登録をRESTで作成してみます。登録処理なので、HTTPリクエストのPOSTメソッドを使います。ユーザー登録画面に、RESTでユーザー登録のリクエストを送信するためのボタンを追加します（図B-7）。

通常、実務の開発でRESTかどうかによってボタンを複数用意することはありません。ここでは、これまで作成した（通常の画面遷移を伴う）ソースコードを残しておくために、あえてボタンを2つに分けています。あくまで学習用の構成であることをご理解ください。



図B-7 ユーザー登録画面



以下の手順で、POST メソッドを用いたユーザー登録処理を実装していきます。

- ① ファイルなどの作成
- ② ユーザーサービスの修正（部署IDの固定値の削除）
- ③ REST リクエストの作成（jQuery）
- ④ セキュリティ設定
- ⑤ ユーザー登録画面（HTML）の修正
- ⑥ jQuery による POST リクエストの実装

① ファイルなどの作成

図 B-8 のディレクトリ構成となるように、ファイルなどを作成してください。

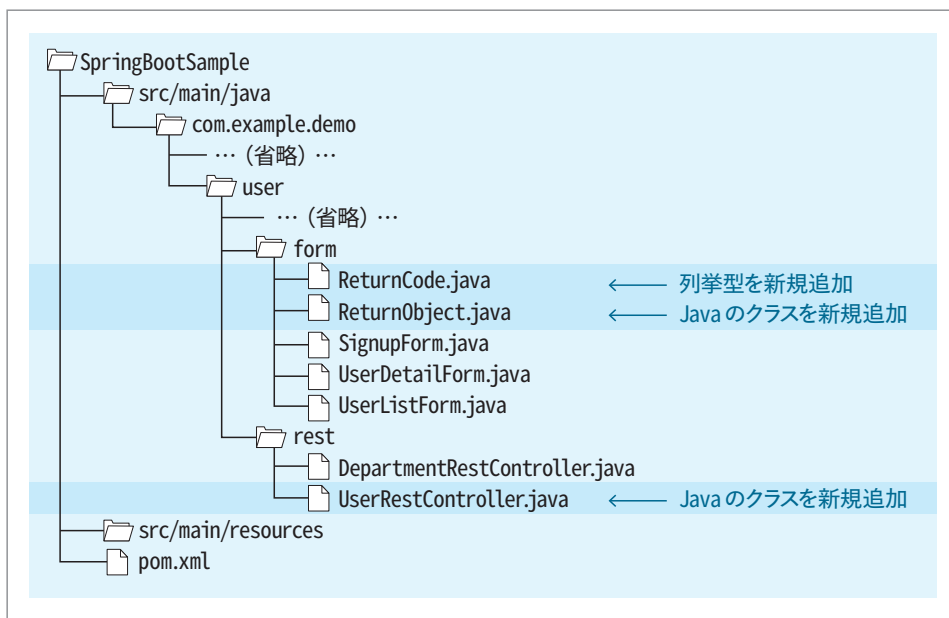


図 B-8 パッケージ・エクスプローラーの構成

② ユーザーサービスの修正（部署IDの固定値の削除）

ユーザーサービス（UserServiceImpl.java : MyBatis 版）では、部署IDに固定値を設定していました。部署IDはユーザー登録画面から送信される値を使用するため、該当部分をコメントアウトします（リスト B-8）。

リスト B-8 UserServiceImpl.java (MyBatis 版)

```
... (省略) ...

@Service
@RequiredArgsConstructor
@Slf4j
public class UserServiceImpl implements UserService {

    ... (省略) ...

    @Override
    public void signup(MUser user) {
        //user.setDepartmentId(1); // 部署
        user.setRole("ROLE_GENERAL"); // ロール

        // パスワードのハッシュ化
        String rawPassword = user.getPassword();
        user.setPassword(encoder.encode(rawPassword));

        int count = mapper.insertOne(user);
        log.info("登録件数={}件", count);
    }

    ... (省略) ...
}
```

同様に、`UserServiceImpl2.java` (JPA 版) でも、部署 ID に固定値を設定している箇所をコメントアウトします (リスト B-9)。

リスト B-9 UserServiceImpl2.java (JPA 版)

```
... (省略) ...

@Service
@RequiredArgsConstructor
@Slf4j
@Primary
public class UserServiceImpl2 implements UserService {

    ... (省略) ...

    @Override
    public void signup(MUser user) {
        // 既登録チェック
        boolean isExists = repository.existsById(user.getUserId());
        if (isExists) {
```

```

        throw new DuplicateKeyException("既に存在するユーザーです");
    }

    //user.setDepartmentId(1); // 部署
    user.setRole("ROLE_GENERAL"); // ロール

    // パスワードのハッシュ化
    String rawPassword = user.getPassword();
    user.setPassword(encoder.encode(rawPassword));

    repository.save(user);
}

... (省略) ...
}

```

次に、画面からサーバーに値を渡すフォームクラスに、選択された部署IDを受け取るフィールドを追加します (リストB-10)。

リストB-10 SignupForm.java

```

... (省略) ...

@BirthdayAge(birthdayFieldName = "birthday", ageFieldName = "age")
@Data
public class SignupForm {

    ... (省略) ...

    @NotNull
    private Integer gender;

    @NotNull A
    private Integer departmentId;
}

```

Aで、必須入力のバリデーション (@NotNull) を設定しています。これにより、部署が選択されていない場合はエラーになります。

③ RESTリクエストの作成 (jQuery)

ここからRESTの作成を行ないます。RESTを作成するときには大切なのは**インターフェース仕様**です。**インターフェース仕様** (以下、**I/F仕様**) とは、リクエストとレスポンスのルールを定義したものです。

◆ I/F仕様

ユーザー登録のI/F仕様は以下のように定義します。

● HTTPリクエストのI/F仕様（定義例）

- メソッド : POST
- URL : /user/rest/signup
- HTTPボディ : userId、passwordなどを設定

続いて、レスポンスのI/F仕様も以下のように定義します。

● HTTPレスポンスのI/F仕様（定義例）

- HTTPステータスコード : 成功時は200、サーバーエラー時は500
- 成功時は以下のようなJSONが返される

```
{
  "returnCode": 0,
  "message": "ユーザー登録成功"
}
```

returnCode=0は成功、returnCode=9は失敗と定義します。

なお、上記のI/F仕様はあくまでも簡易的なもので、実際の開発ではより詳細に定義します。

I/F仕様を定義しておくことで、サーバーにどのようなリクエストを送ればよいのか、またどのようなレスポンスが返ってくるのかを把握できます。これらはドキュメントとしてまとめておくのが一般的です。そうすることで、他の開発者に毎回RESTの仕様を説明する必要がなくなります。

また、I/F仕様に変更がなければ、内部の処理を変更しても影響はありません。一方で、I/F仕様に変更がある場合、その影響は大きくなります。たとえば、リクエストURLが変更された場合、そのRESTを利用しているすべての箇所に修正が必要になります。このような影響を避けるためにも、I/F仕様はあとから変更が少なくて済むように設計することが重要です。

たとえば、URIにバージョン番号を含める方法があります。/v1/user/1のようにバージョン番号（v1）を含めることで、既存のAPI（v1）を維持したまま、新しいAPI（v2）を追加することができます。

◆ レスポンスクラス

続いて、ユーザー登録を行なう**レスポンスクラス**を作成します（[リストB-11](#)）。**レスポンスクラス**は、処理結果を表わすリターンコード（成否の判定コード）とメッセージを保持するクラスです。

リストB-11 ReturnObject.java

```
package com.example.demo.user.form;

import lombok.Builder;
import lombok.Getter;
import lombok.ToString;

@Builder
@Getter
@ToString
public class ReturnObject {
    /** リターンコード */
    private final int returnCode;
    /** メッセージ */
    private final String message;
}
```

補足



▶ @Builder

@Builderは、Lombokが用意しているアノテーションです。このアノテーションを付けると、インスタンス生成のコードがわかりやすくなります。実際の使い方は後述します。

部署一覧取得ではデータそのものを返していましたが、登録・更新・削除などの処理ではリクエストが成功したかどうかを返すのが一般的です。

このような処理結果を返す理由は、失敗にも様々な原因があるためです。たとえば、以下のような場合が考えられます。

- 発生が予測可能なエラー（バリデーションNGなど）
- 予測不可能なエラー（ソースコードにバグがあった場合など）

リクエストに失敗した場合、どのような失敗だったのかを説明するためのメッセージを返すこともあります。なぜなら、ユーザー側で何らかの対応が必要になるためです。たとえば、バリデーションのエラーであれば、入力内容を修正すれば処理を実行できますし、バグなどが発生していた場合は、システム管理者に連絡する必要があります。

補足



▶ リアクション

リアクションとは、アクションに対する反応のことです。画面にはこのリアクションが必要です。たとえば、時間がかかる処理を実行するとします。このとき、グルグルと回っているマーク（スピナー）が表示されることがあります。他にも、処理が完了した、失敗したというメッセージが表示されます。

リアクションがあることによって、きちんと処理している（あるいは処理した）、ということがユーザーに伝わります。それによって、ユーザーの安心感にもつながります。

逆に、リアクションがなかった場合のことを考えてみてください。「時間のかかる処理を実行しているのに、画面が何も変わらないと、本当に動いているのか？」と怪しくなります。そうなると、ユーザーは同じボタンを何回も押してしまい、同じ処理のリクエストが何回も送られることになってしまいます。

開発現場では、こうした問題を防ぐために、適切なリアクションを実装することが重要です。

◆ 定数クラス

次は、リターンコードの定数クラス（enum）を作成します（[リストB-12](#)）。

リストB-12 ReturnCode.java

```
package com.example.demo.user.form;

public enum ReturnCode {
    SUCCESS(0),
    ERROR(9);

    /** 値 */
    private final int value;

    /** コンストラクタ */
    private ReturnCode(int value) {
        this.value = value;
    }

    /** getter */
    public int getValue() {
        return this.value;
    }
}
```

補足



enum

enum とは、定数を定義するためのクラスです。enum の定数は、以下のように使用します。

```
// 成功のリターンコードを取得する場合
System.out.println(ReturnCode.SUCCESS.getValue()); ← 0が表示される

// 失敗のリターンコードを取得する場合
System.out.println(ReturnCode.ERROR.getValue()); ← 9が表示される
```


ReturnCode クラスの中の定義 (SUCCESS や ERROR) から getter (getValue()) を呼び出すことで、対応する数値を取得できます。

なぜこのような定数クラスを作るかというと、ソースコードからマジックナンバーを排除するためです。マジックナンバーとは、意味がわかりにくい数値や文字列のことです。たとえば、以下のようなコードを考えてみてください。

```
if (value == 0) {  
    ... (何らかの処理) ...  
} else {  
    ... (何らかの処理) ...  
}
```

ここでは `value == 0` が何を表わすのか、コードを読んだだけでは判別できません。しかし、定数を使用することで、その意味が明確になります。

```
if (value == ReturnCode.SUCCESS.getValue()) { ← 成功だった場合ということが読み取れる  
    ... (何らかの処理) ...  
} else {  
    ... (何らかの処理) ...  
}
```

マジックナンバーがあると、コードの意味がわかりにくくなります。実際の開発では、コードは他の人が読んだり修正したりすることを前提に書く必要があります。

また、マジックナンバーが複数使われていると、その値が変更になった場合に修正箇所が増えてしまいます。そのため、定数クラスにまとめておくことで、変更にも柔軟に対応できます。

なお、マジックナンバーは、数値だけでなく文字列も対象となります。文字列であっても意味がわかりにくい場合は、ここで取り上げたマジックナンバーと同様の問題を引き起こします。

◆ RESTコントローラー

次は、ユーザー関連のRESTコントローラーを作成します。RESTコントローラーでは、これまで作ってきた通常のコントローラーと同様の処理を行ないます (リストB-13)。これまでの違いは、戻り値として `ReturnObject` (JSONとして返却されるオブジェクト) を返している点です。

リストB-13 UserRestController.java

```
package com.example.demo.user.rest;  
  
import org.modelmapper.ModelMapper;  
import org.springframework.web.bind.annotation.PostMapping;
```

```

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

import com.example.demo.user.domain.model.MUser;
import com.example.demo.user.domain.service.UserService;
import com.example.demo.user.form.ReturnCode;
import com.example.demo.user.form.ReturnObject;
import com.example.demo.user.form.SignupForm;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

@RestController
@RequiredArgsConstructor
@RequestMapping("/user/rest")
@Slf4j
public class UserRestController {

    private final UserService service;

    private final ModelMapper modelMapper;

    /** ユーザー登録 */
    @PostMapping("/signup")
    public ReturnObject signup(SignupForm form) {
        log.info(form.toString());
        try {
            // formをMUserクラスに変換
            MUser user = modelMapper.map(form, MUser.class);
            service.signup(user);
            // 戻り値の生成
            ReturnObject returnObject = ReturnObject.builder()
                .returnCode(ReturnCode.SUCCESS.getValue())
                .message("ユーザー登録成功")
                .build();
            return returnObject;

        } catch (Exception e) {
            log.error("ユーザー登録エラー(REST)", e);
            // 戻り値の生成
            ReturnObject returnObject = ReturnObject.builder()
                .returnCode(ReturnCode.ERROR.getValue())
                .message("ユーザー登録失敗(エラー)")
                .build();
            return returnObject;
        }
    }
}

```

```
}  
}  
}
```

補足



builder()

ReturnObjectに@Builderアノテーションを付けると、インスタンス生成が簡単になると説明しました。その実例は、サンプルコードの以下の部分です。

```
// 戻り値の生成
```

```
ReturnObject returnObject = ReturnObject.builder() ← builder()メソッドを呼び出す  
    .returnCode(ReturnCode.SUCCESS.getValue()) ← returnCodeの値をセット  
    .message("ユーザー登録成功") ← messageの値をセット  
    .build();
```

@Builderアノテーションを付けると、**builder()**メソッドをLombokが自動生成してくれます。このメソッドを起点に、各フィールドの値を**メソッドチェーン**（複数のメソッドを連続で呼び出す技法）で設定できます。

この方法により、コンストラクタやsetterを使用する場合と比べて、直感的で読みやすいコードになります。また、各フィールドをfinalにできるという利点もあります。

すべてのフィールドがfinalであり、getterのみを持つオブジェクトは、その値をあとから変更できません。このようなオブジェクトを**不変オブジェクト**（**イミュータブルオブジェクト**）と呼びます。

補足



不変オブジェクト（イミュータブルオブジェクト）

通常、クラスのインスタンスは他のクラスから参照されます。その際、フィールドの値が意図せず変更されることで、想定外の動作が発生することがあります。

不変オブジェクト（**イミュータブルオブジェクト**）であれば、生成後にフィールドの値が変更されることがないため、複数の箇所から参照されても安全に扱うことができます。

すべてのフィールドに対してgetterとsetterを用意するのが常に適切とは限りません。setterが不要なオブジェクトであれば、値を変更できない不変オブジェクトとして設計することで、より安全で意図が明確な設計になります。

④ セキュリティ設定

ログインしなくても、ユーザー登録画面で使用するRESTを使えるように、セキュリティを設定しておきます（[リストB-14](#)）。

リスト B-14 SecurityConfig.java

```
... (省略) ...

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    ... (省略) ...

    /** セキュリティの対象外を設定 */
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        // セキュリティ対象外の設定
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                .requestMatchers("/login").permitAll()
                .requestMatchers("/user/signup").permitAll()
                .requestMatchers("/error").permitAll()
                .requestMatchers("/department/**").permitAll()
                .requestMatchers("/user/rest/signup").permitAll()
                .requestMatchers("/h2-console/**").permitAll()
                .requestMatchers("/admin").hasAuthority("ROLE_ADMIN")
                .anyRequest().authenticated()
            ).formLogin(login -> login
                ... (省略) ...
            ).logout(logout -> logout
                ... (省略) ...
            ).rememberMe(remember -> remember
                ... (省略) ...
            );

        ... (省略) ...
    }

    ... (省略) ...
}
```

⑤ ユーザー登録画面 (HTML) の修正

RESTでユーザー登録のリクエストを送るボタンを追加します (リスト B-15)。

リストB-15 signup.html

```

... (省略) ...
<body class="bg-light">
  <form id="signup-form" method="post" action="/user/signup" class="form-signup mt-5" →
  th:object="${signupForm}">
    <input type="hidden" th:name="{_csrf.parameterName}" th:value="{_csrf.token}" />
    <h2 class="text-center" th:text="{user.signup.title}"></h2>

    ... (省略) ...

    <!-- エラーメッセージ -->
    <p th:if="{errorMessage != null}" class="text-danger mt-3" th:text="{errorMessage}"></p>
    <!-- 登録ボタン -->
    <input type="submit" th:value="{user.signup.btn}" class="btn btn-primary w-100 mt-4">
    <input type="button" id="signup-rest" class="btn btn-primary w-100 mt-4" →
    value="ユーザー登録(REST)">
    <div class="text-center mt-2">
      <a th:href="{@{/login}}" th:text="{link.login}"></a>
    </div>
  </form>
</body>
</html>

```

⑥ jQueryによるPOSTリクエストの実装

最後に、RESTでリクエストを送るボタン (id="signup-rest") を押した際の処理を作成します (リストB-16)。

リストB-16 signup.js

```

'use strict';

const RETURN_CODE_SUCCESS = 0;

/** 画面ロード時の処理 */
$(function() {
  // 部署一覧取得
  getDepartmentList();

  /** ユーザー登録(REST)ボタンを押したときの処理 */
  $('#signup-rest').click(function (event) { — A
    signup();
  });
});

```

```

/** 部署一覧取得 */
function getDepartmentList() {
    ... (省略) ...
}

/** ユーザー登録 */
function signup() {
    // CSRFトークンの値を取得
    var csrfToken = $('input[name="_csrf"]').val(); —②
    // formに入力された値を取得
    var formData = $('#signup-form').serializeArray(); —③

    // Ajax通信
    $.ajax({
        type : "POST",
        url : '/user/rest/signup',
        data : formData, —C
        timeout : 5000,
        headers : {
            "X-CSRF-TOKEN":csrfToken —B
        }
    }).done(function(data) {
        // Ajax成功時の処理
        console.log(data);
        // リターンコードによる判定
        if (data.returnCode === RETURN_CODE_SUCCESS) {
            alert('ユーザー登録に成功しました');
            // ログイン画面にリダイレクト
            window.location.href="/login"; —D
        } else {
            alert('ユーザー登録に失敗しました');
        }
    }).fail(function(jqXHR, textStatus, errorThrown) {
        // Ajax失敗時の処理
        console.log(jqXHR);
        console.log(textStatus);
        console.log(errorThrown);
        alert('ユーザー登録のリクエストでエラーが発生しました');
    }).always(function() {
        // 常に実行する処理 (特になし)
    });
}

```

jQueryでは、[A](#)のように画面ロード時にイベントハンドラを登録し、ボタンが押された

タイミングで処理を実行します。このサンプルコードでは、REST用のユーザー登録ボタンが押された場合に、`signup()` メソッドを呼び出します。

AjaxでPOSTリクエストを送る場合、セキュリティ上の理由から[B]のようにHTTPヘッダーの"`X-CSRF-TOKEN`"にCSRFトークンの値をセットする必要があります。

[C]では、`form`タグ内の入力内容 (`input`タグの値) を取得しています。jQueryの`serializeArray()` メソッドを使うことで、フォームの入力値を配列形式 (`name`と`value`の組) で取得できます。その値を`data`プロパティに設定することで、フォームの内容がHTTPリクエストとして送信されます。

ユーザー登録に成功した場合、[D]のようにJavaScriptでログイン画面 (`/login`) に遷移させています。このようにPOST後に別画面へ遷移することで、結果画面の再読み込みによる二重送信を防ぐことができます。これは **PRG パターン** (Post-Redirect-Get) と同様の考え方です。

実行

Spring Bootを実行して、ブラウザからユーザー登録画面にアクセスしてください。

<http://localhost:8080/user/signup>

画面が表示されたら、各項目を入力して [ユーザー登録 (REST)] ボタンを押します。これにより、RESTを利用したユーザー登録を実行できます (図B-9)。

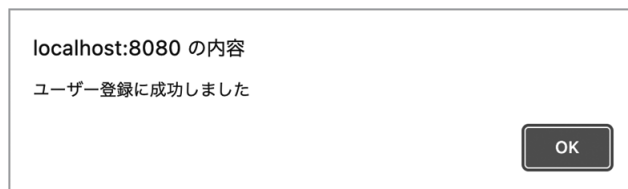


The form is titled "ユーザー登録" (User Registration). It contains the following fields and controls:

- ユーザーID:
- パスワード:
- ユーザー名:
- 誕生日:
- 年齢:
- 性別: ☒ 男性 ☐ 女性
- 部署:
- Buttons: "ユーザー登録", "ユーザー登録(REST)", and a link "ログインはこちら".

図B-9 ユーザー登録画面 (図B-7再掲)

ユーザー登録が正常に完了すると、ブラウザにメッセージボックスが表示されます (図B-10)。



図B-10 ユーザー登録成功のメッセージ

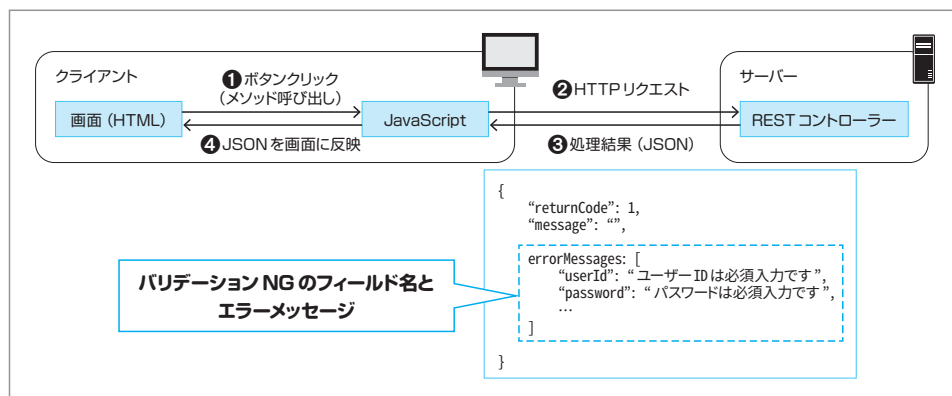
これで、RESTを利用したユーザー登録の基本的な処理は完成です。ただし、現状のプログラムでは入力内容のチェック（バリデーション）がまだ行なわれていません。

実際のアプリケーションでは、入力値の検証は非常に重要な処理です。次は、RESTにおけるバリデーションを実装し、より実用的なWeb APIへと上げていきましょう。

B-5

RESTの実装（3）—— バリデーション

これまでのバリデーションは、エラーメッセージが含まれた画面（HTML）がサーバーから返されていました。一方、RESTでリクエストを行なう場合は、画面（HTML）ではなくJSONが返されます（図B-11）。



図B-11 バリデーションエラーがあった場合

バリデーションでエラーがあった場合、どこの入力項目で、どのようなエラー内容だったのか（エラーメッセージ）がレスポンスのJSONに含まれるようにします。クライアント側では、その内容をもとにエラーメッセージを画面に反映する処理をJavaScriptで実装します。



サンプルアプリケーションの作成

サンプル ChB_REST/B-5

以下の手順で、RESTにおけるバリデーション機能を実装していきます。

- ① RESTの戻り値（レスポンスクラス）の修正
- ② RESTリクエストのバリデーションの実施
- ③ バリデーションエラーの反映（jQuery）
- ④ エラーメッセージの編集

① RESTの戻り値 (レスポンスクラス) の修正

まずは、バリデーションエラーが発生した場合に、その詳細内容をRESTの戻り値として設定できるように修正します。具体的には、`ReturnObject`にバリデーションによるエラーメッセージを保持するための`Map`などを追加します (リストB-17)。

リストB-17 `ReturnObject.java`

```
... (省略) ...

import java.util.Map;

@Builder
@Getter
@ToString
public class ReturnObject {
    /** リターンコード */
    private final int returnCode;

    /** メッセージ */
    private final String message;

    /** バリデーションエラーメッセージ */
    private final Map<String, String> errorMessages; — A

    /** バリデーションの共通エラーメッセージ */
    private final String validCommonErrorMessage; — B
}
```

Aのように、`Map`のキーにフィールド名、値にバリデーションエラーのメッセージを格納するようにします。これにより、クライアント側 (JavaScript) で「どの入力項目のエラーか」を判別できるようになります。

Bには、特定のフィールドではなく、入力全体に関するエラー (相関チェックなど) が発生した場合の共通メッセージを格納します。

◆ リターンコードの追加

続いて、バリデーションエラーを表わすリターンコードを`enum`に追加します (リストB-18)。

リストB-18 `ReturnCode.java`

```
... (省略) ...

public enum ReturnCode {
    SUCCESS(0),
```

```

VALID_ERROR(1),
ERROR(9);

... (省略) ...
}

```

◆ ② REST リクエストのバリデーションの実施

次に、REST のリクエストにバリデーションの処理を追加します。REST のリクエストであっても、第8章で学んだアノテーションを利用することで、通常の画面処理と同様にバリデーションを実行できます（[リスト B-19](#)）。

リスト B-19 UserRestController.java

```

... (省略) ...

import java.util.HashMap;
import java.util.Locale;
import java.util.Map;
import org.springframework.context.MessageSource;
import org.springframework.dao.DuplicateKeyException;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.validation.annotation.Validated;

@RestController
@RequiredArgsConstructor
@RequestMapping("/user/rest")
@Slf4j
public class UserRestController {

    private final UserService service;

    private final ModelMapper modelMapper;

    private final MessageSource messageSource;

    /** ユーザー登録 */
    @PostMapping("/signup")
    public ReturnObject signup(@Validated SignupForm form,
                               BindingResult bindingResult, Locale locale) { — A
        log.info(form.toString());
    }
}

```

```

// 入力チェック結果
if (bindingResult.hasErrors()) {
    Map<String, String> errorMessages = new HashMap<>();
    // エラーメッセージ取得
    for (FieldError error : bindingResult.getFieldErrors()) { ①
        String message = messageSource.getMessage(error, locale);
        errorMessages.put(error.getField(), message);
    }

    // 戻り値の生成
    ReturnObject returnObject = ReturnObject.builder()
        .returnCode(ReturnCode.VALID_ERROR.getValue())
        .message("バリデーションエラー")
        .errorMessages(errorMessages)
        .build();
    return returnObject;
}

try {
    // formをMUserクラスに変換
    MUser user = modelMapper.map(form, MUser.class);
    service.signup(user);
    // 戻り値の生成
    ReturnObject returnObject = ReturnObject.builder()
        .returnCode(ReturnCode.SUCCESS.getValue())
        .message("ユーザー登録成功")
        .build();
    return returnObject;
} catch (DuplicateKeyException e) {
    log.error("既に存在するユーザーです", e);
    // 戻り値の生成
    ReturnObject returnObject = ReturnObject.builder() — B
        .returnCode(ReturnCode.VALID_ERROR.getValue())
        .message("バリデーションエラー")
        .validCommonErrorMessage("既に存在するユーザーです")
        .build();
    return returnObject;
} catch (Exception e) {
    ... (省略) ...
}
}
}

```

ポイント① バリデーションエラー内容の取得

バリデーションエラーが発生していた場合、`BindingResult`の`getFieldErrors()`メソッドで、エラーが発生した項目の一覧 (List) を取得できます。このリストには、エラー内容を表わす `FieldError` オブジェクトが格納されています。

拡張 `for` 文で `FieldError` を 1 つずつ取り出し、`MessageSource` の `getMessage()` メソッドに渡すことで、第7章・第8章で作成したメッセージプロパティからエラーメッセージを取得できます。

また、引数に `Locale` を渡すことで、ブラウザの言語設定に応じたメッセージを取得することが可能です。

補足



国際化対応

`Locale` とは、言語や地域を表わす Java のクラスです。これを利用することで、ブラウザの言語設定に応じた言語でレスポンスを返すことができます。

このように、複数の言語に対応した Web アプリケーションを作成することを **国際化対応** と言います。

[A] のように、`@Validated` アノテーションと `BindingResult` を組み合わせることで、REST でも自動的にバリデーション結果を取得できます。バリデーションエラーがある場合は、エラー内容を JSON としてレスポンスに含めて返します。

また、[B] のように、すでに同じユーザーが登録されていた場合 (`DuplicateKeyException` 発生時) などは、特定の入力項目にひもづかないため、「全体に対する共通エラーメッセージ」としてレスポンスに含めます。

③ バリデーションエラーの反映 (jQuery)

バリデーションエラーが発生した場合、その内容を画面に表示できるように JavaScript を修正します (リスト B-20)。REST 用のユーザー登録ボタンを押した場合の処理の流れは、以下ようになります。

- ① バリデーションエラーのメッセージをクリア
- ② REST のリクエスト送信 (Ajax)
- ③ ユーザー登録に成功した場合：ログイン画面にリダイレクト
- ④ バリデーションエラーが発生した場合：エラーメッセージを画面に表示

最初にバリデーションエラーのメッセージをクリアしておかないと、古いメッセージが残ったまま新しいエラーが追加されてしまうため、事前にクリアしておく必要があります。

リストB-20 signup.js

```

'use strict';

const RETURN_CODE_SUCCESS = 0;
const RETURN_CODE_VALID_ERROR = 1;

/** 画面ロード時の処理. */
$(function() {
    // 部署一覧取得
    getDepartmentList();

    /** ユーザー登録(REST)ボタンを押したときの処理. */
    $('#signup-rest').click(function (event) {
        signup();
    });
});

/** 部署一覧取得 */
function getDepartmentList() {
    ... (省略) ...
}

/** ユーザー登録 */
function signup() {
    // バリデーション結果をクリア
    removeValidResult();

    // CSRFトークンの値を取得
    var csrfToken = $('input[name="_csrf"]').val();
    // formに入力された値を取得
    var formData = $('#signup-form').serializeArray();

    // Ajax通信
    $.ajax({
        type : "POST",
        url : '/user/rest/signup',
        data : formData,
        timeout : 5000,
        headers : {
            "X-CSRF-TOKEN":csrfToken
        }
    }).done(function(data) {
        // Ajax成功時の処理
        console.log(data);
        // リターンコードによる判定
        if (data.returnCode === RETURN_CODE_SUCCESS) {

```

```

alert('ユーザー登録に成功しました');
// ログイン画面にリダイレクト
window.location.href="/login";

```

```

} else if (data.returnValue === RETURN_CODE_VALID_ERROR) {
    // validationエラー時の処理
    $.each(data.errorMessages, function (key, value) {
        reflectValidResult(key, value)
    });

    if (data.validCommonErrorMessage !== null) {
        // 全体に共通するバリデーションエラー
        reflectCommonValidResult(data.validCommonErrorMessage);
    }
}

```

A

```

alert('入力内容に誤りがあります');

```

```

} else {
    alert('ユーザー登録に失敗しました');
}
}).fail(function(jqXHR, textStatus, errorThrown) {
    // Ajax失敗時の処理
    console.log(jqXHR);
    console.log(textStatus);
    console.log(errorThrown);
    alert('ユーザー登録のリクエストでエラーが発生しました');
}).always(function() {
    // 常に実行する処理（特になし）
});
}

```

```

/** バリデーション結果をクリア */
function removeValidResult() {
    $('.is-invalid').removeClass('is-invalid');
    $('.invalid-feedback').remove();
    $('.text-danger').remove();
    $('#validCommonErrorMessage').remove();
}

```

C

```

/** バリデーション結果の反映 */
function reflectValidResult(key, message) {

```

```

    // エラーメッセージ追加

```

```

    if (key === 'gender') { — B

```

```
// CSS適用
$('input[name=' + key + ']').addClass('is-invalid');
// エラーメッセージ追加
$('input[name=' + key + ']').parent().parent()
    .append('<div class="text-danger">' + message + '</div>');

} else if (key === 'departmentId') { —B
    // CSS適用
    $('select[id=' + key + ']').addClass('is-invalid');
    // エラーメッセージ追加
    $('select[id=' + key + ']').parent()
        .append('<div class="text-danger">' + message + '</div>');

} else { —B
    // CSS適用
    $('input[id=' + key + ']').addClass('is-invalid');
    // エラーメッセージ追加
    $('input[id=' + key + ']').after('<div class="invalid-feedback">' + message + '</div>');
}
}

/** バリデーション結果の反映（全体共通のエラーメッセージ） */
function reflectCommonValidResult(message) {
    // エラーメッセージ追加
    $('input[id=signup-button]')
        .before('<p id="valideCommonErrorMessage" class="text-danger mt-3">'
            + message + '</p>');
}
```

A の \$.each は、jQuery 特有の for 文です。バリデーションエラーが発生した場合、ReturnObject.java の errorMessages フィールド（Map 型）の件数分ループします。ループ内で reflectValidResult() メソッドを呼び出し、各項目のエラーメッセージを画面に反映します。また、全体に共通するエラーメッセージがある場合は、別途画面に表示します。

B のように、入力項目ごとに HTML の構造（DOM）が異なります。特に、性別（gender）や部署（departmentId）は、他の入力項目と構造が異なるため、項目ごとに処理を分岐しています。

バリデーションエラーが発生した場合は、input タグや select タグに is-invalid（Bootstrap の CSS クラス）を付与し、さらにエラーメッセージ用の要素（text-danger または invalid-feedback）を追加します。これにより、エラーメッセージを赤色で強調表示できます。

C の removeValidResult() メソッドでは、エラー表示を初期状態に戻す処理を行なっています。具体的には、以下の処理を行ないます。

- **input** タグの **is-invalid** という CSS クラスを削除
- **class** 属性に **invalid-feedback** が付いている要素を削除
- **class** 属性に **text-danger** が付いている要素を削除

D では、全体共通のバリデーションエラーメッセージを、登録ボタンの前に表示しています。

◆ 登録画面の修正

JavaScriptの修正に伴い、登録画面を一部修正します（リストB-21）。

リストB-21 signup.html

```
... (省略) ...
<body class="bg-light">
  <form id="signup-form" method="post" action="/user/signup" class="form-signup mt-5" =>
    th:object="{${signupForm}}"

    ... (省略) ...

  </div>
  <!-- エラーメッセージ -->
  <p th:if="{errorMessage != null}" class="text-danger mt-3" th:text="{errorMessage}"></p>
  <!-- 登録ボタン -->
  <input type="submit" id="signup-button" th:value="{user.signup.btn}" =>
class="btn btn-primary w-100 mt-4"> — A
  <input type="button" id="signup-rest" class="btn btn-primary w-100 mt-4" =>
value="ユーザー登録(REST)">
  <div class="text-center mt-2">
    <a th:href="{@{/login}}" th:text="{link.login}"></a>
  </div>
</form>
</body>
</html>
```

A では、jQueryで登録ボタンを特定し、その直前に共通エラーメッセージを挿入できるように、id属性（signup-button）を付与しています。

4 メッセージ編集

最後にメッセージを編集します。バリデーションメッセージではフィールド名がキーとして使用されるため、部署の入力項目でフィールド名が英語のまま表示されないよう、`messages.properties`に画面表示用の日本語名を定義します (リストB-22)。

リストB-22 `messages.properties`

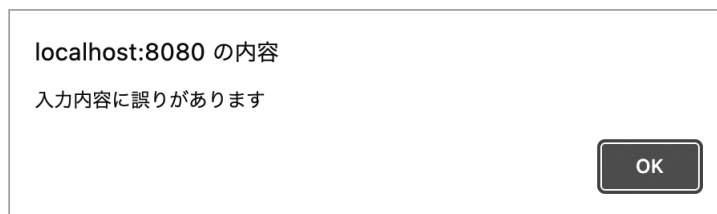
```
user.signup.title=ユーザー登録
user.signup.btn=ユーザー登録
userId=ユーザーID
password=パスワード
userName=ユーザー名
birthday=誕生日
age=年齢
male=男性
female=女性
gender=性別
departmentId=部署
link.login=ログインはこちら
```

実行

Spring Bootを実行して、ブラウザからユーザー登録画面にアクセスしてください。

<http://localhost:8080/user/signup>

何も入力せずに [ユーザー登録 (REST)] ボタンを押すと、サーバー側でバリデーションが実行され、JSON形式のレスポンスが返ります。画面には「入力内容に誤りがあります」というアラートが表示されます (図B-12)。



図B-12 バリデーションエラー時のメッセージ

このアラートを閉じると、jQueryによってHTMLが動的に書き換えられ、各入力項目の下に赤文字でエラーメッセージが表示されます(図B-13)。

これで、データの登録だけでなく、エラーハンドリングも含めた実用的なREST処理がすべて完成しました。

図B-13 ユーザー登録画面（バリデーション発生時）

B-6

RESTの実装（4）—— PUT/DELETEメソッド

最後は、更新と削除の処理を作ります。図B-14のように、ユーザー詳細画面に更新と削除のREST用のボタンを追加します。

図B-14 ユーザー詳細画面

これらのボタンを押すと、更新（PUT）と削除（DELETE）をRESTで行なえるようにします。

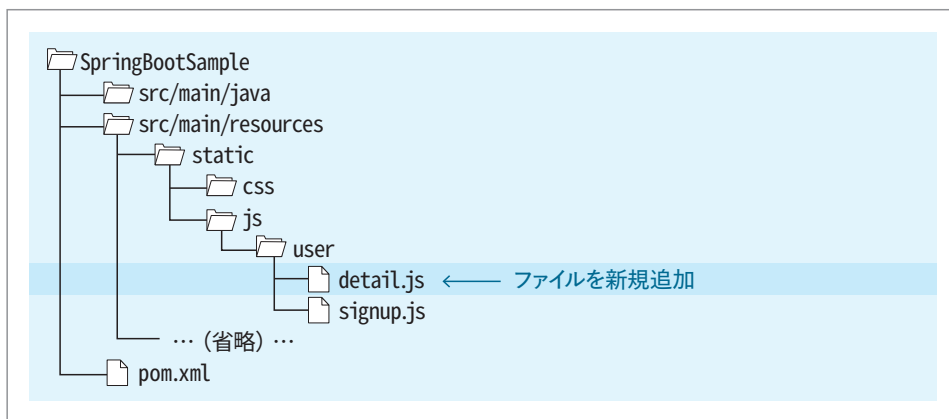


以下の手順で、PUTメソッドとDELETEメソッドを用いた機能を実装していきます。

- ① ファイルなどの作成
- ② RESTコントローラーの修正 (PUT/DELETEの追加)
- ③ ユーザー詳細画面 (HTML) の修正
- ④ jQueryによるリクエスト送信の実装

① ファイルなどの作成

図B-15の構成となるように、ファイルなどを作成してください。



図B-15 パッケージ・エクスプローラーの構成

② RESTコントローラーの修正 (PUT/DELETEの追加)

次に、ユーザー関連のRESTコントローラー (UserRestController.java) に更新と削除の処理を追加します (リストB-23)。

リストB-23 UserRestController.java

```
... (省略) ...  
  
import org.springframework.web.bind.annotation.DeleteMapping;  
import org.springframework.web.bind.annotation.PutMapping;  
import com.example.demo.user.form.UserDetailForm;  
  
@RestController  
@RequiredArgsConstructor
```

```

@RequestMapping("/user/rest")
@Slf4j
public class UserRestController {

    ... (省略) ...

    /** ユーザー登録 */
    @PostMapping("/signup")
    public ReturnObject signup(@Validated SignupForm form,
                               BindingResult bindingResult, Locale locale) {
        ... (省略) ...
    }

    /** ユーザー更新 */
    @PutMapping("/update") — ①
    public ReturnObject updateUser(UserDetailForm form) {
        try {
            // ユーザーを更新
            service.updateUserOne(form.getUserId(),
                                  form.getPassword(),
                                  form.getUserName());

            // 戻り値の生成
            ReturnObject returnObject = ReturnObject.builder()
                .returnCode(ReturnCode.SUCCESS.getValue())
                .message("ユーザー更新成功")
                .build();
            return returnObject;

        } catch (Exception e) {
            log.error("ユーザー更新でエラー", e);
            // 戻り値の生成
            ReturnObject returnObject = ReturnObject.builder()
                .returnCode(ReturnCode.ERROR.getValue())
                .message("ユーザー更新失敗(エラー)")
                .build();
            return returnObject;
        }
    }

    /** ユーザー削除 */
    @DeleteMapping("/delete") — ①
    public ReturnObject deleteUser(UserDetailForm form) {
        try {
            // ユーザーを削除
            service.deleteUserOne(form.getUserId());

```

```

// 戻り値の生成
ReturnObject returnObject = ReturnObject.builder()
    .returnCode(ReturnCode.SUCCESS.getValue())
    .message("ユーザー削除成功")
    .build();
return returnObject;

} catch (Exception e) {
    log.error("ユーザー削除でエラー", e);
    // 戻り値の生成
    ReturnObject returnObject = ReturnObject.builder()
        .returnCode(ReturnCode.ERROR.getValue())
        .message("ユーザー削除失敗(エラー)")
        .build();
    return returnObject;
}
}
}

```

ポイント ① @PutMapping、@DeleteMapping

@PutMapping は、HTTPのPUTメソッド（更新用）のリクエストを受け取るためのアノテーションです。同様に、**@DeleteMapping** はHTTPのDELETEメソッド（削除用）に対応したアノテーションです。どちらのアノテーションも、使い方は**@GetMapping**や**@PutMapping**と同様です。

サンプルコードでは、以下のように処理とURLを対応させています。

- 更新処理 ➡ /user/rest/update
- 削除処理 ➡ /user/rest/delete

また、内部で行なっている更新や削除の処理自体は、第10章で作成した通常のコントローラー（`UserDetailController.java`）とほとんど同じ内容です。

③ ユーザー詳細画面（HTML）の修正

次に、ユーザー詳細画面の修正を行います。まず、jQueryを読み込みます（[リスト B-24](#)）。ユーザー詳細画面はレイアウト機能を使用しているため、どの画面でも共通して使用するJavaScript（jQueryなど）は、レイアウト用のテンプレートで一括して読み込むようにします。

リスト B-24 layout.html

```
... (省略) ...
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <!-- 共通CSS読み込み -->
  <link rel="stylesheet" th:href="@{/webjars/bootstrap/css/bootstrap.min.css}">
  <link rel="stylesheet" th:href="@{/css/layout.css}">
  <!-- 共通JS読み込み -->
  <script th:src="@{/webjars/bootstrap/js/bootstrap.min.js}" defer></script>
  <script th:src="@{/webjars/jquery/jquery.min.js}" defer></script>
  <title>Layout</title>
</head>
... (省略) ...
```

続いて、ユーザー詳細画面（HTML）を修正します（リスト B-25）。

リスト B-25 detail.html

```
... (省略) ...
<head>
  <title>ユーザー詳細</title>
  <!-- JS読み込み -->
  <script th:src="@{/js/user/detail.js}" defer></script> — A
</head>
<body>
  <div layout:fragment="content">
    <div class="header border-bottom">
      <h1 class="h2">ユーザー詳細</h1>
    </div>
    <form id="user-detail-form" method="post" th:action="@{/user/detail}"
      class="form-signup" th:object="${userDetailForm}">
      <input type="hidden" th:field="*{userId}">
      <!-- ユーザー詳細情報 -->
      <table class="table table-striped table-bordered table-hover">
        ... (省略) ...
      </table>
      <!-- ボタンエリア -->
      <div class="text-center">
        <!-- 削除ボタン -->
        <button class="btn btn-danger" type="submit" name="delete">
          削除
        </button>
        <!-- 更新ボタン -->
        <button class="btn btn-primary" type="submit" name="update">
          更新
      </div>
    </form>
  </div>
</body>
```

```

        </button>
      </div>
      <!-- ボタンエリア(REST) -->
      <div class="text-center mt-2">
        <!-- 削除ボタン -->
        <button class="btn btn-danger" type="button" id="delete-rest">
          削除(REST)
        </button>
        <!-- 更新ボタン -->
        <button class="btn btn-primary" type="button" id="update-rest">
          更新(REST)
        </button>
      </div>
      <!-- 給料情報 -->
      <th:block th:if="{salaryList != null and salaryList.size() > 0}">
        ... (省略) ...
      </th:block>
    </form>
  </div>
</body>
</html>

```

B

A では、ユーザー詳細画面専用のJavaScriptを読み込みます。

B では、REST用のボタンを追加しています。

4 jQueryによるリクエスト送信の実装

最後に、ユーザー詳細画面から REST でリクエストを送るためのJavaScriptを作成します。基本的にはユーザー登録時に作成した処理とほとんど同じですが、HTTPメソッドにPUTやDELETEを指定する点異なります（[リストB-26](#)）。

リストB-26 detail.js

```

'use strict';

const REQUEST_SUCCESS = 0;

/** 画面ロード時の処理. */
$(function() {

  /** 更新(REST)ボタンを押したときの処理 */
  $('#update-rest').click(function (event) {
    updateUser();
  });
});

```

```

    /** 削除(REST)ボタンを押したときの処理 */
    $('#delete-rest').click(function (event) {
        deleteUser();
    });
});

/** ユーザー更新 */
function updateUser() {
    // formに入力された値を取得
    var formData = $('#user-detail-form').serializeArray();

    // Ajax通信
    $.ajax({
        type : "PUT",
        url : '/user/rest/update',
        data : formData,
        timeout : 5000,
    }).done(function(data) {
        // Ajax成功時の処理
        console.log(data);
        // リターンコードによる判定
        if (data.returnValue === REQUEST_SUCCESS) {
            alert('ユーザー更新に成功しました');
            // ユーザー一覧画面にリダイレクト
            window.location.href="/user/list";
        } else {
            alert('ユーザー更新に失敗しました');
        }
    }).fail(function(jqXHR, textStatus, errorThrown) {
        // Ajax失敗時の処理
        alert('ユーザー更新のリクエストでエラーが発生しました');
    }).always(function() {
        // 常に実行する処理（特になし）
    });
}

/** ユーザー削除 */
function deleteUser() {
    // formに入力された値を取得
    var formData = $('#user-detail-form').serializeArray();

    // Ajax通信
    $.ajax({
        type : "DELETE",

```



```

url : '/user/rest/delete',
data : formData,
timeout : 5000,
}).done(function(data) {
    // Ajax成功時の処理
    console.log(data);
    // リターンコードによる判定
    if (data.returnCode === REQUEST_SUCCESS) {
        alert('ユーザー削除に成功しました');
        // ユーザー一覧画面にリダイレクト
        window.location.href="/user/list";
    } else {
        alert('ユーザー削除に失敗しました');
    }
}).fail(function(jqXHR, textStatus, errorThrown) {
    // Ajax失敗時の処理
    alert('ユーザー削除のリクエストでエラーが発生しました');
}).always(function() {
    // 常に実行する処理 (特になし)
});
}

```

実行

Spring Bootを実行してログインし、ブラウザからユーザー詳細画面にアクセスしてください。画面が表示されたら、REST用の削除・更新ボタンを確認します (図 B-16)。

SpringSample

ログアウト

ユーザー一覧

ファイル一覧

ユーザー詳細

ユーザーID	user1@co.jp
パスワード	
ユーザー名	ユーザー1
誕生日	2000/01/01
年齢	21
性別	女性
部署名	営業部

削除

更新

削除 (REST)

更新 (REST)

図 B-16 ユーザー詳細画面 (図 B-14 再掲)

内容を編集して[更新 (REST)] ボタンを押し、成功するとメッセージが表示され (図 B-17)、ユーザー一覧画面へ遷移します。

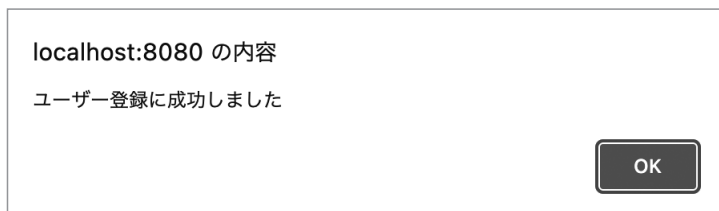


図 B-17 ユーザー更新成功のメッセージ

同様に、[削除 (REST)] ボタンを押して成功すると、削除完了のメッセージが表示されます (図 B-18)。

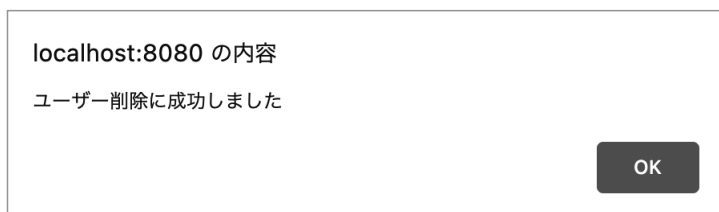


図 B-18 ユーザー削除成功のメッセージ

これで、主要なHTTPメソッドGET、POST、PUT、DELETEを用いたREST APIの作成はすべて終了です。



補足

RESTのセキュリティ

REST (Web API) でもセキュリティ対策は不可欠です。RESTは主にシステム間の連携を目的とするため、ブラウザによるログイン画面ではなく、以下のような方法で安全性を高めるのが一般的です。

- 通信の暗号化
- 認証情報の秘匿 (URL にユーザーIDやパスワードなどを含めない)
- ネットワーク制限 (特定のIPアドレスやセグメントからのみリクエストを許可する) など

中でも広く使われているのが **APIキー** による認証です。これは、API利用者に事前に発行した専用のキーを、リクエストのHTTPヘッダーに含めて送信してもらい、サーバー側でその正当性を検証する仕組みです (図 B-19)。

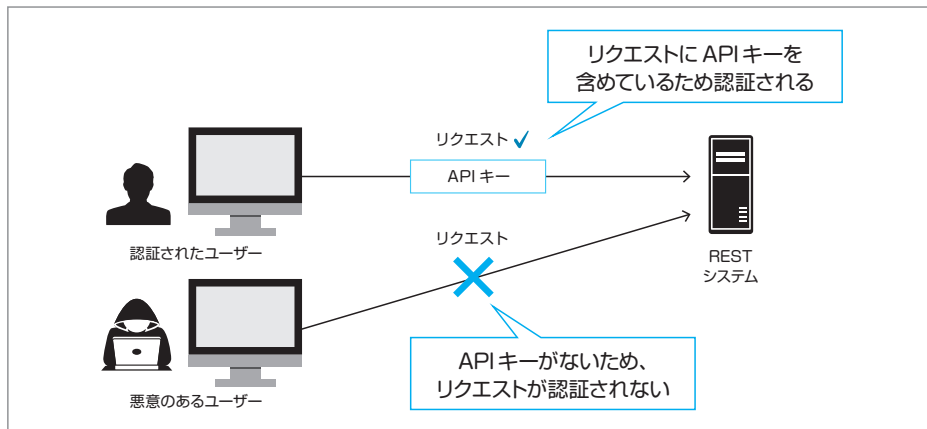


図 B-19 APIキーとは？

サンプルアプリケーション

サンプル ChB_REST/B-7

ここからは、サンプルアプリケーションを用いて、Spring Securityを使ってAPIキー認証を実装する際の流れを説明します（図B-20）。実際にすべてを作成する必要はありませんが、興味がある方は作成にチャレンジしてみてください。

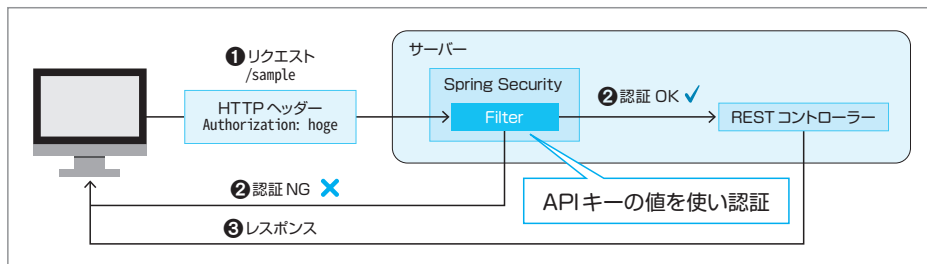


図 B-20 APIキーを使用したRESTの概要

- ① リクエスト ➡ /sample への送信時、ヘッダーのAuthorization項目にAPIキーを含める。
- ② 認証 ➡ フィルター（Filter）でAPIキーを検証する。認証が成功すればRESTコントローラーへ処理が渡され、失敗した場合はHTTPステータス403が返される。
- ③ レスポンス ➡ 認証成功時のみ、コントローラーの処理結果を返す。

コントローラー

まずは、認証後に呼び出されるシンプルなRESTコントローラーを用意します（[リストB-27](#)）。

リストB-27 SampleRestController.java

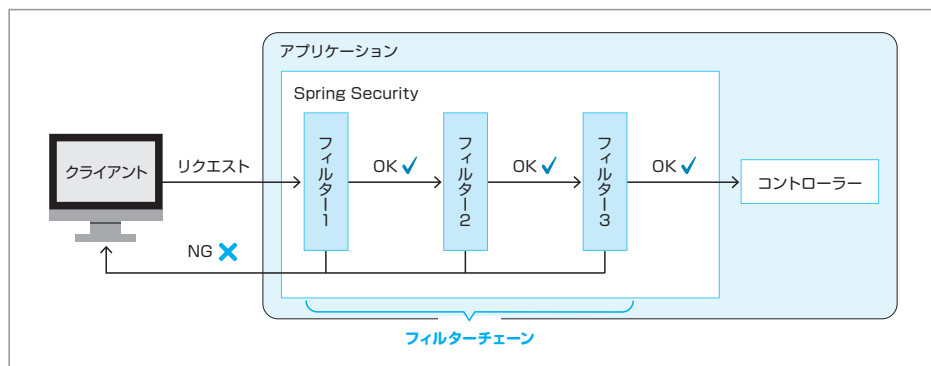
```
@RestController
public class SampleRestController {

    @GetMapping("/sample")
    public String getSample() {
        return "sample success";
    }
}
```

このコントローラーでは、/sampleにリクエストが来たら、sample successという文字列を返します。

フィルターの概要

Spring Securityでは、リクエストがコントローラーに届く前に、複数の**フィルター**が連なる**フィルターチェーン**を通過します（[図B-21](#)）。つまり、コントローラーに到達する前の段階で認証・認可処理が実行される仕組みです。



図B-21 フィルターチェーンとは？

API キー認証を実現するには、図 B-22 のようなフィルター構成をします。

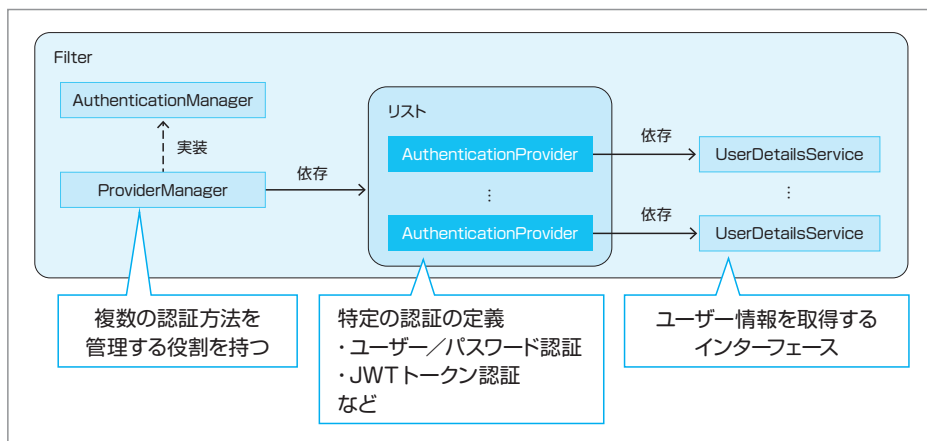


図 B-22 SpringSecurityのフィルター概要

- **AuthenticationManager** ➡ 複数の認証を管理するインターフェース。実際には、その実装クラスである ProviderManager を使う。ProviderManager では、Authentication Provider をリストで持っている。ProviderManager が複数の AuthenticationProvider の認証処理を呼び出し、いずれか 1 つでも認証に成功すれば、そのリクエストは認証されたものとして扱われる。
- **AuthenticationProvider** ➡ 何の認証を行なうかを定義するためのインターフェース。認証方法に応じた実装クラスが用意されている。ユーザー/パスワードによる認証、JWT トークンや OAuth2 の認証に対応した実装クラスがある。実装クラスによっては、UserDetailsService を呼び出すことがある。
- **UserDetailsService** ➡ 認証を行なう際に、ユーザー情報を取得するインターフェース。第 15 章では、DB からユーザー情報を取得した。

つまり、1 つのフィルターで複数の認証方法を扱うことができます。Authentication Provider は Spring Security が用意しているため、適切なクラスを選択するだけで利用できます。また、必要に応じて UserDetailsService を実装します。

フィルターの実装

まずはリクエストの内容を取得して、認証情報（Authentication）を作成するフィルターを実装します（リストB-28）。

リストB-28 MyPreAuthenticatedProcessingFilter.java

```
public class MyPreAuthenticatedProcessingFilter extends AbstractPreAuthenticatedProcessingFilter {

    private final String API_KEY_HEADER_NAME = "Authorization";

    @Override
    protected Object getPreAuthenticatedPrincipal(HttpServletRequest request) {
        return "";
    }

    @Override
    protected Object getPreAuthenticatedCredentials(HttpServletRequest request) {
        // APIキーの取得
        return request.getHeader(API_KEY_HEADER_NAME);
    }
}
```

様々なフィルターのクラスがSpringにより用意されています。本書ではAbstractPreAuthenticatedProcessingFilterを継承したクラスを作成します。このクラスは、外部のシステムで既に認証されたリクエストを処理するためのフィルターです。APIキーを発行しているということは、事前に認証されています。

そのため、このクラスでは認証は行わず、リクエストから認証に関する情報を抽出します。以下の2つの抽象メソッドを実装します。

- getPreAuthenticatedPrincipal()
- getPreAuthenticatedCredentials()

上記メソッドはそれぞれ、HTTP リクエストからPrincipalとCredentialsを作成するためのメソッドです。認証方法などによってそれぞれの意味が変わりますが、CredentialsがユーザーID、Principalがパスワードというイメージです。

上記のサンプルコードでは、getPreAuthenticatedCredentials()メソッド内で、HTTPヘッダーのAuthorizationという項目の値を取得しています。

ユーザー認証の実装

次に、HTTPヘッダーから取得したAPIキーを使って、ユーザー認証を行なうサービスを実装します (リストB-29)。

リストB-29 MyAuthenticationUserDetailsService.java

```
public class MyAuthenticationUserDetailsService
    implements AuthenticationUserDetailsService<PreAuthenticatedAuthenticationToken> {

    @Override
    public UserDetails loadUserDetails(PreAuthenticatedAuthenticationToken token) throws ➡
        UsernameNotFoundException {
        // APIキーの取得
        String apiKey = token.getCredentials().toString();

        if (apiKey == null || "".equals(apiKey)) {
            throw new UsernameNotFoundException("User not found");
        }

        // ユーザーの作成
        return new User("user",
            "password",
            AuthorityUtils.createAuthorityList("ROLE_USER"));
    }
}
```

APIキーを取得して、ユーザー認証を行なうサービスを作るために、AuthenticationUserDetailsServiceを実装したクラスを作成します。このクラスは、認証情報 (Authentication) からユーザー (UserDetails) を作成するためのインターフェースです。認証情報 (Authentication) は先ほど作成したフィルターで作りました。そして、PreAuthenticatedAuthenticationTokenというクラスで認証情報 (Authentication) を保持しています。

あとは、実装したメソッドの中でユーザー (UserDetails) を作成します。本来なら、取得したAPIキーを使ってDBに問い合わせなどを行ない、APIキーの有効性を確認します。上記はサンプルのため、そのような処理は割愛しています。

セキュリティ設定

最後にセキュリティ設定を実装します（[リストB-30](#)）。

リストB-30 SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    /** ユーザー認証サービス */
    @Bean
    AuthenticationUserService<PreAuthenticatedAuthenticationToken> userDetailsService() {
        return new MyAuthenticationUserService();
    }

    /** プロバイダー */
    @Bean
    AuthenticationProvider authenticationProvider() {
        PreAuthenticatedAuthenticationProvider provider =
            new PreAuthenticatedAuthenticationProvider();
        provider.setPreAuthenticatedUserDetailsService(userDetailsService());
        return provider;
    }

    /** AuthenticationManager */
    @Bean
    AuthenticationManager authenticationManager(
        AuthenticationProvider authenticationProvider) throws Exception {
        return new ProviderManager(authenticationProvider);
    }

    /** フィルター */
    @Bean
    AbstractPreAuthenticatedProcessingFilter filter(AuthenticationManager authenticationManager) {
        MyPreAuthenticatedProcessingFilter filter =
            new MyPreAuthenticatedProcessingFilter();
        filter.setAuthenticationManager(authenticationManager);
        return filter;
    }

    /** セキュリティ設定 */
    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {
```



```
// セッション無効化
http.sessionManagement(session -> session
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

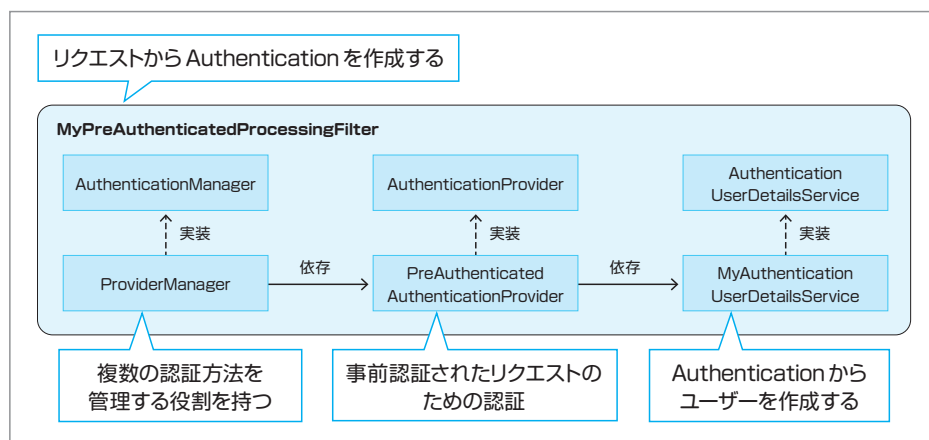
// すべてのリクエストに対して、APIキー認証フィルターを適用
http
    .csrf(csrf -> csrf.disable())
    .authorizeHttpRequests(authorize -> authorize
        .anyRequest().authenticated())
    .addFilter(filter(authenticationManager));

return http.build();
}
}
```

まずは以下のBeanを実装します。

- **UserDetailsService** (ユーザー認証サービス)
- **AuthenticationProvider**
- **AuthenticationManager** (**ProviderManager**)
- フィルター

これらのクラスは図B-23のような関係になっています。



図B-23 作成したSpringSecurityのフィルター

そして、セキュリティ設定を行ないます。RESTのサービスを提供するシステムのため、まずはセッションを無効化します (リストB-31)。

リストB-31 SecurityConfig.java (セッション無効化の箇所を抜粋)

```
http.sessionManagement(session -> session
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

CSRFも無効化します (リストB-32)。そして、すべてのリクエストに対して認証が必要となるように設定したうえで、作成したフィルターを使用するように設定しています。

リストB-32 SecurityConfig.java (セキュリティ設定の箇所を抜粋)

```
// すべてのリクエストに対して、APIキー認証フィルターを適用
http
    .csrf(csrf -> csrf.disable()) ← CSRF無効化
    .authorizeHttpRequests(authorize -> authorize
        .anyRequest().authenticated()) ← すべてのリクエストに認証が必要
    .addFilter(filter(authenticationManager)); ← 作成したフィルターを使用
```

これでサンプルの実装は完了です。動作確認のために、curl コマンド (WindowsはリストB-33、MacはリストB-34) を実行します。

リストB-33 curlコマンド Windows

```
curl.exe -H "Authorization:hoge" -i http://localhost:8080/sample
```

リストB-34 curlコマンド Mac

```
curl -H 'Authorization:hoge' -i http://localhost:8080/sample
```

これは、HTTPヘッダーにAuthorization=hogeを設定してからHTTPリクエストを送っています。以下のような実行結果が表示されれば、認証に成功しています。

実行結果

```
HTTP/1.1 200 ← HTTPステータスが200になっているため、リクエスト成功
Set-Cookie: JSESSIONID=89204CC028E21D527C1981DC2FCD7D9F; Path=/; HttpOnly
X-Content-Type-Options: nosniff
... (省略) ...
sample success
```

次に、HTTPヘッダーがないリクエストを送ります。

curlコマンド

```
curl -i http://localhost:8080/sample
```

実行結果

```
HTTP/1.1 403 ← HTTPステータスが403 (権限なし)
X-Content-Type-Options: nosniff
... (省略) ...
リクエストが失敗します。
```

B-7 まとめ

この付録で学んだ内容は、以下の通りです。

→ 概要

- ✓ 広義の意味の REST は、シンプルで柔軟な分散システムを構築するためのアーキテクチャスタイルのこと。REST の原則に沿って作られたシステムを RESTful という。

- メリット

- ✓ システム間連携が行なえるため、シンプルで柔軟な分散システムを構築しやすい
- ✓ 分散システムでは機能の修正・追加がしやすいため、ビジネスの競争力が上がる

- デメリット

- ✓ システム関連機能の開発・テストする必要がある

- ✓ 狭義の意味の REST は Web API のこと。単に「REST」と言うと、狭義の意味の REST を指すことが多い。

- メリット

- ✓ 1 つの機能を複数のシステムで作る必要がなくなるため、機能の開発・保守が楽になる
- ✓ 非同期通信を行なうことで、ユーザーの操作性（ユーザビリティ）が上がる

- デメリット

- ✓ リクエストの送信、レスポンスを画面に反映する JavaScript の開発が必要になる

→ REST の実装

- ✓ `@RestController` アノテーションをクラスに付けると、`@GetMapping` などのアノテーションが付いている全メソッドの戻り値をそのままレスポンスとして返せるようになる。つまり、`@ResponseBody` アノテーションをすべてのメソッドに付ける手間が省ける。なお、デフォルトでは JSON 形式のレスポンスになる。
- ✓ バリデーションエラーが発生している場合、`BindingResult` の `getFieldErrors()` メソッドで、エラーが発生した項目の `List` を取得できる。バリデーションエラーのフィールドを表わす `FieldError` 型のオブジェクトが、この `List` に格納されている。
- ✓ `@PutMapping` は、HTTP の PUT メソッドのリクエストを処理するためのアノテーション。`@DeleteMapping` は、HTTP の DELETE メソッドに対応したアノテーション。どちらのアノテーションも、使い方は `@GetMapping` や `@PostMapping` と同様。

付録C

ビルド

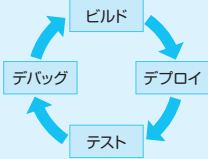
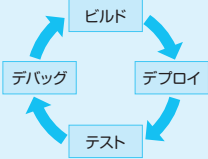
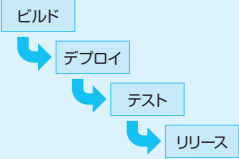
ここまではEclipse上でアプリケーションを動かしてきました。しかし、開発が完了に近づくくと、ローカルPC以外の環境で動作させる必要が出てきます。第17章でも説明したように、テスト環境、ステージング環境、本番環境といった、それぞれの環境でアプリケーションを稼働させなければなりません。

この付録では、作成したアプリケーションを実際のサーバー上で動作させるための具体的な方法について学習します。

C-1

リリースまでの作業

まずは、開発工程においてどのような作業が行なわれるかを整理します（図C-1）。

工程	開発～テスト	受け入れテスト	リリース
作業			
環境	テスト環境	ステージング環境	本番環境
サイクル数	多	少	—

図C-1 工程ごとの作業

◆ 開発～テスト

この工程では、ビルド、デプロイ、テスト、デバッグ（修正）を何度も繰り返します。なお、「テスト」の工程には単体テストや結合テストなど複数の種類があり、プロジェクトによって名称や定義が異なる場合があります。

◆ 受け入れテスト

開発者によるテスト完了後、検収（発注通りに作られているかの確認）のためにユーザー企業（発注者）が行なうテストです。ユーザーの想定と開発者の認識にズレがないかを確認します。ここでズレが発覚すると、大幅な作り直しが発生し、リリース直前の過密なスケジュールの中で修正と再テストが求められることもあります。

◆ リリース

受け入れテストをパスしたあと、本番環境でアプリケーションを正式に稼働させることです。

では、上記の説明で出てきた各用語について説明します。

ビルドとは？

ビルドとは、ソースコードから実際のサーバーで実行可能なファイルを作成することです。Javaであれば、**jar** ファイルや**war** ファイルを作成することがビルドにあたります。

ソースコードは人が理解できるように記述されており、それをコンピューターが理解できる形式に変換することを**コンパイル**といいます。Javaのソースコードをコンパイルすると、**class** ファイルが生成されます。通常、1つの**java** ファイルから1つ以上の**class** ファイルが生成されます。

jar ファイルは、作成した**class** ファイル群を**zip**形式でまとめたファイルです。**zip**形式であるため、7-Zipなどのツールで解凍して中身を確認できます。

war ファイルとは、Java EE仕様に準拠したWebアプリケーションをまとめたファイルです。こちらも**zip**形式であるため、ツールで解凍して中身を確認できます。

Spring Bootでは、**jar** ファイルを用いることでデプロイ作業が簡略化されるため便利です。一方で、既存のシステムでは**war** ファイルが利用されている場合もあります。

デプロイとは？

デプロイ（deploy）とは、直訳すると「配備する」という意味です。開発現場では、「アプリケーションをサーバー上で実行できる状態にすること」を指します。

jar ファイルをサーバーに配置しただけでは、アプリケーションは動作しません。実行可能

な状態にするためには、様々な作業が必要です。たとえば、環境変数の設定や、プログラムで使用するディレクトリの作成などがあります。

リリースとは？

リリースとは、エンドユーザーがアプリケーションを利用できるようにすることです。本番環境でアプリケーションを稼働させることを指します。既存のシステムを停止し、ユーザーからのアクセスを一時的に遮断する場合があります。

どの環境でも起こりうることですが、修正後のアプリケーションをデプロイすると、それまで正常に動作していたプログラムが動かなくなることがあります。そのため、修正前の状態に戻せるようにしておくことが重要です。たとえば、修正前の **jar** ファイルをバックアップしてから、新しい **jar** ファイルを配置する方法があります。

補足



CI/CD

ビルド、デプロイ、テストは、何度も繰り返し実施するのが一般的です。これらを毎回手作業で行なうのは大きな負担となります。特にテストは同じ作業を繰り返すことが多く、人的ミスも発生しやすくなります。

そこで、近年ではビルド、デプロイ、テストを自動化する取り組みが広く行なわれています。これを **Continuous Integration**（継続的インテグレーション）／**Continuous Delivery**（継続的デリバリー）と言います。これらをまとめて **CI/CD**（シーアイ／シーディー）と呼びます。

CI（Continuous Integration）は主にビルドやテストの自動化を指し、CD（Continuous Delivery）はリリースまでの工程を自動化する考え方です。

CI/CDを導入するには、ツールの設定や運用設計など一定の知識と準備が必要です。しかし、自動化によって作業の効率化や品質の向上が期待でき、その効果は非常に大きいものです。

C-2

Executable Jar

ここからは、これまで作ってきたWebアプリケーションから **jar** ファイルを作成します。なお、**jar** ファイルを使って実際にWebアプリケーションを動かしてみます。



以下の手順でサンプルアプリケーションを作成します。

- ❶ pom.xml の修正
- ❷ Java のインストール
- ❸ jar ファイルの作成
- ❹ jar ファイルの配置

❶ pom.xml の修正

実行可能な jar ファイル (**Executable Jar**) を作成するために、pom.xml を [リスト C-1](#) のように修正します。

リスト C-1 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ... (省略) ...
  ... (省略) ...
  <version>0.0.1-SNAPSHOT</version> — B
  <name>SpringBootSample</name>
  ... (省略) ...

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <excludes>
            <exclude>
              <groupId>org.projectlombok</groupId>
              <artifactId>lombok</artifactId>
            </exclude>
          </excludes>
          <executable>true</executable> — ①
        </configuration>
      </plugin>
    </plugins>
    <finalName>spring-boot-sample-${version}</finalName> — A
  </build>
</project>
```

ポイント① Executable Jar

通常の jar ファイルを作成した場合、これまで作成してきた Web アプリケーションのファイルを 1 つにまとめることができますが、この jar の中に Tomcat (Web サーバー) は含まれていません。そのため、Web アプリケーションを実行するには、各サーバーに Tomcat をインストールし、個別に設定を行なう必要があります。サーバーの台数が多いほど、Tomcat のバージョンアップや設定変更のたびに同じ作業を繰り返すことになり、設定漏れなどのミスが発生するリスクが高まります。

これに対して、**Executable Jar** (実行可能 jar ファイル) では、jar ファイルの中に Tomcat 自体も含まれています。そのため、Tomcat のインストールや設定を別途行なう必要がありません。**Executable Jar** とは、実行可能な形式の jar ファイルのことです。**jar ファイル 1 つで実行できる**ため、非常に便利です。

Maven で Executable Jar を作成するには、**executable** タグに **true** を設定します。Spring Boot では、この Executable Jar の仕組みが標準でサポートされており、特別な設定を行わなくても簡単に作成できます。

A のように、**finalName** タグを使用すると、出力される jar ファイルの名称を設定できます。なお、EL 式を使用することで、**pom.xml** 内の他のタグの値を参照できます。上記のサンプルコードでは、**B** の **version** タグの値が jar ファイル名に組み込まれます。その結果、**spring-boot-sample-0.0.1-SNAPSHOT.jar** というファイルが作成されます。

次に、ログファイルの出力で、文字コードを指定します。

リスト C-2 logback-spring.xml

```
<!-- ファイル出力 -->
<appender name="APP_LOG" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <!-- 出力先 -->
  <file>${file.path}</file>
  <!-- ローターション設定 -->
  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>${file.path.pattern}</fileNamePattern>
    <maxHistory>3</maxHistory>
  </rollingPolicy>
  <encoder>
    <charset>${charset}</charset> ← ここを新規追加
    <pattern>${format}</pattern>
  </encoder>
</appender>
```


② Javaのインストール

ここからはjarファイルを実行する必要があります。そのため、PCにJavaをインストールします。本書ではOpenJDKを使用してJava 17をインストールしますが、Java 17以降のバージョンであれば問題ありません。なお、本書の執筆時点では、EclipseでJava 17を使う場合、Java 17系のバージョンが利用されています。すでにJava 17以降のバージョンがインストールされている場合は、この手順は省略してください。

補足



OpenJDK

Javaにはいくつかの配布形態があり、ライセンスやサポート内容が異なります。特にOracle JDKは商用利用に関して注意が必要です。そのため、本書では無料で利用できるOpenJDKを使用します。

OpenJDKとは、オープンソースとして公開されているJavaの実装であり、各種団体がビルドを提供しています。本書では、AWSが提供するOpenJDKディストリビューションであるCorretto (コレット) を使用します。

まずは、AWSが提供しているOpenJDKであるCorrettoをインストールします。以下のサイトからダウンロードしてください (図C-2)。本書のサンプルではJDK 17を使用します。

- Amazon Corretto : <https://aws.amazon.com/jp/corretto/>

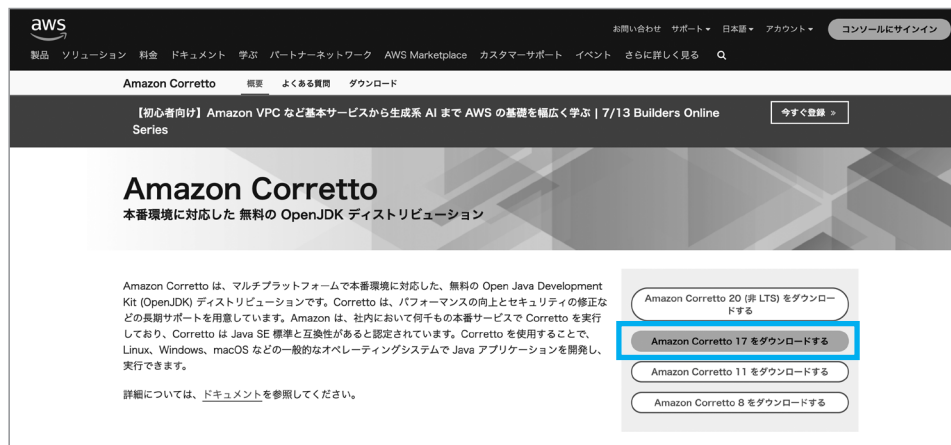


図 C-2 Correttoのダウンロード

お使いのOSに合わせて、インストーラーをダウンロードしてください (図 C-3)。

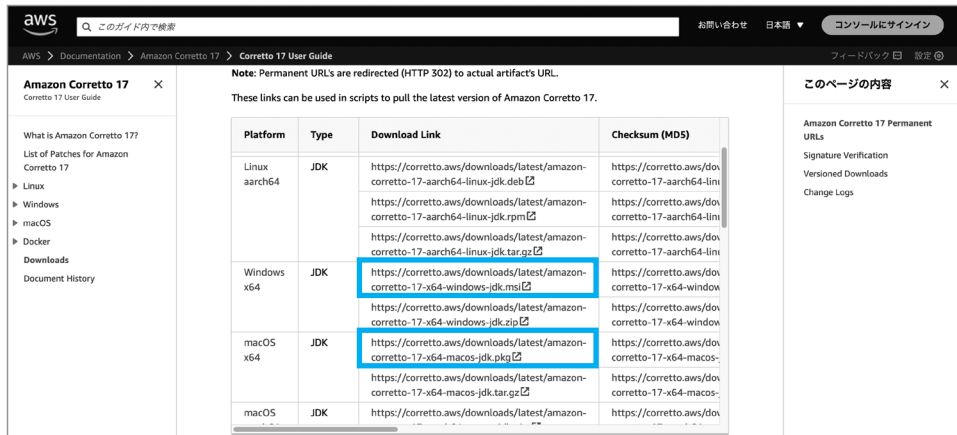


図 C-3 Correttoのインストーラーを選択

ダウンロードしたインストーラーを使ってインストールします (図 C-4)。すべてデフォルト設定のままで問題ありません。



図 C-4 Correttoのインストール

Javaのインストール後、以下のコマンドラインアプリケーションを起動します。

- Windowsの場合 ➡ コマンドプロンプト
- Macの場合 ➡ ターミナル

上記のコマンドラインアプリケーションは、以降の説明ではすべて**コマンドライン**と表記します。

まず、コマンドラインで**リストC-1**のコマンドを実行し、Javaがインストールされていることを確認しましょう。

```
java -version
```

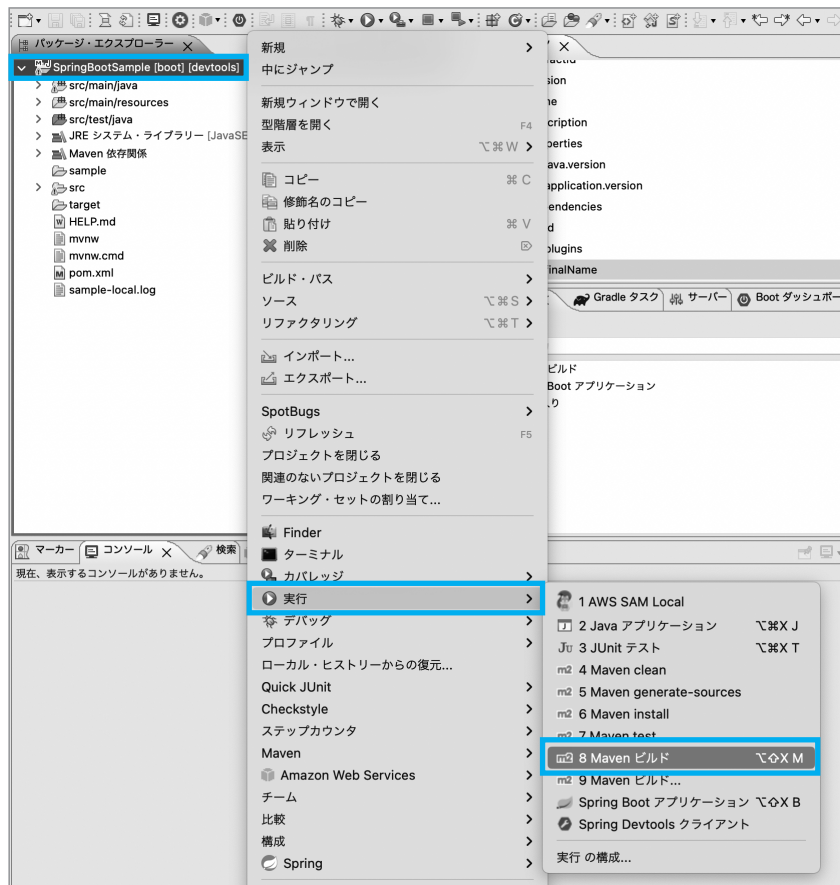
結果 ログ

```
openjdk version "17.0.12" 2024-07-16 LTS
OpenJDK Runtime Environment Corretto-17.0.12.7.1 (build 17.0.12+7-LTS)
OpenJDK 64-Bit Server VM Corretto-17.0.12.7.1 (build 17.0.12+7-LTS, mixed mode, sharing)
```

上記のように17系のバージョンが出力されれば、Javaが正しくインストールされています。

③ jar ファイルの作成

続いて、jar ファイルを作成、実行できる状態にします。Eclipseでプロジェクトフォルダーを右クリックし、[実行] → [Mavenビルド] を選択します (図C-5)。



図C-5 Mavenビルド (jar ファイルの作成)

「構成の編集」画面で、「ゴール」に「package」と入力し、[テストのスキップ] をチェックしてから、[実行] ボタンを押します (図 C-6)。



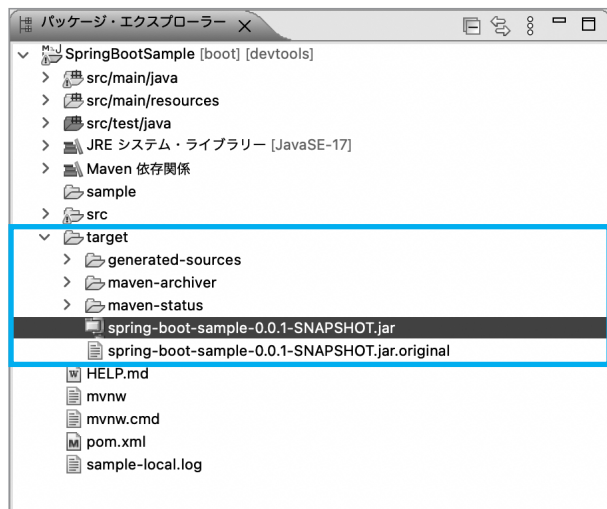
図 C-6 Mavenビルドの設定

これで、EclipseのコンソールにMavenビルドの結果が出力されます。以下のように「BUILD SUCCESS」と出力されていれば、ビルドは成功です。

結果 ログ (一部抜粋)

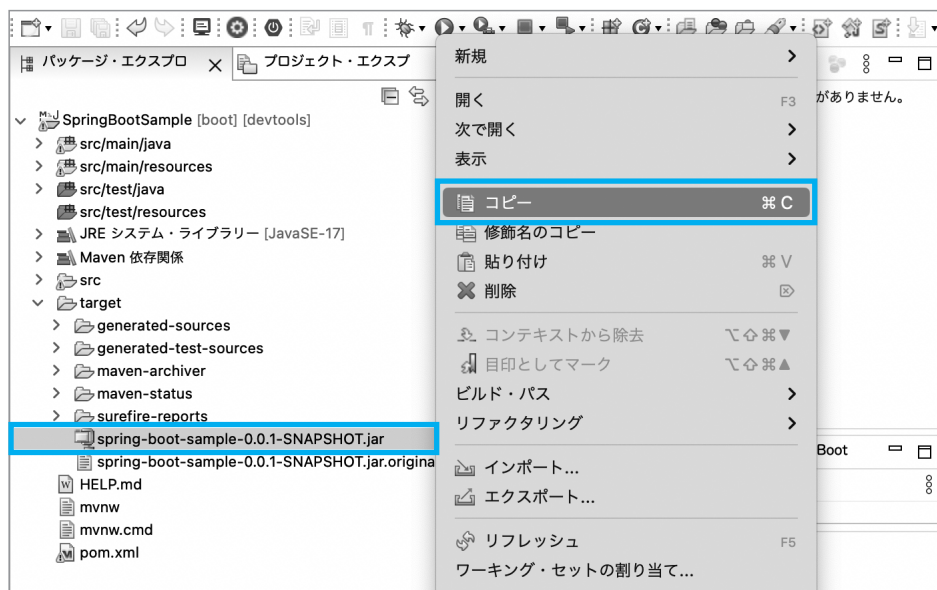
```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 4.420 s  
[INFO] Finished at: 2023-06-12T23:20:31+09:00  
[INFO] -----
```

ビルドに成功すると、**target**ディレクトリに**jar**ファイルが出力されます (図C-7)。ファイルが表示されない場合は、**target**ディレクトリを選択してキーボードの**[F5]**キーを押してください。ディレクトリの内容が更新され、ファイルが表示されます。



図C-7 Mavenビルドの結果

このあと作成する実行用フォルダーに**jar**ファイルを配置するため、コピーしておきましょう。作成された**jar**ファイルを右クリックし、**[コピー]**を選択します (図C-8)。



図C-8 jarファイルの配置

④ jar ファイルの配置

次に、以下のフォルダーを作成します。

●Windowsの場合 ➡ C: ¥work¥spring¥jar

●Macの場合 ➡ /var/spring/jar

なお、Macで/varを開くには、Finder上でキーボードから`Command` + `Shift` + `G`キーを押してください。パスを入力できるため、/varと入力します。

そして、上記で作成したフォルダーに、先ほどコピーしたjarファイルを配置します。



実行

コマンドラインで、jarファイルを実行するコマンド（Windowsは[リストC-3](#)、Macは[リストC-4](#)）を入力します。なお、EclipseでSpring Bootのアプリケーションを起動している場合は、停止しておいてください。

リストC-3 jar実行コマンド Windows

```
chcp 65001 ← ターミナルの文字コードをUTF-8に設定するコマンド
set SPRING_PROFILES_ACTIVE=local
set LOG_FILE_PATH=C:/work/spring/jar/sample-local.log
java -jar C:/work/spring/jar/spring-boot-sample-0.0.1-SNAPSHOT.jar
```

[リストC-4](#)のsetコマンドは、Windowsで環境変数を設定するコマンドです。

リストC-4 jar実行コマンド Mac

```
export SPRING_PROFILES_ACTIVE=dev
export LOG_FILE_PATH=/var/spring/jar/sample-dev.log
java -jar /var/spring/jar/spring-boot-sample-0.0.1-SNAPSHOT.jar
```

[リストC-3](#)のexportコマンドは、MacとLinuxで環境変数を設定するコマンドです。

出力結果と確認

jarファイルを実行すると、EclipseでSpring Bootを起動する際に出力されるログがコマンドラインに出力されます。以下のように起動時間が表示されれば、Spring Bootの起動に成功しています。

結果 Spring Boot起動ログ

```
2023-06-15T08:34:27.917+09:00 INFO 69586 --- [main] c.e.demo.SpringBootSampleApplication: ➡
Started SpringBootSampleApplication in 7.863 seconds (process running for 8.633)
```

起動に成功したあと、

<http://localhost:8080/login>

にアクセスしてください。今まで作成してきたログイン画面が表示されます。また、jar ファイルを置いたディレクトリにログファイルが出力されます。

ターミナルでSpring Bootを停止するためには、キーボードで以下を入力します。

- Windowsの場合 ➡ **Ctrl** + **C** キー
- Macの場合 ➡ **Control** + **C** キー

補足



Apache と Tomcat の連携

Tomcat は、**アプリケーションサーバー（APサーバー）**です。実際の運用ではWeb サーバーを構築し、APサーバーと連携させます。WebサーバーからAPサーバーへ通信が転送される仕組みになります。

現在ではNginx、Apache、IISなどのWebサーバーがあります。ここでは、WebサーバーとしてApacheを導入する場合について説明します。

ApacheからTomcatに通信を転送する場合、AJPというプロトコルが使われます。そのため、そのプロトコルの通信を許可するように設定する必要があります。この場合、[リスト C-5](#)のような設定用クラスを1つ用意します。

リスト C-5 Apache と Tomcat を連携するための設定クラス

```
import java.util.Optional;

import org.apache.catalina.connector.Connector;
import org.apache.coyote.ajp.AbstractAjpProtocol;
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class TomcatConfig {

    @Bean
    public WebServerFactoryCustomizer<TomcatServletWebServerFactory> servletContainer() {
        return server ->
            Optional.ofNullable(server)
                .ifPresent(s -> s.addAdditionalTomcatConnectors(redirectConnector()));
    }
}
```

```

private Connector redirectConnector() {
    Connector connector = new Connector("AJP/1.3");
    connector.setScheme("http");
    connector.setPort(8009);
    connector.setRedirectPort(8443);
    connector.setSecure(false);
    connector.setAllowTrace(false);

    AbstractAjpProtocol protocol = (AbstractAjpProtocol) connector.getProtocolHandler();
    connector.setSecure(false);
    protocol.setSecretRequired(false);

    return connector;
}
}

```

Apache と Tomcat 間の通信をどのような設定にするかによって、上記の設定も変更する必要があります。Connector クラスの各種 setter を呼び出すことで、設定を変更できます。

これで、jar ファイルの作成は完了です。

C-2

まとめ

この章で学んだ内容をまとめておきましょう。

→ リリースまでの作業

- ✓ ビルドとは、ソースコードから実際のサーバーで実行可能なファイルを作成することを指す。Java であれば jar ファイルや war ファイルを作成すること。
- ✓ デプロイ (deploy) とは、「配備する」という意味。開発現場では、「アプリケーションをサーバー上で実行できる状態にすること」をデプロイと言う。
- ✓ デプロイが完了すると、実際にアプリケーションが動くかどうか、テストを行なう。環境やアプリケーションによってテストの内容は異なる。
- ✓ デバッグとは、バグを修正する作業のこと。
- ✓ リリースとは、エンドユーザーがアプリケーションを使えるようにすること。本番環境でアプリケーションを動かせるようにする。そのために、既存のシステムを停止して、ユーザーからのアクセスを停止する。

→ Executable Jar

- ✓ Executable Jar とは、実行可能な形式の jar ファイルのこと。
- ✓ Executable Jar は jar ファイルの中に Tomcat も含まれるため、サーバー側に別途 Tomcat をインストールする必要がなく、jar ファイル 1 つでどこでも動かせるメリットがある。

付録D

テスト

付録Dでは、まず知っておきたいテストの基礎知識について簡単に解説します。テストは、それだけで本が何冊も書けるぐらい奥が深い分野です。そのため、テストコードの作成部分を重点的に説明していきます。

まずはテストの概要について説明し、そのあとテストコードを作成していきます。

D-1

テストの概要

テストにはいくつかの種類があります。テストの種類に応じて目的と範囲が変わってくるため、それらについて説明します。テストの概要について知っている場合は、この節は読み飛ばしてください。



開発工程とテストの種類

大規模なシステム開発は、いまだに**ウォーターフォールモデル**で進めることが多いです。ウォーターフォールモデルでは、上流から下流へと一方向に開発工程を進めていきます。工程が終わったら次の工程に進みます。基本的に、前の工程に戻ることはありません。

ウォーターフォールモデルの開発工程は、以下の順番で進めていきます。

- ① 要件定義
- ② 基本設計
- ③ 詳細設計
- ④ 開発

- ⑤ 単体テスト
- ⑥ 結合テスト
- ⑦ システムテスト
- ⑧ 受け入れテスト

この開発工程とテストをひもづけた **V字モデル** という考え方があります（図 D-1）。

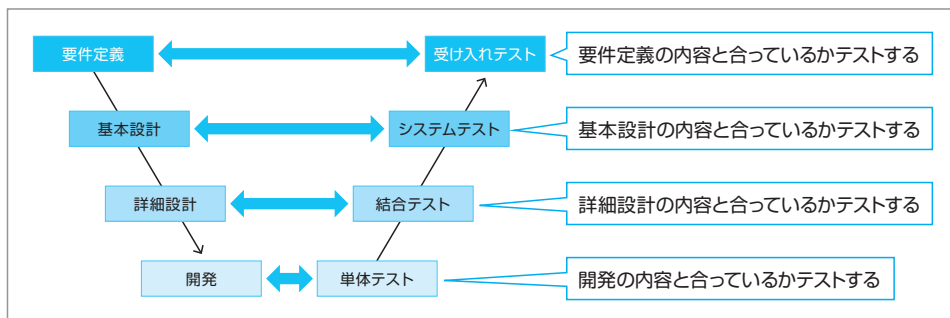


図 D-1 V字モデル

V字モデルでは、以下のように各工程に対応したテストを行ないます。

- 要件定義の内容は受け入れテストで確認
- 基本設計の内容はシステムテストで確認
- 詳細設計の内容は結合テストで確認
- 開発（実装）の内容は単体テストで確認

これにより、要件定義や設計などの内容がシステムに反映されているかを確認できます。また、各テストの範囲や内容を明確にすることができます。

上記で挙げたテストでは、以下の内容について確認します。

● 単体テスト

単体テストでは、各モジュールやコンポーネントが正しく動作するかを確認します。Webアプリケーションで言えば、1つの機能や部品（クラスやメソッドなど）を対象にテストします。たとえば、ユーザー登録画面のバリデーションや表示内容、データベースへのアクセスなどをテストします。

● 結合テスト

結合テストでは、モジュールやコンポーネントを組み合わせたときの動作を検証します。Webアプリケーションでは、画面間の動きが正しいかどうかをテストします。たとえば、登録画面で入力した内容が確認画面に表示されるかをテストします。

● システムテスト

システムテストでは、ユーザーが実際に行なう操作（利用シナリオ）に基づいてシステム全体をテストします（**総合テスト**と呼ばれることもあります）。たとえば、「ユーザー登録をしてから、そのユーザーで正しくログインできるか」といった機能要件（システムに必要な仕様）を確認します。

また、**非機能要件**もテストの対象となります。**非機能要件**とは、機能以外にシステムに求められる仕様のことです。

- 画面のレスポンスが3秒以内であること
- 1万ユーザーが同時アクセスしてもシステムが正常に稼働すること

など、「レスポンス速度」や「高負荷時の安定性」といった要件がこれに当たります。非機能要件については、IPA（独立行政法人情報処理推進機構）が公開している「非機能要求グレード」という資料^{〔※1〕}が非常に参考になります。

● 受け入れテスト

システムテストでは、システムが要件を満たしているかを、最終的に発注元（エンドユーザー）が確認します。これが完了すると「検収完了」となります。

補足



▶ アジャイル開発

ウォーターフォールモデルは基本的に後戻りができないため、最終段階で開発者とユーザーのあいだの認識の違いが発覚することがあります。そうなると、修正コストが膨大になります。

これを解消するのが **アジャイル開発**です。機能を少しずつ（要件定義～テストのサイクルを繰り返して）作っていくため、ユーザーが頻繁に実際の動作を確認でき、認識の相違を最小限に抑えられます。

アジャイル開発では、短い期間で何度も開発とテストを繰り返すため、手動テストだけでは時間が足りなくなります。そのため、自動的に実行できるテストコードの作成が大前提となります。



テストコード

ここまで説明したテストは、手動で行なう場合もあれば、プログラムによって自動化する場合もあります。ここでは、テストを自動化するための**テストコード**について説明します。

テストでバグが見つかり修正を行なった際、その箇所だけでなく、それまでパスしたテスト

〔※1〕 非機能要求グレード（IPA）
<https://www.ipa.go.jp/archive/digital/iot-en-ci/jyouryuu/hikinou/>

をすべてやり直す必要があります。修正によって他の正常な箇所に影響（デグレード）が出ていないかを確認するためです。このようなテストを**レグレッションテスト（回帰テスト）**と言います。

開発初期はバグが多く、修正のたびに手で全テストをやり直すと膨大な労力がかかります。また、単調な繰り返し作業はミスやチェック漏れの原因にもなります。

テストコードを用意してテストを自動化すれば、コンピューターが命令通りに何度でも正確に実行してくれるため、これらの問題を解決できます。ただし、テストコードを作成すると全体の開発量が2倍以上になるというデメリットもあります。ソースコードだけでなくテストコード自体の保守も必要になるため、プロジェクトの実情によっては、あえて手動テストが選ばれるケースも少なくありません。

補足



▶ テスト駆動開発（TDD）

実際のプログラムを作るよりも先にテストコードを書き、そのテストを通るように実装を進める手法を**テスト駆動開発（TDD：Test Driven Development）**と言います。

「どうテストするか」を先に考えることで、自然と呼び出しやすく設計のきれいなメソッドやクラスが作られ、結果として全体の品質向上につながります。

テストコードでのテストの種類

テストコードによる自動テストには、主に以下の種類があります。

● ユニットテスト（単体テスト）

Javaのクラス単位でテストを行なう手法です。クラスごとにテストコードを用意します。たとえば、クラスのメソッドを呼び出して、戻り値が想定通りであることを確認します。

テストの内容はシンプルですが、作成するテストコードの量は一番多くなります。開発工程の「単体テスト」に相当する範囲をカバーするのが一般的です。

● インテグレーションテスト

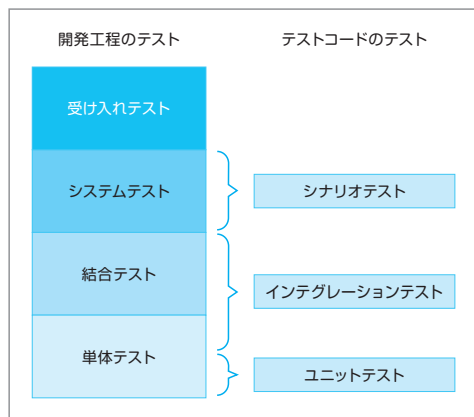
関係するモジュール同士を結合したテストを行なう手法です。Webアプリケーションでは、1つ、または複数の画面にまたがった動作を検証します。たとえば、ユーザー登録画面を対象として、バリデーションが正しく動くか、実際にユーザーが登録されるかを確認します。登録後にログイン画面へ戻ることもテストの範囲に含まれます。

開発工程の「単体テスト」から「結合テスト」に相当する範囲のテストを行ないます。

● シナリオテスト

ユーザーの実際の操作シナリオに沿ったテストを行なう手法です。開発工程の「システムテスト」に相当する範囲のテストを行ないます。

開発工程のテストと、テストコードにおけるテストの関係を表わすと図D-2のようになります。

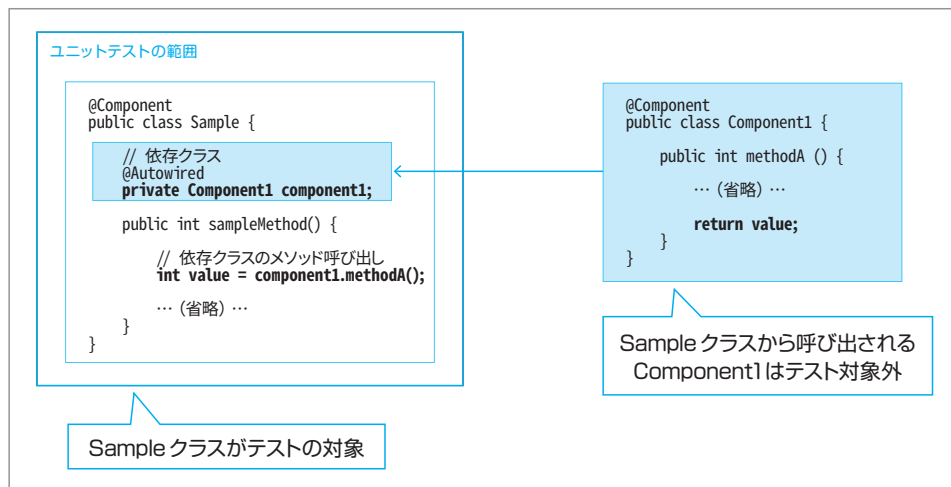


図D-2 開発工程のテストとテストコードにおけるテストの関係

プロジェクトやアプリケーションによって、テストの範囲は変動するため、上記はあくまでも簡単な目安と考えてください。

ユニットテストとモック

ユニットテストでは、特定の「1つのクラス」だけをテスト対象とします（図D-3）。

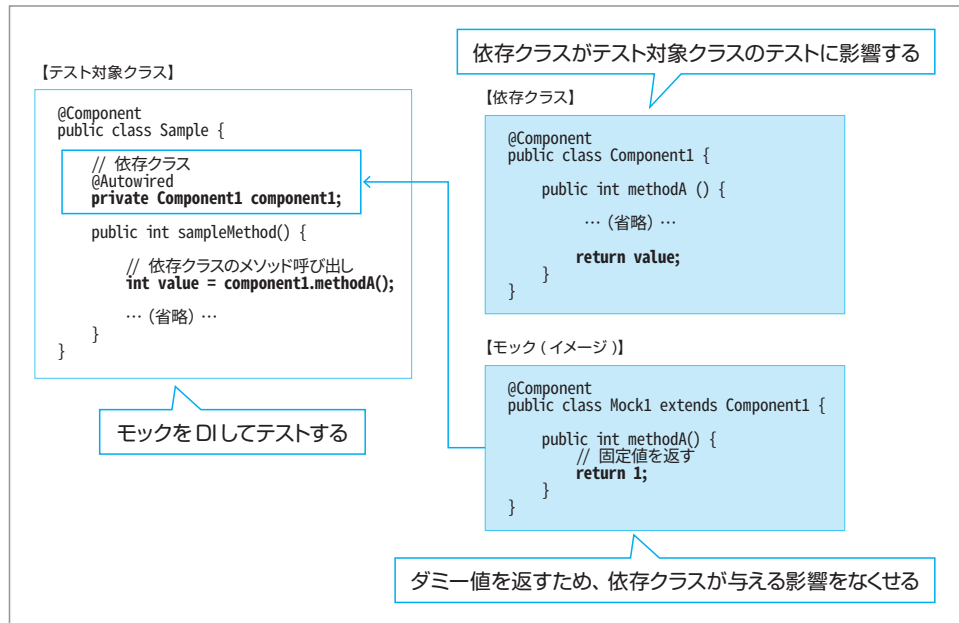


図D-3 ユニットテストの範囲

とはいえ、あるクラスから別のクラスのメソッドを呼び出すことは多々あります。しかし、呼び出し先のクラスはテストの範囲外です。そのため、呼び出し先のクラスによる影響をなくしてテストをしなければなりません。その影響とは、たとえば以下のようなものです。

- テストとしては「A」という値を返してほしいが、呼び出し先が未完成などの理由でどのような値が返ってくるかわからない
- 呼び出し先のクラスでデータベースを使用しているため、テストのために膨大なデータを用意しなければならない

そこで、ユニットテストでは**モック**を使用します。**モック**とは、あらかじめ決めたダミーの値を返してくれる「身代わり」のオブジェクトのことです (図D-4)。



図D-4 モックとは？

DI（依存性の注入）を利用すれば、呼び出し先のクラスをモックに差し替えることも簡単にできます。実際には、モック用のクラスを一から作る必要はありません。モックを簡単に作成するためのフレームワークを使用し、「Xメソッドが呼び出されたらYという値を返す」という設定を行なうだけです。

🔌 テストコードを作成するフレームワーク

テストコードを作成するためには、様々なライブラリやフレームワークがあります。ユニットテストを実行するための **JUnit**（ジェイユニット）や、モックを作成するための **Mockito**（モキート）などが代表的です。

Springでは、これらのライブラリを簡単に使えるようにするための **Spring Test** というフレームワークが用意されています。Spring Testでは、Springで作成したアプリケーションを

テストしやすくするための機能が豊富に提供されています。たとえば、Spring Securityと連携して「一般権限ユーザーがログイン済みの状態」として画面をテストすることも可能です。

補足



JunitとMockito

JUnitは、Javaでユニットテストを自動化するための標準的なフレームワークです。Javaでテストコードを書く際に最も広く利用されています。

Mockitoは、ユニットテストにおいて依存するクラスの「モック」を生成し、特定の挙動をさせるために特化したフレームワークです。

D-2

ユニットテストコードの作成① ——JPAのリポジトリ

ここからは、実際にテストコードを作成していきましょう。この付録では、先ほど解説した**ユニットテスト**（**単体テスト**）の実装方法を詳しく学んでいきます。

一般的に、Webアプリケーションのユニットテストは、プログラムの役割（レイヤー）ごとに作成します。ここでは、以下の3つのレイヤーに分けて、それぞれのテストの作り方を順に説明していきます。

- リポジトリ（データアクセス層）
- サービス（ビジネスロジック層）
- コントローラー（制御層）

この節ではまず、データベースとのやり取りを直接担当するJPAのリポジトリ（UserRepository）のユニットテストコードを作成します。



サンプルアプリケーションの作成

サンプル ChD_Test/D-2_1

～

サンプル ChD_Test/D-2_4

以下の手順で、テスト環境の構築から実装、実行までを進めていきます。

- ① ファイルなどの作成
- ② テスト用設定の作成
- ③ テストコードの作成 (1) **D-2_1**
- ④ テストの実行 (1)

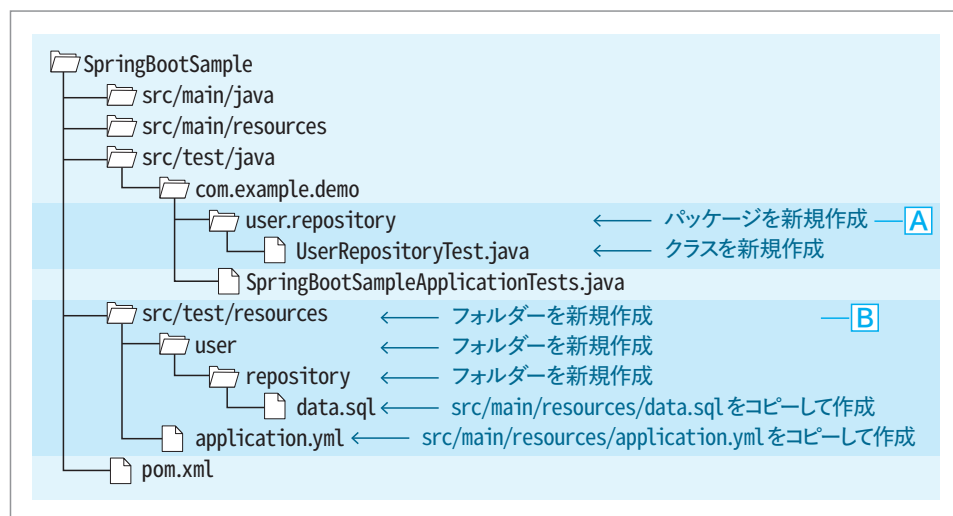
- ⑤ テストコードの作成 (2) [D-2 2](#)
- ⑥ テストの実行 (2)
- ⑦ テストコードの作成 (3) [D-2 3](#) [D-2 4](#)
- ⑧ テストの実行 (3)

テストコードの書き方をより深く理解するために、「テストコードの作成」と「テストの実行」を細かく繰り返しながら進めます。

実際の現場でも、「テストコードを少し書いては実行して、期待通りに動くか確認する」というサイクルを回すのが、バグの早期発見につながる最も効率的な進め方です。

① ファイルなどの作成

図D-5の構成となるように、ファイルなどを作成してください。



図D-5 パッケージ・エクスプローラーの構成

テストコードは、[A](#)のようにsrc/testの下に作成します。一般的には、src/main/javaと同じパッケージ構成にし、クラス名の末尾に「Test」を付けるのが通例です。たとえば、com.example.demo.user.repository.UserRepositoryをテストする場合は、同じ名前のパッケージをsrc/test/java内に作成し、その中にUserRepositoryTest.javaというファイルを作成します。

[B](#)のフォルダーは、[図D-6](#)の手順で作成します。Eclipseでプロジェクトフォルダーを右クリックし、[新規] → [フォルダー] を選択してください。

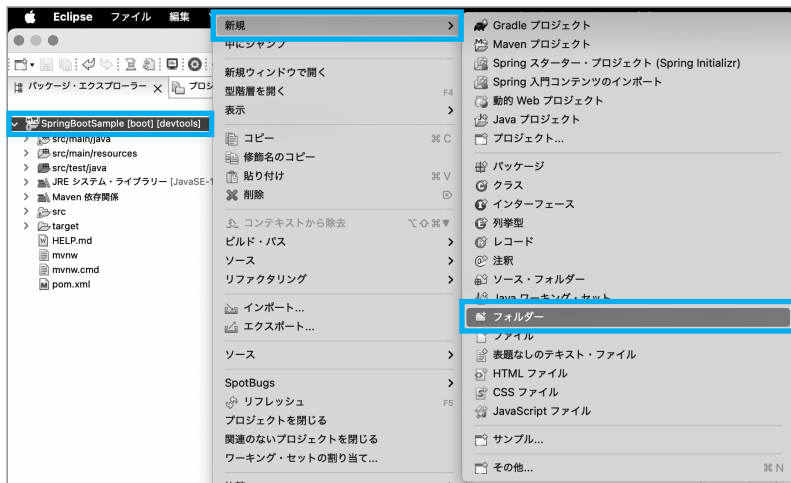


図 D-6 test用resourcesの作成

「src/test」フォルダーを選択し、フォルダー名に「resources」と入力します (図 D-7)。

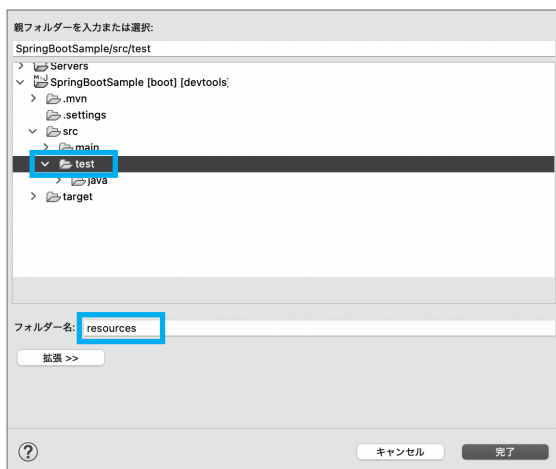


図 D-7 ソース・フォルダーの作成

これでsrc/test/resourcesが作成されます (図 D-8)。

このresources配下に置いたファイルは、テスト実行時にのみ有効になります。通常通りSpring Bootを起動した際には、src/test/resources配下の設定ファイルは読み込まれません。

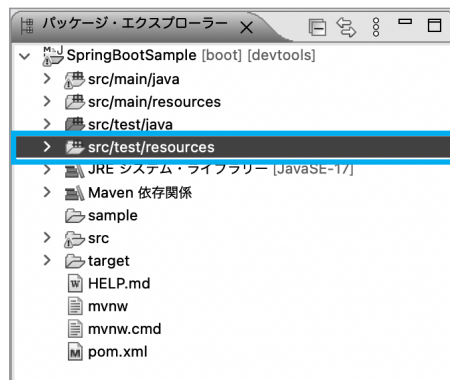


図 D-8 作成されたsrc/test/resources

また、テスト実行時はsrc/main/resourcesよりもsrc/test/resourcesの設定が優先されます。たとえば、両方のフォルダにapplication.ymlが存在する場合、テスト時にはsrc/test/resources/application.ymlの設定が適用されます。一方で、src/test側にはない設定値はsrc/main側のものがそのまま有効になります。

つまり、「テストのときだけ設定を上書きしたい」という場合に、このsrc/test/resourcesを活用するということです。

② テスト用設定の作成

テスト用の設定ファイルを作成します。src/main/resources/application.ymlをコピーして、src/test/resources/application.ymlを作成してください（リストD-1）。背景色が変わっている部分が修正箇所です。

リストD-1 src/test/resources/application.yml

```
spring:
  ... (省略) ...
  # DB初期化
  sql:
    init:
      encoding: UTF-8
      mode: embedded
      schema-locations: classpath:schema.sql
      data-locations: '' ①
  ... (省略) ...
```

ポイント① テストの独立性

テストを実行する前には、テストデータの準備が必要になることがあります。しかし、テストケースによって必要なデータはそれぞれ異なります。

このサンプルアプリケーションの通常実行時にはdata.sqlで初期データを投入していましたが、テストにおいてはそのデータが予期せぬ影響（邪魔）を与える可能性があるため、初期データの自動投入を無効化しています（サンプルコードでは、シングルクォーテーション2つ''を設定して空文字にしています）。

これは、テストコードはそれぞれが独立している必要があるためです。たとえば、以下の2つのテストを連続で行なうとします。

- ユーザー検索
- ユーザー更新

もしユーザーの更新をしたあとに同じユーザーを検索すると、検索結果が変わってしまいます。その結果、本来成功するはずの「ユーザー検索」のテストが想定通りに動作せず、失敗してしまう可能性があります。

なお、テストコードの実行順序を指定することも可能ですが、基本的にはテストの実行順序に依存しないように設計するのが鉄則です。テストが他のテストに影響を与えないようにするためには、テストごとに使用するデータを個別に用意する必要があります。各テストでテストデータを用意する方法については後述します。

③ テストコードの作成 (1)

サンプル ChD_Test/D-2_1

それでは、`UserRepository`のテストコードを作成します。まずは、Spring Data JPAが自動生成する `findAll()` メソッドや `save()` メソッドの動作を確認するためのテストを記述します(リストD-2)。

リストD-2 `UserRepositoryTest.java`

```
package com.example.demo.user.repository;

import static org.assertj.core.api.Assertions.*; —①

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import com.example.demo.user.domain.model.MUser;

@DataJpaTest —②
public class UserRepositoryTest {

    @Autowired
    private UserRepository repository;

    @Test —③
    public void save_user() {
        // 準備
        MUser testUser = new MUser();
        testUser.setUserId("save-test@co.jp");
        testUser.setUserName("saveテスト");
        testUser.setPassword("save-test-pass");
        testUser.setRole("ROLE_GENERAL"); —④
    }
}
```

```
// 実行
MUser user = repository.save(testUser);

// 検証
assertThat(user.getId()).isEqualTo("save-test@co.jp");
assertThat(user.getUserName()).isEqualTo("saveテスト");
assertThat(user.getPassword()).isEqualTo("save-test-pass");
assertThat(user.getRole()).isEqualTo("ROLE_GENERAL");
}
}
```

④

ポイント① staticインポート

import文に**static**を付けることで、**staticインポート**を使用できます。**staticインポート**とは、クラス名を省略してstatic変数やstaticメソッドを使用できるようになるJavaの構文です。

たとえば、staticインポートを使用しない場合と使用する場合で比較すると、以下のようなソースコードになります。

staticインポートを使用しない場合

```
Assertions.assertThat(...) ← 毎回Assertionsを付けなければいけない
```

staticインポートを使用する場合

```
assertThat(...) ← Assertionsを省略できる
```

Assertionsクラスでは、テストの実行結果を検証するためのメソッドが多数用意されています。ユニットテストでは、これらのメソッドを頻繁に使用するため、staticインポートを利用することで、ソースコードの記述量を減らすことができます。

なお、staticインポートは通常のimport文と異なり、Eclipseが自動補完してくれない場合があるため、開発者自身で記述する必要があります。

ポイント② @DataJpaTest

リポジトリにJPAを使用している場合、テストクラスに**@DataJpaTest**アノテーションを使用します。それにより、以下のことが可能となります。

- (1) Spring Bootを起動した状態でテストコードが実行される
- (2) テストごとにトランザクションがロールバックされる

(1) Spring Bootを実行してからテストコードが実行される

@DataJpaTest アノテーションを使用すると、Spring Bootが起動した状態でテストが実行されます。そのため、BeanをDI（注入）することができます。サンプルコードでは、UserRepositoryにインスタンスがDIされます。なお、コンストラクタインジェクションは使用できないため、@Autowiredによるフィールドインジェクションを使用しています。

また、Spring Bootの起動時にテーブル作成用のSQLが実行されるため、テスト中にそれらのテーブルを利用できます。

(2) テストごとにトランザクションがロールバックされる

テストには独立性が必要です。**テストの独立性**とは、テストの結果が他のテストに影響を与えないことを指します。

サンプルコードでは、JPAのsave() メソッドを使用してユーザー登録のテストを行っています。つまり、データが1件追加されます。この状態のまま別のテスト（たとえば、全件検索）を行なうと、想定よりもデータが1件多い結果となってしまいます。

このような問題を防ぐために、テストごとに**トランザクション**（データベース操作のひとまとまり）を開始し、終了時に自動で**ロールバック**（データの巻き戻し）を行ないます。これにより、データベースの状態をテスト開始前に戻すことができます。

補足



H2データベース以外のデータベースを使用する場合

@DataJpaTest アノテーションは、デフォルトでは組み込みのインメモリデータベース（H2データベース）を使用します。そのため、PostgreSQLやMySQLなど、実際に利用するデータベースに接続してテストを行ないたい場合は、以下のようにreplace = Replace.NONEを指定するアノテーションを追加します。

H2データベース以外のデータベースを使用する場合

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class UserRepositoryTest {
    ... (省略) ...
}
```

ポイント ③ @Test

@Test アノテーションが付いているメソッドだけが、テストコードとして実行されます。@TestはJUnitが提供している最も基本的なアノテーションです。

ポイント④ 4フェーズテスト

テストコードは**4フェーズテスト**という考え方に基づいて作成します。これは、「前処理 → 実行 → 検証 → 後処理」の4つの流れでテストを記述する、広く使われている手法です。[表D-1](#)のフェーズを意識してテストコードを記述します。

表D-1 4フェーズテスト

No	フェーズ	説明
1	前処理	テスト対象のメソッドを実行するための準備を行なう。具体的には、インスタンスの生成や、検索テスト用のデータをデータベースに投入する作業などがこれにあたる。
2	実行	テスト対象のメソッドを実際に呼び出し、結果（戻り値）を取得する。
3	検証	実行結果が想定通り（期待値）であることを確認する。戻り値のチェックだけでなく、データベースに正しく保存されたか、特定のメソッドが呼び出されたかなどを、 Assertions クラス（ assertThat など）を使って検証する。
4	後処理	テストの実行結果や前処理で行なった内容を元に戻す。後処理の目的は、他のテストコードに影響を与えないようにすることである。たとえば、テストで登録したデータや作成したファイルを削除し、環境を元の状態に戻す。これにより、後続のテストが前のテストの影響を受けるのを防ぎ、テストの独立性を保つことができる。

ただし、すべてのフェーズを必ずしも実行する必要はありません。前処理や後処理が不要な場合もあります。いずれにしても、どのフェーズを行なっているのかわかるように、コメントを書いておくことをおすすめします。

本書のサンプルコードでは、**前処理**（準備）として登録データを用意しています。そして、**save()**メソッドを呼び出してテストを**実行**しています。テストの実行とは、テスト対象のメソッドを呼び出すことです。

検証は、AssertJというライブラリの**Assertions**クラスが持つ**assertThat()**メソッドを使用しています。**assertThat()**は以下のように使用します。

assertThat()の書式

```
assertThat(実際の値).isEqualTo(期待値)
```

assertThat()に続く**isEqualTo()**メソッドでは、実際の値と期待値が同じことをチェックします。値が一致しない場合は、その時点でテストは失敗となります。

他にも、以下のようなメソッドがあります。

- **isTrue()** ➡ 実際の値がTrueかどうかをチェック
- **isFalse()** ➡ 実際の値がFalseかどうかをチェック
- **isNotEqualTo(期待値)** ➡ 実際の値が期待値でないことをチェック

これ以外にも多くのメソッドが用意されています。詳細はAssertJの公式サイトなどを

チェックしてください。

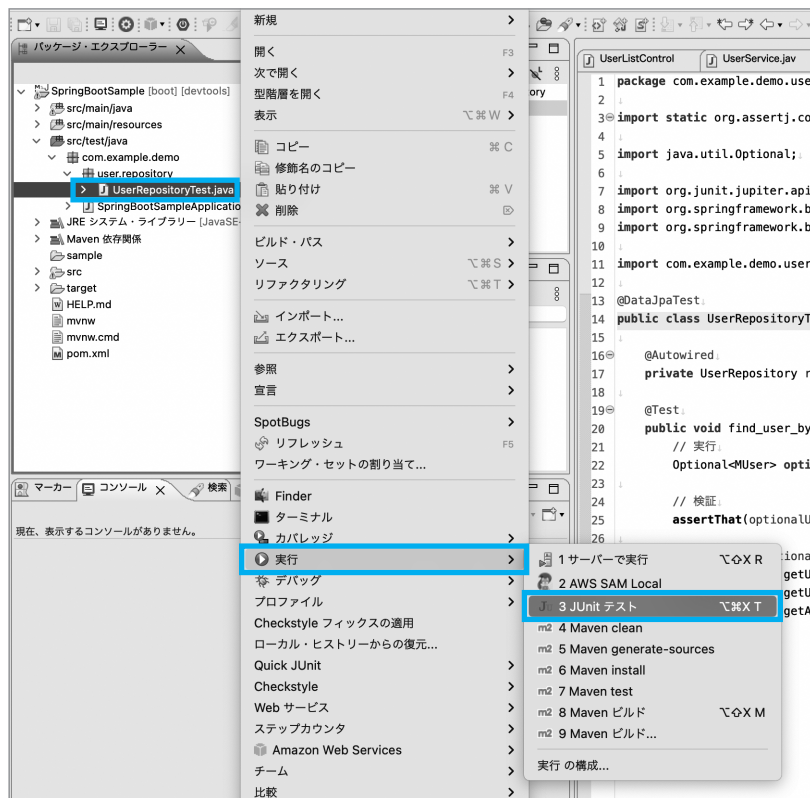
- AssertJの公式サイト

<https://assertj.github.io/doc/>

なお、テストの実行でデータを登録していますが、`@DataJpaTest` アノテーションによってデータベースの変更は自動でロールバックされるため、通常は後処理が不要です。

4 テストの実行 (1)

テストコードを作成したら、実際にテストをしてみましょう。Eclipseで`UserRepositoryTest.java`を右クリックし、[実行] → [JUnitテスト] を選択します (図D-9)。



図D-9 JUnitの実行

JUnitの実行と同時にSpring Bootが実行されて、コンソールにログが出力されます。そして、テストの実行が終わると、「JUnit」タブに実行結果が出力されます (図D-10)。

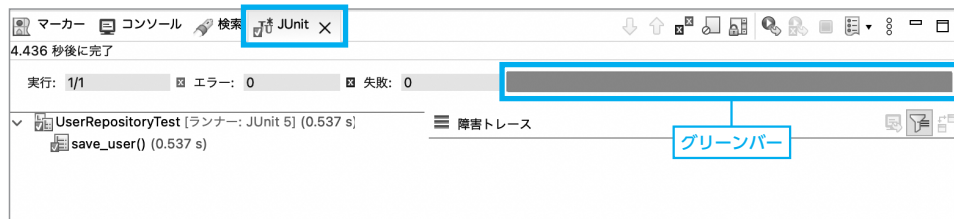


図 D-10 JUnit実行結果

ここには、テストの総実行数や成功数、失敗数などの情報が表示されます。また、実行されたメソッド名やかかった時間も確認できます。

テストがすべて成功すると、進捗バーが緑色になります。これを**グリーンバー**と呼び、大量のテストを実行した際でも一目で「すべて成功した」ことがわかります。

逆に、テストが1つでも失敗した場合は赤色の**レッドバー**が表示されます。その際は、どのテストが失敗したのかをリストから特定し、修正を行ないます。

なお、実行時にコンソールに以下のような警告が出力されることがありますが、テストの動作自体には影響ありませんので無視してください。

実行結果 コンソールに出力される警告文

```
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
```

⑤ テストコードの作成 (2)

サンプル ChD_Test/D-2.2

次に、ユーザー全件検索（`findAll()`）のテストを追加します。全件検索のテストでは、あらかじめデータベースに複数のデータが存在している必要があるため、SQL ファイルを使ってテストデータを投入します（リスト D-3）。

リスト D-3 UserRepositoryTest.java

```
... (省略) ...

import java.util.List;
import org.springframework.test.context.jdbc.Sql;

@DataJpaTest
public class UserRepositoryTest {

    ... (省略) ...

    @Test
    public void save_user() {
        ... (省略) ...
    }
}
```



```

    }

    @Test
    @Sql("/user/repository/data.sql") ①
    public void find_all(){
        // 実行
        List<MUser> userList = repository.findAll();
        // 検証
        assertThat(userList.size()).isEqualTo(11);
    }

```

ポイント ① @Sql

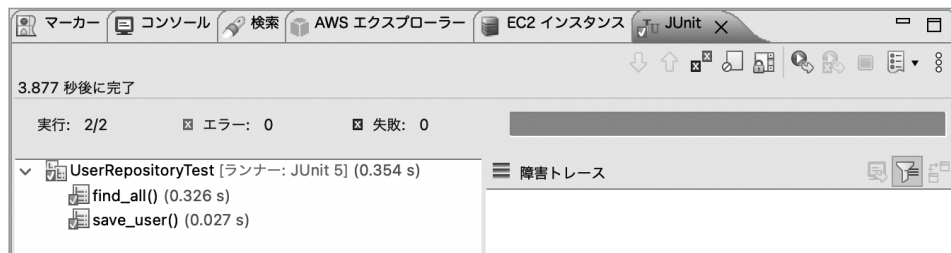
@Sql アノテーションを使用すると、指定したSQLファイルを実行してからテストメソッドを起動できます。ファイルパスはsrc/test/resourcesからの相対パスで指定します。この例では、用意しておいたsrc/test/resources/user/repository/data.sqlが「前処理」として実行されます。

なお、このアノテーションを使用する際は、テストの独立性を保つために@Transactionalを併用するのが一般的です。そうしないと、実行されたSQLの内容がデータベースに残ってしまい、他のテストに影響を与えてしまう（テストの独立性が失われる）可能性があります。

このサンプルでは、クラスに付与した@DataJpaTestに@Transactionalの機能が含まれているため、テスト終了後に@Sqlで投入したデータも自動的にロールバック（巻き戻し）されます。

⑥ テストの実行 (2)

それでは、再度、テストを実行してみましょう。EclipseでUserRepositoryTest.javaを右クリックし、[実行] → [JUnitテスト] を選択します (図D-11)。



図D-11 JUnit実行結果

グリーンバーが表示されました。これにより、最初に作成した `save_user()` と、新たに追加した `find_all()` の両方のテストが、互いに影響を与えることなく（テストの独立性を保ったまま）成功したことが確認できました。

⑦ テストコードの作成 (3)

サンプル ChD_Test/D-2_3

サンプルアプリケーションでは、独自のクエリーを実行するメソッドとして、`UserRepository` に `updateUser()` メソッドを追加しています。次は、このメソッドのテストコードを追加しましょう（リストD-4）。

`updateUser()` は、指定されたユーザーのユーザー名とパスワードを変更するメソッドです。テストの手順としては、まず変更対象となるユーザーを事前に登録（準備）し、そのあとにメソッドを実行します。実行後は、Spring Data JPAが提供する `findById()` メソッドを使用してデータベースの最新状態を取得し、正しく更新されているかを検証します。

リストD-4 `UserRepositoryTest.java`

```
... (省略) ...

import java.util.Optional;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository repository;

    @Autowired
    private TestEntityManager entityManager; ①

    @Test
    public void save_user() {
        ... (省略) ...
    }

    @Test
    @Sql("/user/repository/data.sql")
    public void find_all(){
        ... (省略) ...
    }

    @Test
    public void update_user() {
```

```
// 準備
MUser testUser = new MUser();
testUser.setUserId("update-test@co.jp");
testUser.setUserName("updateテスト");
testUser.setPassword("update-test-pass");
testUser.setRole("ROLE_GENERAL");
this.entityManager.persist(testUser); ①

// 実行
int count = repository.updateUser("update-test@co.jp", "password", "updateテスト2");
// 検証
assertThat(count).isEqualTo(1);

Optional<MUser> optionalUser = repository.findById("update-test@co.jp");
MUser user = optionalUser.get();

assertThat(user.getUserId()).isEqualTo("update-test@co.jp");
assertThat(user.getUserName()).isEqualTo("updateテスト2");
assertThat(user.getPassword()).isEqualTo("password");
assertThat(user.getRole()).isEqualTo("ROLE_GENERAL");
}
}
```

ポイント ① TestEntityManager

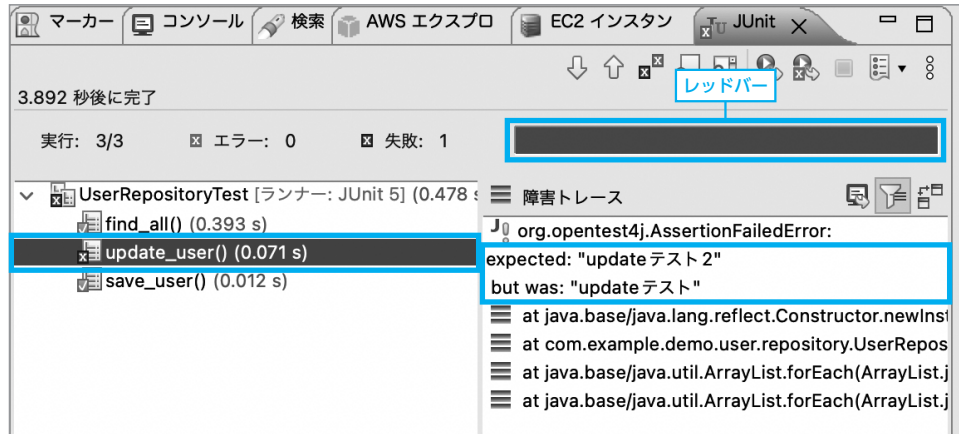
JPAでは、**EntityManager**というインターフェースを介して、データベースのデータとエンティティのインスタンスを管理（永続化）しています。本来、この操作にはJPAの深い知識が必要ですが、Spring Data JPAがその複雑な部分を隠蔽してくれています。

TestEntityManagerは、ユニットテスト用に設計されたツールで、通常の**EntityManager**よりも簡単にデータの登録や削除が行なえます。**@DataJpaTest**アノテーションを使用している環境でDIして利用可能です。

このサンプルでは、**persist()**メソッドを使用してテストデータをデータベースに直接登録（永続化）し、更新テストの「準備」を整えています。これを使うことで、テスト対象のリポジトリメソッドを呼び出す前に、確実に「更新対象のデータ」をメモリ上のデータベースに存在させることができます。

⑧ テストの実行 (3)

それでは、更新メソッド (`updateUser()`) のテストを実行してみましょう。Eclipseで `UserRepositoryTest.java` を右クリックし、[実行] → [JUnitテスト] 選択します (図D-12)。



図D-12 JUnit実行結果

`updateUser()` メソッドのテストが失敗し、レッドバー (赤色) が表示されました。

ここで失敗したテストケース (`update_user`) を選択すると、詳細を確認できます。この画面では、期待値 (更新後の名前) が "update テスト 2" であるのに対し、実際の値が更新前の "update テスト" のままになっています。

実際のアプリケーション画面では正常に更新できていましたが、テストコード上ではなぜか更新が反映されていません。この原因はJPAの仕組みにあります。JPAでは、`EntityManager` がパフォーマンス向上のために「いつデータベースを更新するか」を判断しており、更新クエリーが即座に発行されない (遅延される) ことがあるからです。

◆ テストの修正と実行

サンプル ChD_Test/D-2_4

そこで、`updateUser()` メソッドの実行直後にデータの更新が即座に反映されるよう、テストコードを修正します (リストD-5)。

リストD-5 UserRepository.java

```
public interface UserRepository extends JpaRepository<MUser, String> {

    /** ユーザー更新 */
    @Query("update MUser"
        + " set"
        + "   password = :password"
        + "   , userName = :userName"
```

```
+ " where"
+ "    userId = :userId")
@Modifying(clearAutomatically = true)
public Integer updateUser(@Param("userId") String userId,
    @Param("password") String password,
    @Param("userName") String userName);
}
```

補足



clearAutomatically

JPAでは、パフォーマンス向上のためにSQL（更新クエリー）をすぐに実行せず、メモリ上に保持しておくことがあります。今回のテストで更新内容がデータベースに反映されていなかったのは、この仕組みが原因です。

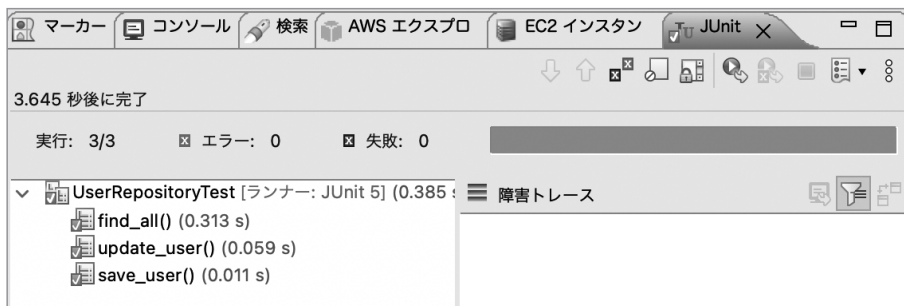
@ModifyingアノテーションのclearAutomatically属性をtrueに設定すると、クエリ実行後に永続性コンテキスト（キャッシュのようなもの）が自動的にクリアされ、最新の状態がデータベースと同期されるようになります。デフォルト値はfalseです。

JPAの内部的なキャッシュ管理や同期の仕組みに興味がある方は、以下のサイトなどを参照してみてください。

- [TERASOLUNA Global Frameworkの機能詳細 5.2. データベースアクセス（JPA編）](https://terasolunaorg.github.io/guideline/public_review/ArchitectureInDetail/DataAccessJpa.html#id82)

https://terasolunaorg.github.io/guideline/public_review/ArchitectureInDetail/DataAccessJpa.html#id82

それでは、再度、テストコードを実行してみましょう（図D-13）。



図D-13 JUnit実行結果

すべてのテストに成功したため、レッドバーからグリーンバーに変わりました。これで、独自の更新クエリー（updateUser）も正しくデータベースと同期され、検証をパスしたことが証明されました。

このサンプルでは代表的なケースのみを作成しましたが、実際にはさらに多くのテストケースが必要です。たとえば、同じメソッドであっても「存在しないユーザーIDを指定した場合

(異常系)」や「バリデーション限界の長い名前を入力した場合（境界値）」など、何パターンものテストケースを検討する必要があります。

同様に、検証内容（`assertThat`）も、単にメソッドの戻り値をチェックするだけでなく、「更新した箇所以外のデータが壊れていないか」や「関連するテーブルの内容も正しく書き換わっているか」など、データベース全体の状態が意図通りであることも確認する必要があります。

D-3

ユニットテストコードの作成② ——MyBatisのリポジトリ

JPAのリポジトリのテストに続き、この節ではMyBatisのリポジトリ（`UserMapper`）のユニットテストコードを作成します。



サンプルアプリケーションの作成

サンプル ChD_Tesy/D-3

以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② ライブラリの追加
- ③ テストコードの作成
- ④ テストの実行

① ファイルなどの作成

図D-14の構成となるように、ファイルなどを作成してください。JPAのときと同様に、テスト用の設定ファイルなどは`src/test/resources`配下に配置します^[※2]。

[※2] MyBatisのリポジトリのテストでは、テスト用データベースにテーブルを作成するために`schema.sql`が必要です。`src/main/resources`にあるファイルを`src/test/resources`へコピーして作成してください。JPAではエンティティからテーブルが自動生成されましたが、MyBatisではこのSQLファイルを使ってテーブルを作成します。

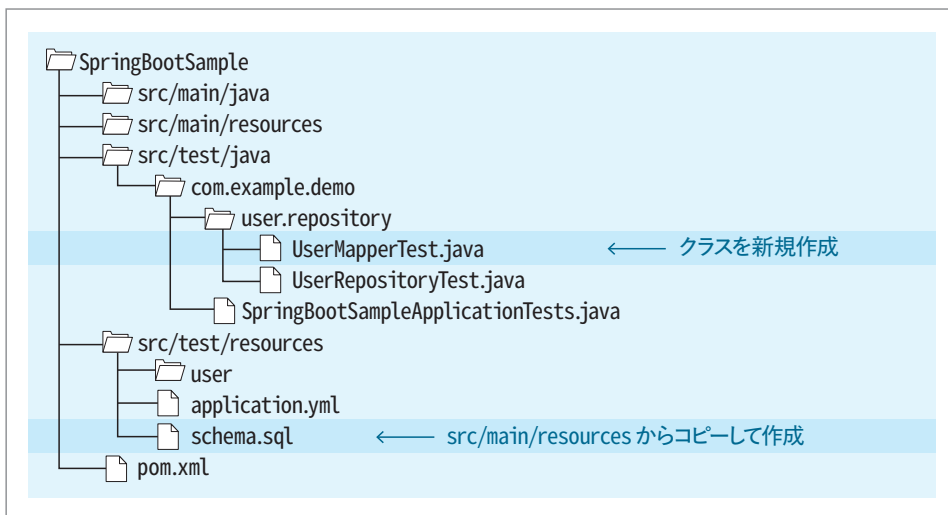


図 D-14 パッケージ・エクスプローラーの構成

② ライブラリの追加

MyBatis でユニットテストを行なうために、専用のライブラリをプロジェクトに追加します。pom.xml を開き、dependencies タグ内に [リスト D-6](#) のコードを追加してください。

リスト D-6 pom.xml

```
<dependencies>
... (省略) ...
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<!-- MybatisTest -->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter-test</artifactId>
  <version>3.0.3</version>
  <scope>test</scope>
</dependency>
</dependencies>
```

③ テストコードの作成

それでは、MyBatisのMapperのテストコードを作成します。UserMapperのfindOne()、findMany()、insertOne()をテストします。

JPAのテストコードを作成した際とほぼ同様の構成でテストコードを記述できます (リストD-7)。

リストD-7 UserMapperTest.java

```
package com.example.demo.user.repository;

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

import java.util.List;

import org.junit.jupiter.api.Test;
import org.mybatis.spring.boot.test.autoconfigure.MybatisTest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.test.context.jdbc.Sql;

import com.example.demo.user.domain.model.MUser;

@MybatisTest — ①
public class UserMapperTest {

    @Autowired
    private UserMapper mapper;

    /** findOneのテスト */
    @Test
    @Sql("/user/repository/data.sql")
    public void find_one(){
        // 実行
        MUser user = mapper.findOne("system@example.co.jp");
        // 検証
        assertThat(user.getId()).isEqualTo("system@example.co.jp");
        assertThat(user.getUserName()).isEqualTo("システム管理者");
        assertThat(user.getAge()).isEqualTo(24);
        assertThat(user.getRole()).isEqualTo("ROLE_ADMIN");
    }

    /** findManyのテスト */
    @Test
```



```

@Sql("/user/repository/data.sql")
public void find_many(){
    // 準備
    MUser user = new MUser();
    //Pageable
    Pageable pageable = PageRequest.of(0, 3); — B
    // 実行
    List<MUser> userList = mapper.findMany(user, pageable);
    // 検証
    assertThat(userList.size()).isEqualTo(3); — B
}

/** insertOneのテスト */
@Test
public void insertOne() {
    // 準備
    MUser newUser = new MUser();
    newUser.setUserId("new-user@test.co.jp");
    newUser.setUserName("new-user");
    newUser.setRole("ROLE-GENERAL");
    // 実行
    int count = mapper.insertOne(newUser);
    // 検証
    assertThat(count).isEqualTo(1);
    MUser user = mapper.findOne("new-user@test.co.jp");
    assertThat(user).isNotNull();
    assertThat(user.getUserId()).isEqualTo("new-user@test.co.jp");
    assertThat(user.getUserName()).isEqualTo("new-user");
    assertThat(user.getRole()).isEqualTo("ROLE-GENERAL");
}
}

```

ポイント ① @MybatisTest

@MybatisTest アノテーションをクラスに付けると、以下のことが可能になります。

- Spring Bootを起動した状態でテストが実行される

テスト実行時にSpring Bootが起動し、MapperのBean（インスタンス）が作成されるため、@Autowiredを使ってDI（注入）できるようになります。

- テストごとにトランザクションがロールバックされる

JPAのテストと同様に、テストメソッドごとにトランザクション（データベース操作のひとまとまり）が開始され、終了後に自動でロールバック（データの巻き戻し）が行なわれます。

このように@MybatisTestは、JPAのテストで使用した@DataJpaTestのMyBatis版だと考えるとわかりやすいでしょう。

なお、デフォルトではH2などのインメモリデータベースを使用しようとしています。実機のデータベースを使用する場合は、以下のように設定を上書きする必要があります。

H2データベース以外のデータベースを使用する場合

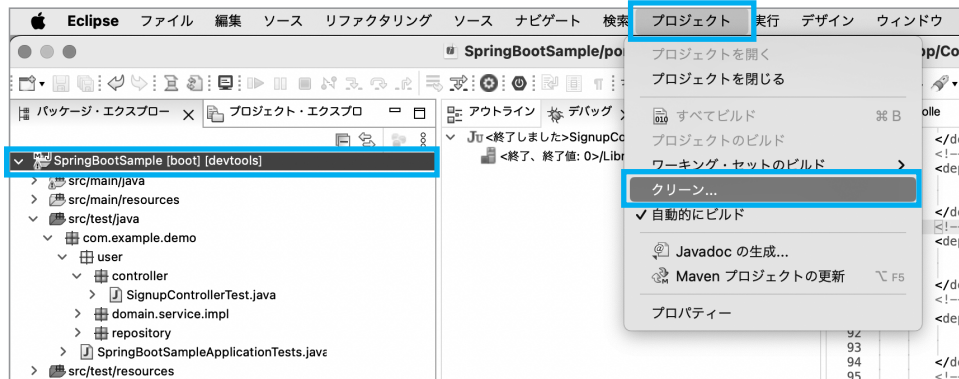
```
@MybatisTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
public class UserMapperTest {
    ... (省略) ...
}
```

Aでは、staticインポートを使用しています。これにより、Assertions.assertThat()をassertThat()だけで呼び出せるようになり、検証コードが読みやすくなります。

Bでは、Pageable（ページャーで使用するインターフェース）をUserMapperのfindMany()メソッドのパラメーターに渡しています。ページャーを使用しているため、1ページに表示する件数分しかデータを取得しません。検証フェーズでは、取得した件数が、あらかじめ設定した1ページあたりの最大表示件数（3件）と一致するかをチェックしています。

4 テストの実行

テストを実行する前に、プロジェクトの**クリーン**を実施しましょう。Eclipseのメニューから[プロジェクト] → [クリーン] を選択します（図D-15）。



図D-15 プロジェクトのクリーン

補足



プロジェクトのクリーン

Eclipseではプロジェクトを自動でビルド（ソースコードを実行可能な形式に変換）してくれます。しかし、ビルドを繰り返していると古いキャッシュが残ってしまい、最新の変更が正しく反映されないことがあります。

Eclipseで**クリーン**を実行することで、これら古い中間ファイルをいったん削除し、プロジェクトを最初からビルドし直すことができます。設定ファイルの変更（schema.sqlのコピーなど）を行なったあとは、クリーンを実行しておくで確実です。

それでは、作成したテストコードを実行してみましょう。EclipseでUserMapperTest.javaを右クリックし、[実行] → [JUnitテスト] を選択します（図D-16）。



図D-16 JUnit実行結果（図D-13再掲）

グリーンバー（テスト成功）が表示されました。これで、MyBatisを使用したリポジトリのテストも無事に成功しました。

D-4

サービスのユニットテスト

ここからはサービスのユニットテストコードを作成します。サービスでは、様々なクラスのメソッドを呼び出しています。そのため、モックを使用してサービスだけをユニットテストします。



サンプルアプリケーションの作成

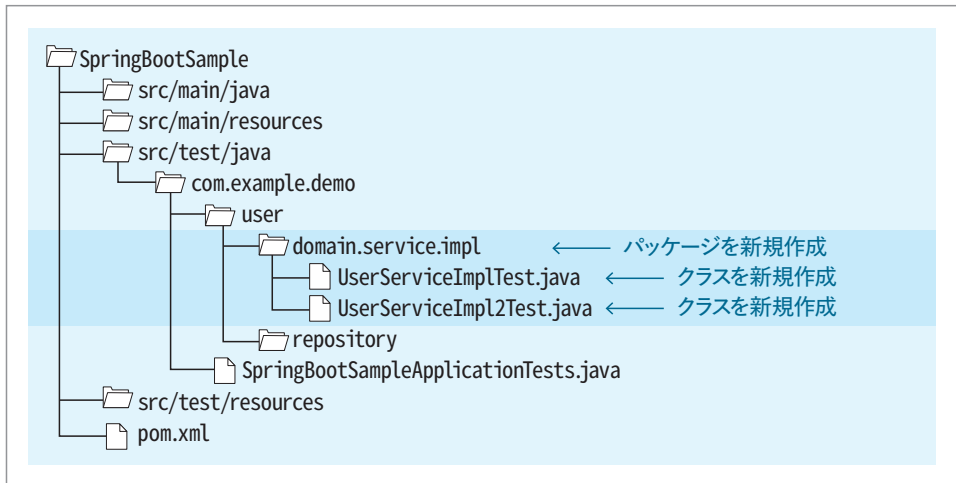
サンプル ChD_Tesy/D-4

以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② サービスのテストコードの作成
- ③ テストの実行

① ファイルなどの作成

図D-17の構成となるように、ファイルなどを作成してください。



図D-17 パッケージ・エクスプローラーの構成

② サービスのテストコードの作成

次に、サービスのテストを作成します。ここでは `UserServiceImpl` と `UserServiceImpl2` のテストコードを作成します。

◆ `UserServiceImpl` のテスト

まずは `UserServiceImpl` からテストコードを作成しましょう（リストD-8）。

[A] では、検証やモック作成に必要なメソッド（`assertThat()` や `when()` など）を `static` インポートしています。これにより、これらのメソッドをクラス名を付けずに直接呼び出すことができます。

リストD-8 `UserServiceImplTest.java`

```
package com.example.demo.user.domain.service.impl;

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

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.TestConfiguration;
```

[A]

```
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.context.annotation.Bean;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.example.demo.user.domain.model.MUser;
import com.example.demo.user.domain.service.UserService;
import com.example.demo.user.repository.UserMapper;

@ExtendWith(SpringExtension.class) —①
public class UserServiceImplTest {

    @Autowired
    private UserService service;

    @MockitoBean —②
    private UserMapper mapper;

    @TestConfiguration —③
    static class UserServiceImplTestConfig {
        // UserServiceImpl内で使用しているBeanを生成
        @Bean
        PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }

        // テスト対象クラスのBeanを生成
        @Bean
        UserService UserService(UserMapper mapper,
                                PasswordEncoder encoder) {
            return new UserServiceImpl(mapper, encoder);
        }
    }

    /** getUserOneメソッドのテスト */
    @Test
    public void get_user_one_test() {
        // 準備 =====
        MUser mockUser = new MUser();
        mockUser.setUserId("test@co.jp");
        mockUser.setUserName("getUserOneテストユーザー");
    }
}
```

```
// モック
when(mapper.findOne("test@co.jp")).thenReturn(mockUser); —②

// 実行 =====
MUser user = service.getUserOne("test@co.jp");

// 検証 =====
assertThat(user.getUserId()).isEqualTo("test@co.jp");
assertThat(user.getUserName()).isEqualTo("getUserOneテストユーザー");
    }
}
```

ポイント① @ExtendWith(SpringExtension.class)

@ExtendWithは、JUnitの拡張機能を使用するためのアノテーションです。このアノテーションの引数に、拡張機能を実装したクラスを指定します。

Spring Testでは、Springの機能（Beanの管理やDIなど）をテストで利用できるようにするための拡張機能として、**SpringExtension**クラスを提供しています。サービスなどのテストでSpring Testを使うには、**@ExtendWith(SpringExtension.class)**を付けると覚えておいてください。

なお、リポジトリのテストで使用した**@DataJpaTest**や**@MybatisTest**には、内部的に**@ExtendWith(SpringExtension.class)**の機能が含まれています。そのため、リポジトリのテストでは**@ExtendWith**の記述を省略できていたのです。

ポイント② モック (@MockitoBean) の使用

@MockitoBeanアノテーションをフィールドに付けると、そのフィールドは「モック」としてBeanに登録されます。モックとして登録されたクラスは、後述する**@TestConfiguration**で個別にBean登録する必要はありません。

また、モックを使用すると、「Xというメソッドを呼び出した場合、Yという値を返す」といった独自の振る舞いを設定できます。サンプルコードでは、以下の部分でモックの戻り値を設定しています。

モックの戻り値の設定

```
import static org.mockito.Mockito.*; ← staticインポート

... (省略) ...

when(mapper.findOne("test@co.jp")).thenReturn(mockUser);
```

`when()` メソッドの引数には、モックのメソッド呼び出しを指定します。それに続く `thenReturn()` メソッドの引数には、どのような値をモックから返すかを指定します。このサンプルコードでは、`UserMapper` の `findOne()` メソッドが呼び出された場合に、実際のデータベースにはアクセスせず、あらかじめ用意した `mockUser` 変数を戻り値として返すように設定しています。

モックの作成には、**Mockito** というフレームワークが使用されています。このように Mockito のクラスを `static` インポートすることで、`Mockito.when()` と書かずに、そのまま `when()` メソッドを呼び出すことができます。ユニットテストではモックの設定が多くなるため、可読性を高めるために `static` インポートを活用することをおすすめします。

なお、Mockito では「メソッドが呼び出された場合に例外を発生させる」などの設定もできます。詳しくは公式サイトなどを参照してください。

- **Mockito 公式サイト**

<https://site.mockito.org/>

ポイント ③ @TestConfiguration

`@ExtendWith(SpringExtension.class)` をクラスに付けても、自動的に Bean が登録されるわけではありません。ユニットテストではコンポーネントスキャン（Bean を自動で探して登録する仕組み）が行なわれない構成にすることが多いです（理由は後述します）。そのため、必要なクラスを個別に Bean 登録する必要があります。このサンプルコードでは、テスト対象である `UserServiceImpl` を Bean 登録しなければなりません。

そこで使用するのが **@TestConfiguration** アノテーションです。このアノテーションを付けたクラスを用意すると、「テストのときだけ有効な JavaConfig（設定クラス）」を作成できます。本書のサンプルでは、内部クラス（`UserServiceImplTestConfig`）として定義しています（図 D-18）。

なお、テスト対象の `UserServiceImpl` は、内部で `PasswordEncoder` と `UserMapper` の Bean を使用しています。そのため、これら 2 つも Bean 登録しておかなければ、`UserServiceImpl` の Bean を作成（インスタンス化）できません。今回のコードでは、`UserMapper` は `@MockitoBean` を使って登録しています。そのため、内部クラスの `UserServiceImplTestConfig` では、残りの `PasswordEncoder` だけを Bean 登録しています。

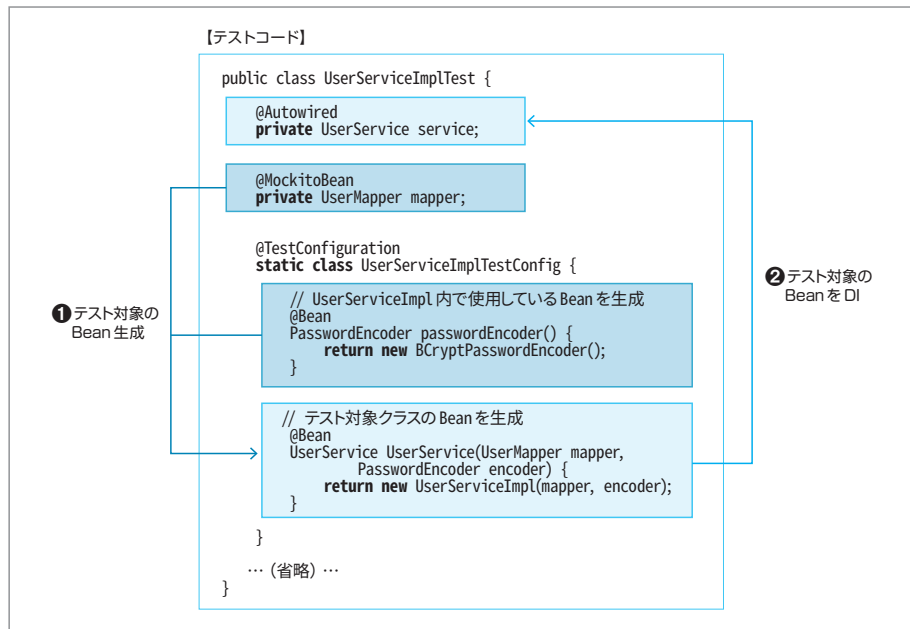


図 D-18 TestConfiguration

ユニットテストでは、「1つのクラス」だけをテスト対象とします。そのため、テスト対象のクラスと、その依存関係のBeanだけを用意すればテストを実行できます。しかし、Spring Bootを起動すると、コンポーネントスキャンによりアプリケーション全体でBeanが作成されます。アプリケーションの規模が大きくなればなるほど、この処理に時間がかかります。

ユニットテストのコードはクラスごとに作成し、開発中は「テストコード自体の不具合を確認する」「修正した箇所だけ動かす」といった理由で、特定のテストを何度も繰り返し実行します。それが、コンポーネントスキャンの影響で時間がかかっていては、非効率です。そのため、@TestConfigurationを使用して、関係ないBeanを作成しないことで、テストの実行時間を減らしているのです。

補足



@DataJpaTestと@MybatisTest

リポジトリのテストで使用していた@DataJpaTestと@MybatisTestアノテーションでは、Spring Bootのテスト用コンテキスト（テストのために必要なBeanだけを読み込んだ実行環境）が起動してからテストコードが実行されます（図 D-19）。コンソールにSpringと大きな文字（アスキーアート）が表示されていれば、Spring Bootが起動しており、必要な機能に限定して動作している状態です。


```

import org.springframework.test.context.junit.jupiter.SpringExtension;

import com.example.demo.user.domain.model.MUser;
import com.example.demo.user.domain.service.UserService;
import com.example.demo.user.repository.UserRepository;

@ExtendWith(SpringExtension.class)
public class UserServiceImpl2Test {

    @Autowired
    private UserService service;

    @MockitoBean
    private UserRepository repository;

    @TestConfiguration
    static class UserServiceImpl2TestConfig {
        // UserServiceImpl2内で使用しているBeanを生成
        @Bean
        PasswordEncoder passwordEncoder() {
            return new BCryptPasswordEncoder();
        }

        // テスト対象クラスのBeanを生成
        @Bean
        UserService userService(UserRepository repository,
                                PasswordEncoder encoder) {
            return new UserServiceImpl2(repository, encoder);
        }
    }

    /** getUserOneメソッドのテスト */
    @Test
    public void get_user_one_test() {
        // 準備 =====
        MUser mockUser = new MUser();
        mockUser.setUserId("test@co.jp");
        mockUser.setUserName("getUserOneテストユーザー");
        Optional<MUser> returnValue = Optional.of(mockUser);
        // モック
        when(repository.findById("test@co.jp")).thenReturn(returnValue);

        // 実行 =====
        MUser user = service.getUserOne("test@co.jp");
    }
}

```

```
// 検証 =====  
assertThat(user.getId()).isEqualTo("test@co.jp");  
assertThat(user.getUserName()).isEqualTo("getUserOneテストユーザー");  
}  
}
```

③ テストの実行

テストコードを作成したら、実際にテストを実行してみましょう。今回は、複数のテストクラスをパッケージ単位でまとめて実行します。

Eclipseでcom.example.demo.user.domain.service.implパッケージを右クリックし、[実行] → [JUnitテスト] を選択します (図D-20)。

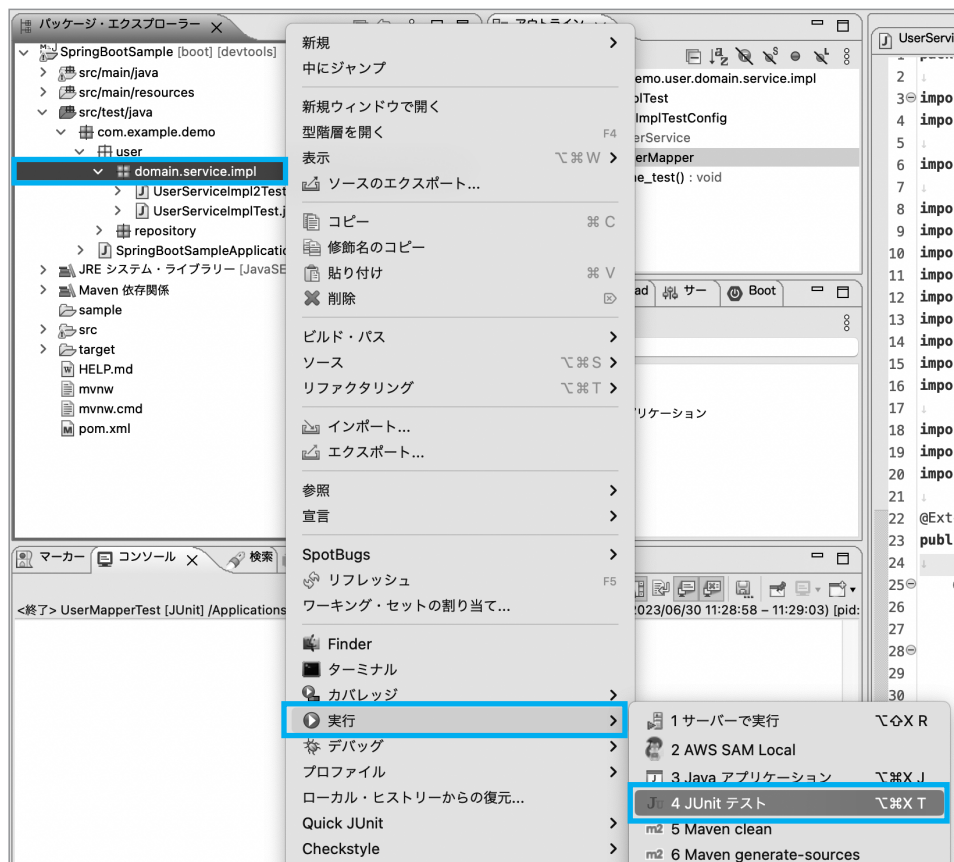


図 D-20 テストコードをまとめて実行

選択したパッケージ配下に含まれているすべてのテストクラスが実行されます。「JUnit」タブには、クラスごとにテスト結果が出力されます（図 D-21）。

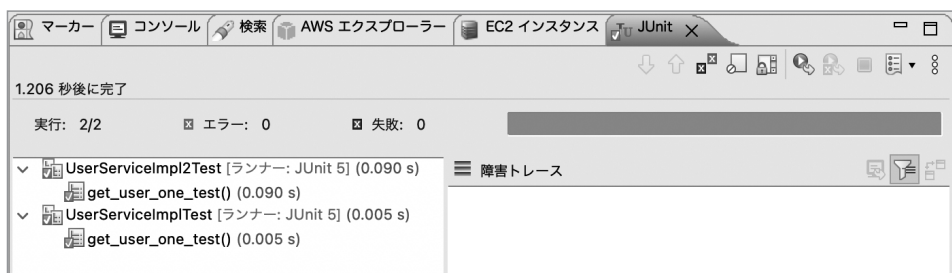


図 D-21 JUnit実行結果

グリーンバー（テスト成功）が表示されました。

サービスのユニットテストでは、アプリケーション全体を起動せず、テスト対象のクラスとその依存関係のみを個別に Bean として登録して実行しています。そのため、リポジトリのテスト時と比較して、テストの実行時間が短くなることが多く、実際にその違いを確認できます。

D-5

コントローラーのユニットテスト (ログインなし)

ここからは、アプリケーションの入り口となるコントローラーのユニットテストを作成します。まずは、ログインが必要ない画面（ユーザー登録画面など）についてのテストコードを作成します。



サンプルアプリケーションの作成

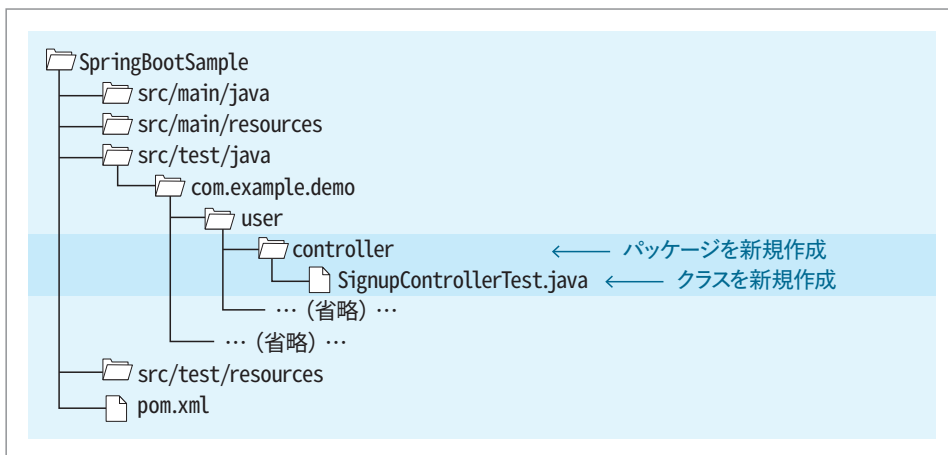
サンプル ChD_Test/D-5

以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② コントローラーのテストコードの作成
- ③ テストの実行

① ファイルなどの作成

図 D-22 のディレクトリ構成となるように、ファイルなどを作成してください。



図D-22 パッケージ・エクスプローラーの構成

② コントローラーのテストコードの作成

ここからは、SignupControllerのテストコードを作成します（リストD-10）。

A では、HTTP リクエストの送信やレスポンスの検証で使用するメソッドを、クラス名を省略して呼び出せるようにするためにstatic インポートを行なっています。

リストD-10 SignupControllerTest.java

```
package com.example.demo.user.controller;

import static org.hamcrest.CoreMatchers.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.test.web.servlet.MockMvc;

import com.example.demo.config.SecurityConfig;
import com.example.demo.user.application.UserService;
```

```

import com.example.demo.user.domain.service.UserService;

@WebMvcTest(SignupController.class) ❶
@Import({SecurityConfig.class}) ❷
public class SignupControllerTest {

    @Autowired
    private MockMvc mockMvc; ❸

    @MockitoBean
    private UserService service;

    @TestConfiguration
    static class SignupControllerTestConfig {
        // ModelMapperのBean登録
        @Bean
        ModelMapper modelMapper() {
            return new ModelMapper();
        }

        // UserApplicationServiceのBean登録
        @Bean
        UserApplicationService userApplicationService(MessageSource messageSource) {
            return new UserApplicationService(messageSource);
        }
    }

    @Test
    @DisplayName("画面表示のテスト")
    public void getSignup() throws Exception {
        this.mockMvc.perform(get("/user/signup"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("ユーザー登録")))
            .andExpect(content().string(containsString("男性"))); ❸
    }
}

```

ポイント ❶ @WebMvcTest

@WebMvcTest アノテーションを使用すると、コントローラーに関連するクラス（@Controller や @ControllerAdvice など）のみが Bean として登録されます。アプリケーション全体を起動せず、Web 層（画面処理）に関係する部品のみを対象にテストするための便利なアノテーションです。

デフォルトではすべてのコントローラーがスキャン対象になるため、他のコントローラーが必要とするBeanが足りずにエラーになる場合があります。そのため、テスト対象とするコントローラー（`SignupController.class`）を`@WebMvcTest`の引数に明示的に指定します。これにより、不要なBeanの読み込みを避け、テストを軽量・高速に実行できます。

なお、コントローラーが依存しているサービスなどは自動登録されません。そのため、今回のサンプルでは、`UserService`を`@MockitoBean`で登録し、その他の必要な部品を`@TestConfiguration`を使って個別にBean登録しています。

ポイント② @Import

`@WebMvcTest`を使用すると、通常のSpring Boot起動時とは異なり、セキュリティ設定（Spring Securityの設定）が自動では読み込まれないことがあります。そのため、セキュリティ設定が必要な場合は、**@Import アノテーション**を使用して明示的に設定クラス（`SecurityConfig.class`）を読み込みます。

`@Import`を使用すると、`@Configuration`が付いたクラスの設定を適用できます。

ポイント③ MockMvc

MockMvcを使うことで、実際にサーバーを起動せずに、コントローラーへのHTTPリクエスト送信とレスポンスの検証をシミュレートできます。本書のサンプルコードでは、以下のように使用しています。

MockMvcの使用箇所

```
import static org.hamcrest.CoreMatchers.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

... (省略) ...

this.mockMvc.perform(get("/user/signup")) ← /user/signupにGETリクエストを送信
    .andExpect(status().isOk()) ← レスポンスのステータスが200（成功）であること⇒
    をチェック
    .andExpect(content().string(containsString("ユーザー登録"))); ← 画面表示内容⇒
    に"ユーザー登録"という文字が含まれていることをチェック
```

`perform()` メソッドでリクエストを実行し、`andExpect()` メソッドで期待される結果を検証します。`get()` や `status()` などのメソッドは、冒頭で `static` インポートしているため、このように簡潔に記述できます。

MockMvcでは、ステータスコードやレスポンス内容の検証だけでなく、リダイレクトやヘッダーの確認、リクエストパラメーターの送信など、様々なテストが可能です。詳細はJavadocなどを参照してください。

- [MockMvcのJavadoc](#)

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html>

補足

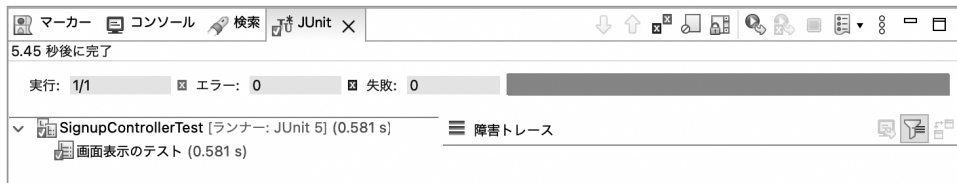


@DisplayName

これまで、テストの実行結果にはメソッド名が表示されていましたが、@DisplayNameアノテーションを使用すると、テストの目的や内容を説明した名前（日本語など）をテスト名として表示できます。たとえば「ユーザー登録画面が正しく表示されること」など、テスト内容を表示することで、どのようなテストを行なうか、どのテストが成功／失敗したかをすぐに判別できるようになります。

③ テストの実行

テストコードを作成したら、実際にテストを実行してみましょう。SignupControllerTest.javaを右クリックし、[実行] → [JUnitテスト] を選択します (図D-23)。



図D-23 JUnit実行結果

無事にグリーンバーが表示されれば成功です。

もしエラーが発生する場合は、プロジェクトを「クリーン」してみてください。プロジェクトを選択した状態で、Eclipseのメニューから【プロジェクト】→【クリーン】を選択します。

クリーンを実行すると古いビルド情報が破棄され、再ビルド（プログラムの再構築）が行われます。そのあとで再度テストを実行すると、正常に動作する場合があります。

D-6

コントローラーのユニットテスト
(ログインあり)

次に、ログインが必要な画面についてのテストコードを作成します。



サンプルアプリケーションの作成

サンプル ChD_Test/D-6

以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② ライブラリの追加
- ③ カスタムユーザーのモックの作成
- ④ コントローラーのテストコードの作成
- ⑤ テストの実行

① ファイルなどの作成

図 D-24 の構成となるように、ファイルなどを作成してください。

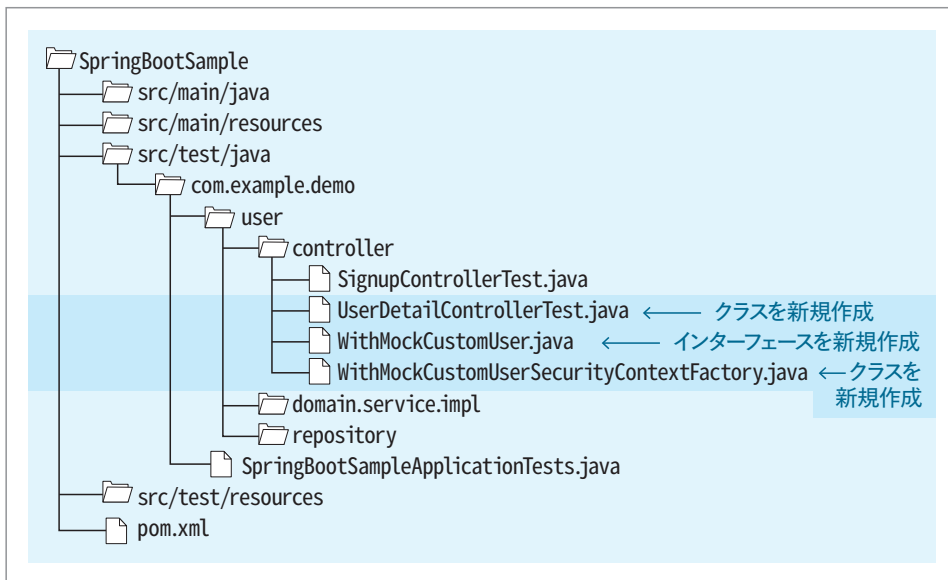


図 D-24 パッケージ・エクスプローラーの構成

② ライブラリの追加

Spring Securityを使用してログイン機能を作成した場合、Spring Securityのテスト用ライブラリをpom.xmlに追加する必要があります（リストD-11）。

リストD-11 pom.xml

```
<dependencies>
  ... (省略) ...
  <!-- MybatisTest -->
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter-test</artifactId>
    <version>3.0.3</version>
    <scope>test</scope>
  </dependency>
  <!-- spring-security-test -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
  </dependency>
  ... (省略) ...
</dependencies>
```

③ カスタムユーザーのモックの作成

Spring Securityが用意するユーザーを使用する場合、通常は@WithMockUserというアノテーションを使います。しかし、本書ではユーザー名を画面に表示するために独自のユーザークラス（LoginUser）を作成しました。

テスト環境でも、この独自ユーザーがログインしている状態を再現するために、カスタムユーザーのモックを作成します。

◆ カスタムユーザー用アノテーションの作成

まずは@WithMockUserの代わりに、テストメソッドに付与するためのアノテーション@WithMockCustomUserを作成します（リストD-12）。

リストD-12 WithMockCustomUser.java

```
package com.example.demo.user.controller;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import org.springframework.security.test.context.support.WithSecurityContext;

@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = WithMockCustomUserSecurityContextFactory.class) — ①
public @interface WithMockCustomUser { — A
    String username() default "user";
    String password() default "password";
    String[] roles() default {"USER"};
    String displayName() default "userName"; — B
}
```

ポイント ① @WithSecurityContext

@WithSecurityContext は、テスト用の **SecurityContext** を作成するためのアノテーションです。**SecurityContext** は、Spring Security が現在ログイン中のユーザー情報（認証情報）を保持するための「入れ物（コンテキスト）」です。

引数の **factory** には、この入れ物に格納する認証情報（ユーザー情報）を組み立てるための **ファクトリークラス**（オブジェクトを生成する役割を持つクラス）を指定します。

アノテーションを定義するため、**A** のように **public @interface** と記述します。

B は、アノテーションの引数として渡せるフィールドです。本書の **LoginUser** クラスは、標準の **User** クラスを継承して **displayName** を追加しているため、アノテーション側でもその値を指定できるようにフィールドを用意しています。

◆ SecurityContext を生成するファクトリークラスの作成

次に、アノテーションの設定値を読み取って、実際に **SecurityContext** を組み立てるファクトリークラスを作成します（[リスト D-13](#)）。

リスト D-13 WithMockCustomUserSecurityContextFactory.java

```
package com.example.demo.user.controller;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.context.SecurityContext;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.test.context.support.WithSecurityContextFactory;

import com.example.demo.user.domain.model.LoginUser;
```

```

public class WithMockCustomUserSecurityContextFactory implements WithSecurityContextFactory➡
<WithMockCustomUser>{ ①

    @Override
    public SecurityContext createSecurityContext(WithMockCustomUser customUser) {
        // LoginUserを生成
        LoginUser principal = new LoginUser(
            customUser.username(),
            customUser.password(),
            AuthorityUtils.createAuthorityList(customUser.roles()),
            customUser.displayName()
        );

        // Authentication生成
        Authentication auth = new UsernamePasswordAuthenticationToken(principal,
            principal.getPassword(),
            principal.getAuthorities());

        // SecurityContextにAuthenticationをセット
        SecurityContext context = SecurityContextHolder.createEmptyContext();
        context.setAuthentication(auth);
        return context;
    }
}

```

ポイント① WithSecurityContextFactory

テスト用のSecurityContextを独自に生成する場合、**WithSecurityContextFactory** インターフェースを実装します。このインターフェースでは、どのアノテーションの情報をを使ってSecurityContextを作るのかをジェネリクス（< >の部分）で指定します。本書のサンプルでは、**WithMockCustomUser** を指定しています。

実装が必要なcreateSecurityContext()メソッドは、テスト実行時にSpring Securityによって自動的に呼び出されます。引数には、テストメソッドに付けた@WithMockCustomUser アノテーションの設定内容（usernameやpasswordなど）が渡されます。つまり、アノテーションに指定した値を、このメソッド内で取得して、ログインユーザー情報の生成に利用できるということです。

@WithMockCustomUserの使用例

```
@WithMockCustomUser(username="hoge", password="fuga", ...)
```

上記の例の場合、usernameにhoge、passwordにfugaという値を持ったWithMockCustomUserが引数として渡されます。

そして、createSecurityContext() メソッドの中で、これらの値を使って LoginUser、Authentication、SecurityContext を順に生成し、ログイン状態を組み立てていきます。

④ コントローラーのテストコードの作成

次に、ユーザー詳細画面のコントローラー (UserDetailController) のテストコードを作成します (リストD-14)。

リストD-14 UserDetailControllerTest.java

```
package com.example.demo.user.controller;

import static org.hamcrest.CoreMatchers.*;
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.test.context.bean.override.mockito.MockitoBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.test.web.servlet.MockMvc;

import com.example.demo.config.SecurityConfig;
import com.example.demo.user.domain.model.MUser;
import com.example.demo.user.domain.service.UserService;

@WebMvcTest(UserDetailController.class)
@Import({SecurityConfig.class})
public class UserDetailControllerTest {

    @TestConfiguration
    static class UserDetailControllerTestConfig {
        @Bean
        ModelMapper modelMapper() {
            return new ModelMapper();
        }
    }
}
```

```

@Autowired
private MockMvc mockMvc;

@MockitoBean
private UserService service;

@Test
@WithMockCustomUser(username="admin@example.co.jp",
    roles={"ROLE_ADMIN"},
    displayName="adminユーザー")
@DisplayName("画面表示のテスト (ADMINユーザー)")
public void getUserAdmin() throws Exception {
    // 準備
    MUser mockUser = new MUser();
    mockUser.setUserId("admin@example.co.jp");
    mockUser.setUserName("adminユーザー");
    mockUser.setRole("ROLE_ADMIN");

    // モック
    when(service.getUserOne("admin@example.co.jp")).thenReturn(mockUser);

    // 実行
    this.mockMvc.perform(get("/user/detail/admin@example.co.jp"))
        .andExpect(status().isOk())
        .andExpect(content().string(containsString("ユーザー詳細")))
        .andExpect(content().string(containsString("adminユーザー")))
        .andExpect(content().string(containsString("アドミン専用")));
}

@Test
@WithMockCustomUser(username="general@example.co.jp",
    roles={"ROLE_GENERAL"},
    displayName="generalユーザー")
@DisplayName("画面表示のテスト (GENERALユーザー)")
public void getUserGeneral() throws Exception {
    // 準備
    MUser mockUser = new MUser();
    mockUser.setUserId("general@example.co.jp");
    mockUser.setUserName("generalユーザー");
    mockUser.setRole("ROLE_GENERAL");

    // モック
    when(service.getUserOne("general@example.co.jp")).thenReturn(mockUser);

```

B

C

B

// 実行

```
this.mockMvc.perform(get("/user/detail/general@example.co.jp"))
    .andExpect(status().isOk())
    .andExpect(content().string(containsString("ユーザー詳細")))
    .andExpect(content().string(containsString("generalユーザー")))
    .andExpect(content().string(not(containsString("アドミン専用"))));
}
}
```

C

A は、static インポートを使用している箇所（assertThat() や when()、get() などクラス名なしで呼び出せるようにしている部分）です。

B は、ユーザー詳細画面はログイン後にしか表示できない画面であるため、ログインしていない状態でアクセスするとエラーになります。そこで、先ほど作成した @WithMockCustomUser アノテーションを使用して、テスト用に「ログインしている状態」を再現しています。これにより、どのようなユーザー（ユーザー名）やロール（権限）でログインしているかをテストごとに細かく設定できます。

C では、アドミン権限（ROLE_ADMIN）と一般権限（ROLE_GENERAL）のユーザーで、それぞれユーザー詳細画面を表示するテストを行なっています。

アドミン権限の場合は、アドミンユーザーだけがアクセスできる機能へのリンクとして、"アドミン専用" という文字列が表示されていることを確認しています。

アドミン権限ユーザーのテストコード（getUserAdmin() メソッド）

```
this.mockMvc.perform(get("/user/detail/admin@example.co.jp"))
    ... (省略) ...
    .andExpect(content().string(containsString("アドミン専用"))); ← 表示されるかチェック
```

一方、一般権限ユーザーでは、"アドミン専用" という文字列が表示されないことを確認しています。

一般権限ユーザーのテストコード（getUserGeneral() メソッド）

```
this.mockMvc.perform(get("/user/detail/general@example.co.jp"))
    ... (省略) ...
    .andExpect(content().string(not(containsString("アドミン専用"))));
```

not() メソッドを使用することで、指定した文字列が含まれていないこと（=表示されていないこと）を検証できます。

⑤ テストの実行

テストコードを作成したら、実際にテストを実行してみましょう。Eclipse で com.example.demo.user.controller パッケージを右クリックし、[実行] → [JUnit テスト] を選択します (図 D-25)。

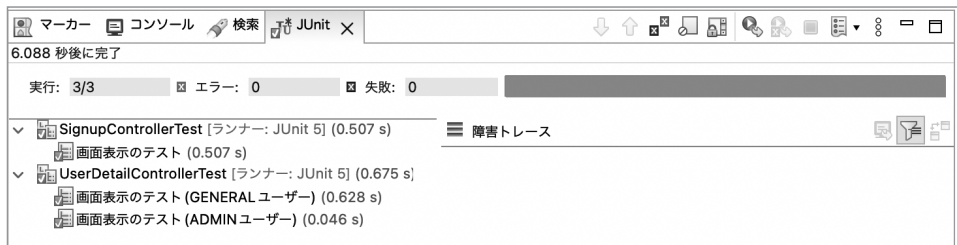


図 D-25 JUnit実行結果

実行すると、@DisplayNameで設定したテスト内容が「JUnit」タブに表示されます。すべてのテストが成功すると、グリーンバー（テスト成功）が表示されます。これでユニットテストコードの作成は完了です。

補足



カバレッジ

カバレッジとは、ソースコード全体のうち、テストコードによって実行された割合（％）を表わす指標です。カバレッジを確認することで、if文などの分岐処理が実際にテストされているかどうかを把握できます。つまり、「どの部分がテストされていないか」を確認するための目安になります。

ただし、カバレッジは必ず100％にしなければならないものではありません。たとえば、意図的に発生させることが難しい例外処理など、テストが現実的でないコードも存在します。そのような場合は、一部のcatch句をカバレッジの対象外にするなどの対応を行ないます。

カバレッジを計測するツールとしては、EclEmmaやJaCoCoなどがあります。これらのツールを導入すると、Eclipseなどでテストを実行する際に、どのコードが実行されたかを可視化しながら確認できます。

D-7

インテグレーションテストの テストコード作成

最後に、インテグレーションテストのテストコードを作成します。

インテグレーションテストとは、複数のクラスや機能を組み合わせて動作を確認するテストで、単体テストから結合テストに相当する範囲を対象としています。

ユニットテストでは各クラス単体の動作を確認しましたが、インテグレーションテストでは、それらを組み合わせたときに正しく動くかを確認します。そのため、基本的にモックは使用せず、実際のBeanを使ってテストを行ないます。



以下の手順でサンプルアプリケーションを作成します。

- ① ファイルなどの作成
- ② テストコードの作成
- ③ テストの実行

① ファイルなどの作成

図 D-26 の構成となるように、ファイルなどを作成してください。

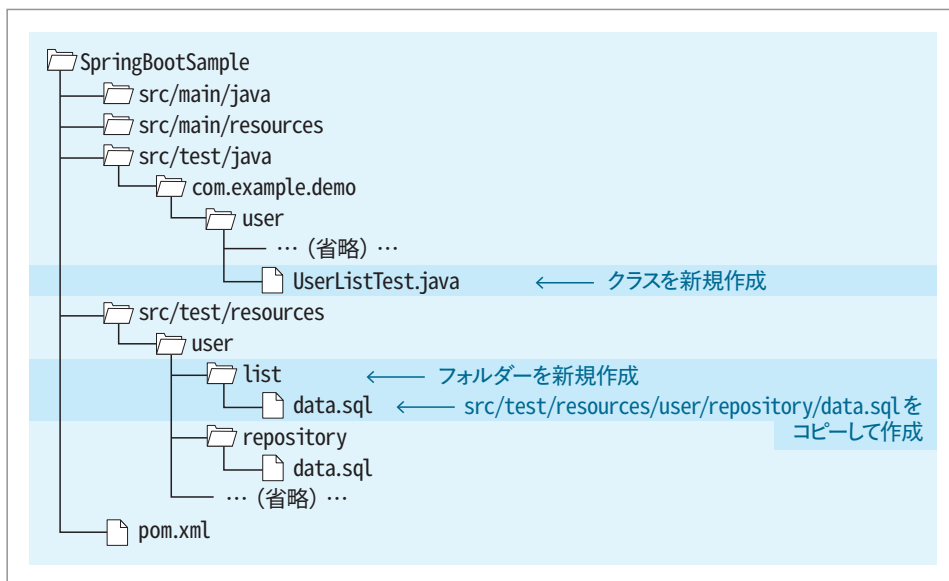


図 D-26 パッケージ・エクスプローラーの構成

data.sql の中身は src/test/resources/user/repository/data.sql と同一です。

② テストコードの作成

次はユーザー一覧画面のテストコードを作成します (リスト D-15)。

リスト D-15 UserListTest

```
package com.example.demo.user;
```

```

import static org.hamcrest.CoreMatchers.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.transaction.annotation.Transactional;

import com.example.demo.user.controller.WithMockCustomUser;

@SpringBootTest
@AutoConfigureMockMvc
@Transactional
public class UserListTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    @WithMockCustomUser(username="admin", roles={"ADMIN"}, displayUserName="アドミン")
    @DisplayName("画面表示のテスト")
    @Sql("/user/list/data.sql")
    public void getRequest() throws Exception {
        this.mockMvc.perform(get("/user/list"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("ユーザー一覧画面")))
            .andExpect(content().string(containsString("システム管理者")));
    }

    @Test
    @WithMockCustomUser(username="admin", roles={"ADMIN"}, displayUserName="アドミン")
    @DisplayName("検索のテスト")
    @Sql("/user/list/data.sql")
    public void search() throws Exception {
        this.mockMvc.perform(get("/user/list")
            .param("userId", "user8"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("ユーザー8")));
    }
}

```

ポイント① @SpringBootTest

@SpringBootTest アノテーションを付けると、Spring Boot アプリケーションを**通常と同じように**起動してからテストが実行されます。ここでいう「通常と同じように」とは、アプリケーションで使用しているすべてのBeanがIoCコンテナに登録された状態のことです。

ユニットテストでは必要なBeanだけを個別に用意しましたが、インテグレーションテストではアプリケーション全体の構成をそのまま使って動作を確認します。

また、以下のように設定を書くことで、テスト時の設定値を変更することもできます

設定値を変更する場合

```
@SpringBootTest(properties = {spring.profiles.active=local})
public class XXXTest {
    ... (省略) ...
}
```

テストコードを作成する際に大きな負担となるのが、Beanの登録です。IoCコンテナに登録しなければならないBeanが多かったり、インスタンスの生成方法が複雑で準備が大変だったりする場合には、**@SpringBootTest**を使用します。

ポイント② @AutoConfigureMockMvc

@SpringBootTest アノテーションだけでは、MockMvcは自動で利用できません。MockMvcを使うためには、**@AutoConfigureMockMvc** アノテーションを付けて設定を有効にする必要があります。これにより、実際にサーバーを起動せずに、HTTPリクエストのテストを行なえるようになります。

補足

**@WebMvcTest との違い**

ユニットテストで用いた**@WebMvcTest** アノテーションには、**@AutoConfigureMockMvc** アノテーションの設定があらかじめ含まれています。そのため、コントローラーのユニットテストでは、追加で**@AutoConfigureMockMvc** アノテーションを記述する必要はありませんでした。

一方、**@SpringBootTest** アノテーションはアプリケーション全体を対象とするため、MockMvcなどの機能は個別に有効化する必要があります。

A は、これまでのテストと同様にstaticインポートを使用している箇所です。これにより、**get()** や **status()** などのメソッドをクラス名なしで簡潔に呼び出せるようになります。

B のように**@Transactional** アノテーションを付けると、テスト終了時にデータベースの変

更内容がロールバック（元に戻る）されます。このサンプルでは、@Sql アノテーションでテスト用データを登録していますが、テスト終了後に自動で元の状態に戻るため、テスト同士の影響を防ぐことができます。

③ テストの実行

テストコードを作成したら、実際にテストを実行してみましょう。UserListTest.java を右クリックし、[実行] → [JUnit テスト] を選択します。



図 D-27 JUnit 実行結果

これでインテグレーションテストの作成は終了です。

補足



Selenium

インテグレーションテストやシナリオテストなどで、[Selenium](#) というフレームワークを利用することがあります。Selenium を使うと、ブラウザを開いて画面を操作したり、スクリーンショットを撮ったりすることができます。実際に画面で確認しないとわかりづらいテスト結果も、Selenium を使うことで視覚的に確認しやすくなります。ただし、利用するには Selenium の基本的な使い方を学ぶ必要があります。

補足



Cucumber

シナリオテストでは、[Cucumber](#) というテストツールを用いることもあります。Cucumber は、開発者・テスト担当者・ビジネス担当者が同じ基盤でシステムの振る舞いを定義し、テストできるツールです。自然言語に近い記述ができるため、技術者でない人にも理解しやすい仕様書を作成しつつ、それを元にした自動テストも実行できます。

D-8 まとめ

この章で学んだ内容は以下の通りです。

→ 概要

- ✓ 開発工程と、それに対応するテスト工程を対にして整理した考え方として **V字モデル** がある。要件定義の内容を受け入れテスト、基本設計の内容をシステムテスト、詳細設計の内容を結合テスト、開発の内容を単体テストで確認するというのがV字モデルの考え方。
- ✓ テストコードを用意することで、テストを自動化できる。
- ✓ テストコードを作成すると、全体の開発量が2倍以上になるというデメリットもある。また、ソースコードだけでなくテストコードの保守も必要になる。
- ✓ ユニットテストでは、Javaのクラス単位でテストを行なう。
- ✓ クラスから別のクラスのメソッドを呼び出すことは多いが、呼び出し先に不具合があるとテストが正しく行なえない。そのような場合にモックを使用する。モックとは、ダミーの値を返すクラスのこと。
- ✓ Spring Testとは、Springでのテストコードを簡単に作成するためのフレームワーク。

→ ユニットテストコードの作成

- ✓ `import` 文に `static` を付けることで、`static` インポートが使用できる。`static` インポートとは、クラス名を省略して `static` 変数や `static` メソッドを呼び出せるようにするJavaの構文のこと。これにより、テストコードの記述を簡潔にできる。
- ✓ リポジトリにJPAを使用している場合、テストクラスに `@DataJpaTest` アノテーションを使用することで、以下が可能になる。
 - ✓ Spring Bootを起動してからテストコードが実行される
 - ✓ `TestEntityManager` クラスを使用できる
 - ✓ テスト単位でロールバックされる
- ✓ `@Test` アノテーションが付いているメソッドのみがテストコードとして実行される。`@Test` はJUnitが提供するアノテーション。
- ✓ `@Sql` アノテーションを使用すると、指定したSQLファイルを実行したあとにテストを実行できる。SQLファイルは `src/test/resources` からの相対パスで指定する。
- ✓ `TestEntityManager` は、`EntityManager` よりも簡単にデータ操作ができるクラスであり、`@DataJpaTest` アノテーション使用時に利用できる。
- ✓ `@DataJpaTest` アノテーションのMyBatis版が `@MybatisTest` アノテーションであり、Mapperのテストに使用する。
- ✓ `@ExtendWith` は、JUnit5の拡張機能を利用するためのアノテーションであり、Spring Testでは重要な役割を持つ。
- ✓ `@TestConfiguration` アノテーションを付けたクラスを用意すると、テスト時のみ有効な `JavaConfig` を定義できる。
- ✓ `@MockitoBean` アノテーションをフィールドに付けると、そのBeanをモックとして登録できる。

- ✓ `@WebMvcTest` アノテーションを使用すると、`@Controller` アノテーションや `@ControllerAdvice` アノテーションなど、コントローラー関連の `Bean` のみが登録される。
- ✓ `@Import` アノテーションを使用すると、`@Configuration` アノテーションが付いたクラスの設定を適用できる。
- ✓ `MockMvc` を使用することで、コントローラーにリクエストを送信し、レスポンスを簡単に検証できる。
- ✓ `@WithMockUser` アノテーションを使用すると、テスト時に任意のユーザーやロール（権限）でログイン状態を再現できる。

→ インテグレーションテストコードの作成

- ✓ `@SpringBootTest` アノテーションを付けると、Spring Boot アプリケーションを通常通り起動した状態（すべての `Bean` が `IoC` コンテナに登録された状態）でテストを実行できる。
- ✓ `@AutoConfigureMockMvc` アノテーションを付けることで、`MockMvc` を利用できるようになる。