実務レベルのシステム作成に必要な知識を解説

超入門テキスト

Spring Boot3で始める Webアプリケーション開発入門

菊田 英明
KIKUTA HIDEAKI

入門を超えて実践へ!

国際化対応、関連エンティティ
ファイルアップロード/ダウンロード
PDF/Excel出力、Login認証

応用編

はじめに

本書は「Spring Boot3で始めるWebアプリケーション開発入門(基礎編)(以下、前書)」の続編です。前書の内容をベースに、Spring Bootを使って**実務レベルのシステム作成に必要な知識**を解説していきます。

想定している読者は、

・ITシステム開発企業の新入社員

・転職してWebプログラマーになりたい人

・自分でも何かWebアプリケーションを作ってみたい人

といった方々です。

Spring Boot 3

Spring Boot 3(https://spring.io/projects/spring-boot)は**Spring Framework**というフレームワーク群(https://spring.io/projects)をシンプルに活用しやすくしたものです。Spring Frameworkを直接使うより少ない手間でWebアプリケーションを構築できます。

このSpring Boot 3はSpring Bootの3番目のメジャーバージョンであり、2022年11月にリリースされました。それ以前のメジャーバージョンであるSpring Boot 2から一部のパッケージ名が変更されるなど、ソースコード的に非互換な部分があります。

本書はこの最新のSpring Boot 3を使って、より実務に近いWebアプリケーションを作っていきます。

なお本書ではSpring Boot3をSpring Bootと表記します。

本書の構成

前書で作成したToDoアプリケーション(以下、ToDoアプリ)に様々な機能を追加しながら、Spring Bootの使い方を解説していきます。このため本書は12章から始まります。

本書の構成は下表のようになっています。

章	タイトル	備考
	はじめに	本章
12章	国際化対応	Spring Bootの基礎知識
13章	操作メッセージ	〃

14章	エンティティの関連	〃
15章	関連するエンティティの更新	〃
16章	関連するエンティティの削除	〃
17章	関連するエンティティの追加	〃
18章	ファイルのアップロード/ダウンロード	各種APIの使いこなし
19章	PDF出力	〃
20章	Excel出力	〃
21章	ログイン認証	認証と認可
22章	アクセス制御とエラー処理	〃
補足	前書・演習課題の実装例	
–	参考情報	

前半は「Spring Bootの基礎知識」ということで、前書で解説できなかった機能を扱います。このうち「15章 関連するエンティティの更新」が、**本書の鬼門**だと思います。ここはデータベースのテーブルに格納されたレコードをオブジェクトのように操作するSpring Data JPAが真価を発揮するところなのですが、ソースコードだけではなかなかイメージしにくいところでもあります。これも前書の「参照」と同じように、パラパラ漫画風の図も使って説明します。

これが理解できると、「部署と従業員」「売上と売上明細」といった、1:n(１対多)の関係にあるデータの検索、追加、更新、削除処理(いわゆるCRUD)を容易に作成できるようになります。この辺も類書ではまとまった説明を見かけませんが、**実務では必ずといっていいほど必要になる機能**です。

後半は解説の比重がSpring BootそのものからAPIや規格といった周辺知識へ移ります。ここからは「Spring Bootの使いこなし」の領域であり、**入門を超えて実践レベル**と言えるでしょう。

前提知識

本書を読み進めるには「Spring Boot」「Java言語」「HTML」「リレーショナルデータベース/SQL」などの知識が必要です。おおよそ以下のようなレベルを想定としています。

・Spring Boot

前書「Spring Boot 3で始めるWebアプリケーション開発入門(基礎編)」で解説した内容

・Java言語

Listなどのコレクション、ジェネリックス、ストリーム/ライターがある程度わかればベストです。もし不安でしたら専門書を都度参照してください。

・HTML

form要素、input要素、table/th/tr/td要素などがある程度わかればOKです。

・リレーショナルデータベース/SQL

簡単なSQL文(SELECT,INSERT,UPDATE, DELETE)を知っている程度で十分です。

本書の進め方

前書と同様に各章の冒頭で何をやるか、何を作るかを説明します。あわせて画面のスクリーンショットやUMLのシーケンス図を提示します。その後プログラムをできるだけ細かく解説します。

プログラムは前の章で作ったSTS(SpringToolSuite)のプロジェクトをコピーして、そこへコード・定義を追加していきます。プロジェクトのコピー方法については、前書7章「補足：プロジェクトのコピー方法」などを参照してください。

前書から読み進めている方は、「**補足：演習課題の実装例**」から読み始めることをお勧めします。ここでは前書の最後に出題した演習課題の実装例を解説していますが、このプログラムに若干の変更を加えたTodolist7xが本書のスタート地点です。12章はこのプロジェクトをコピーして始めてください(手順は後述)。

本書の前提環境

本書で使用している環境を下表に示します。いずれも2022年12月時点の安定版です。実際にインストールするものとは多少バージョンが異なると思いますが、適宜読み替えてください。

なお前書で環境を作成した方は、**バージョンアップする必要はありません**。そのままで大丈夫です。新規にインストールしたい方は、前書2章などを参照してください。

No.	名称	Ver	機能	備考
1	Windows10(64bit)	-	-	
2	Chrome	107.0	Webブラウザ	
3	Eclipse Temurin JDK	17.0.5+8	Javaプログラム開発キット(Java Development Kit)	No.4 の前提ソフト
4	Spring Tool Suite 4	4.16.1	Spring Bootアプリ開発環境	Spring Boot 3.0.1
5	Eclipse Web開発ツール	3.27	HTML/CSS編集用プラグイン	
6	Lombok	1.18.24	Java定型コード生成ツール	
7	PostgreSQL	15.1	リレーショナルデータベース管理システム	

サポートサイト

本書掲載のプログラムは以下のURLから入手できます。追加情報があれば、あわせて掲載します。

https://kktworks.github.io/

kktworks@gmail.com(お問い合わせ)

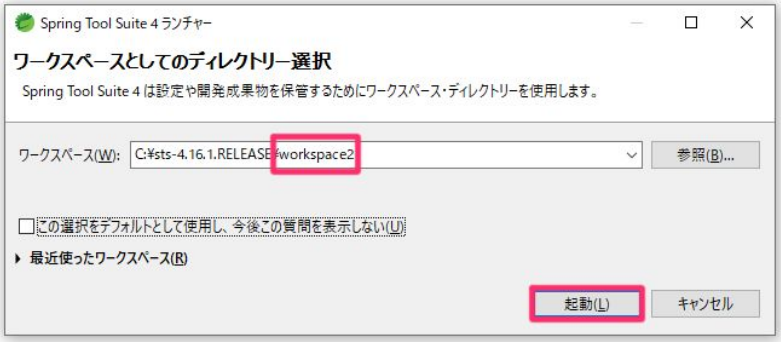
@kktworks1(Twitter)

本書開始の準備

最初に以下の手順でプロジェクトTodolist7xを作成します。12章のTodolist8は、これをコピーしてからコードや定義を追加していってください。

1)上記サポートサイトからソースコードをダウンロードし、任意のフォルダへ解凍する。
→c:¥tempへ解凍したとします

2)STSを起動する →「ワークスペースとしてのディレクトリー選択」が表示される。
前書ではworkspaceを使いましたが、本書でも多くのプロジェクトを作成するので別にした方が良いでしょう。
→本書では**<STS_DIR>¥workspace2**へ作成することにします。
(＜STS_DIR＞はsts-*.*.*.RELEASE.jarを解凍して作成したSTSのフォルダのこと)



3)プロジェクトTodolist7xの作成

3-1)STSのメニューから[ファイル(F)] > [新規(N)] > [Spring スターター・プロジェクト]を選択する。

3-2)「新規Springスターター・プロジェクト」ダイアログが表示される。

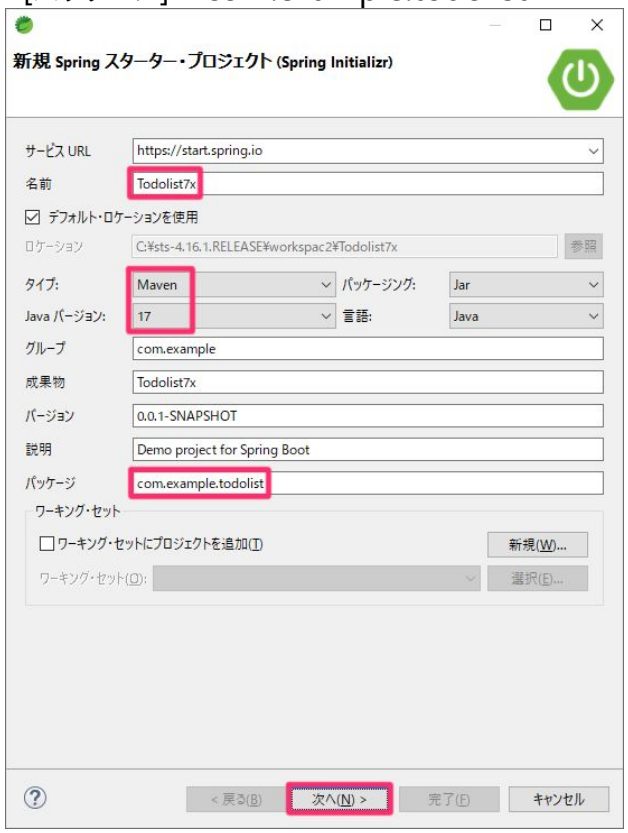
以下を入力して[次へ(N)]ボタンをクリックする。

・[名前]：Todolist7x

・[タイプ]：Maven

・[Javaバージョン]：17

・[パッケージ]：com.example.todolist



3-3)依存関係を設定する

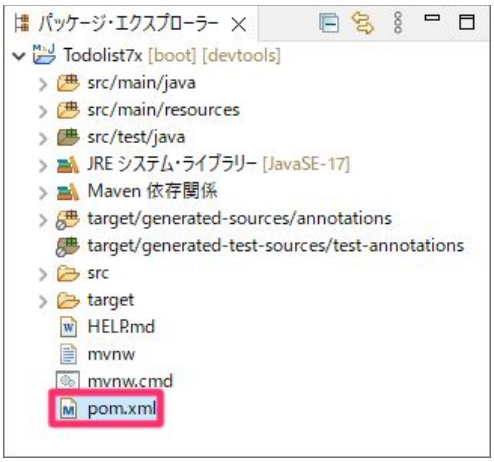
[Spring Bootバージョン]は**3.0.0以降**で**(SNAPSHOT)と付いてないもの**を選択してください。2.7.Xなどを選択すると、本書のプログラムで文法エラーになるところがあります。

さらに以下を選択し、[完了(F)]ボタンをクリックする。

Spring Web, Spring Boot DevTools, Thymeleaf, Lombok, Spring Data JPA, PostgreSQL Driver, Validation(検証)

4)メタクラス生成ツールhibernate-jpamodelgenの設定

4-1)パッケージ・エクスプローラーでTodolist7xプロジェクト直下にあるpom.xmlをダブルクリックして開く。


パッケージ・エクスプローラー
Todolist7x [boot] [devtools]
src/main/java
src/main/resources
src/test/java
JRE システム・ライブラリー [JavaSE-17]
Maven 依存関係
target/generated-sources/annotations
target/generated-test-sources/test-annotations
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml

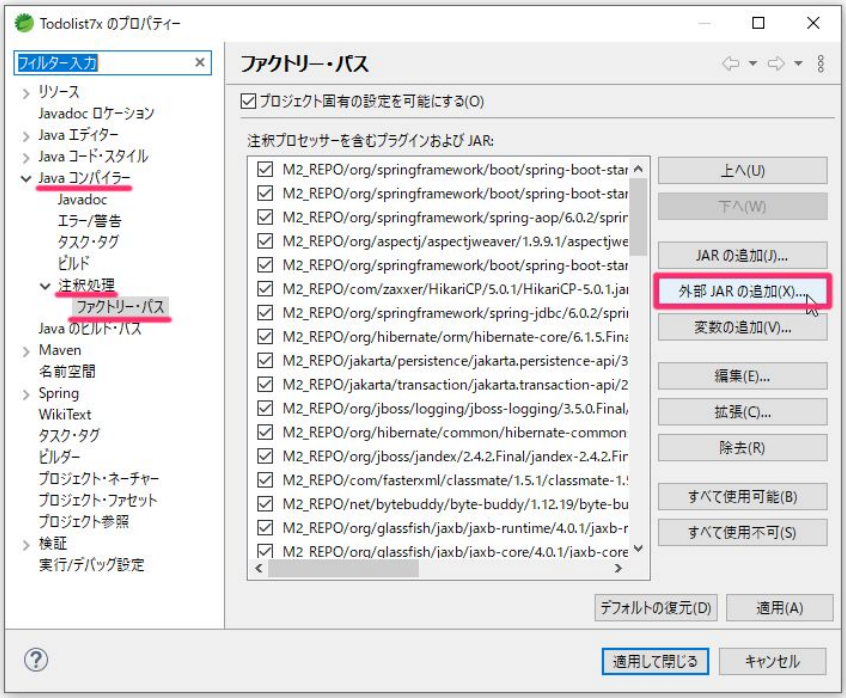
4-2)</dependencies>の直前に以下の内容を挿入して保存する([CTRL]+S)。
→このタイミングでjpamodelgenのjarファイルがダウンロードされる。
■格納場所
C:¥Users¥<ユーザー名>¥.m2¥repository¥org¥hibernate¥orm¥hibernate-jpamodelgen¥6.1.5.Final

```
<dependency>
....<groupId>org.hibernate.orm</groupId>
....<artifactId>hibernate-jpamodelgen</artifactId>
....<version>6.1.5.Final</version>
</dependency>
```

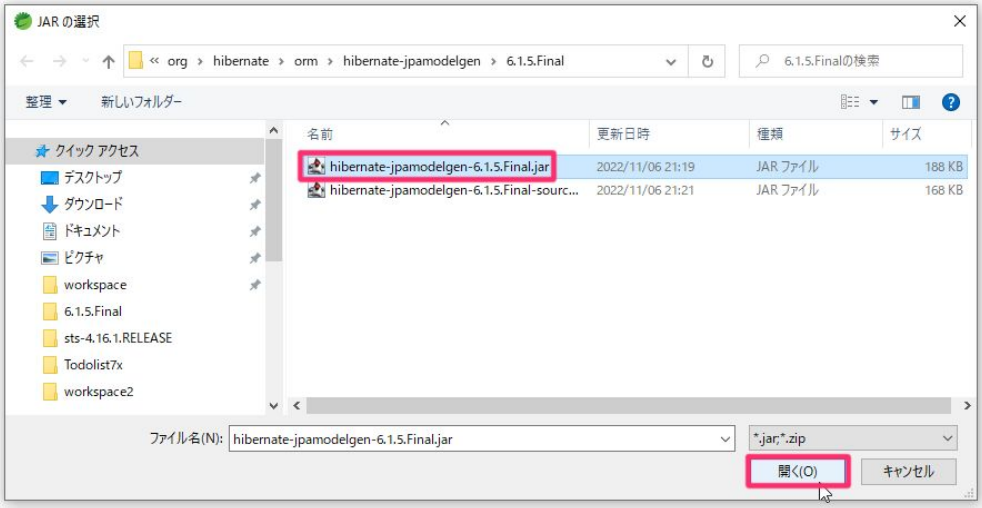
4-3)パッケージ・エクスプローラーでTodolist7xを右クリック > [プロパティ(R)] > [Javaコンパイラー]

> [注釈処理] > [ファクトリー・パス]をクリックする。

4-4)[外部JARの追加(X)...]ボタンをクリックする。

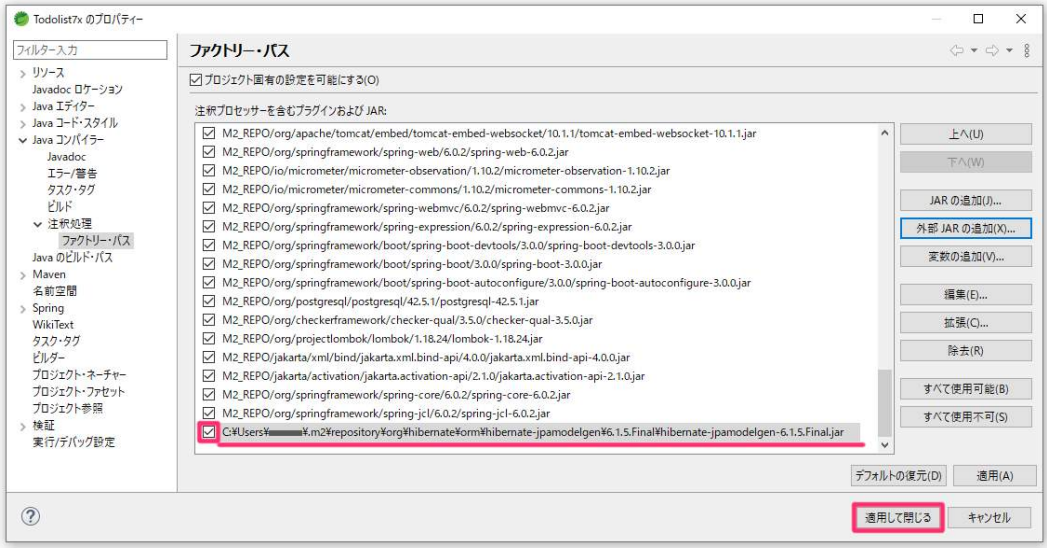


4-5)4-2)格納場所のhibernate-jpamodelgen-6.1.5.Final.jarを選択し[開く(O)]ボタンをクリックする。

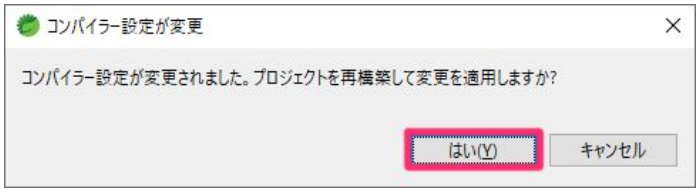


4-6)[注釈プロセッサーを含むプラグインおよびJAR]に4-5)で選択したjarファイルがチェックされていることを確認して

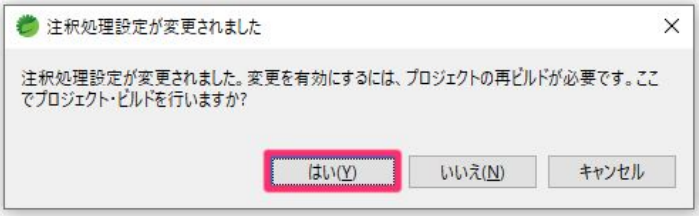
[適用して閉じる]ボタンをクリックする。



4-7)「コンパイラー設定が変更」ダイアログが表示された場合は、[はい(Y)]ボタンをクリックする。



4-8)「注釈処理設定が変更されました」ダイアログに対して[はい(Y)]ボタンをクリックする。

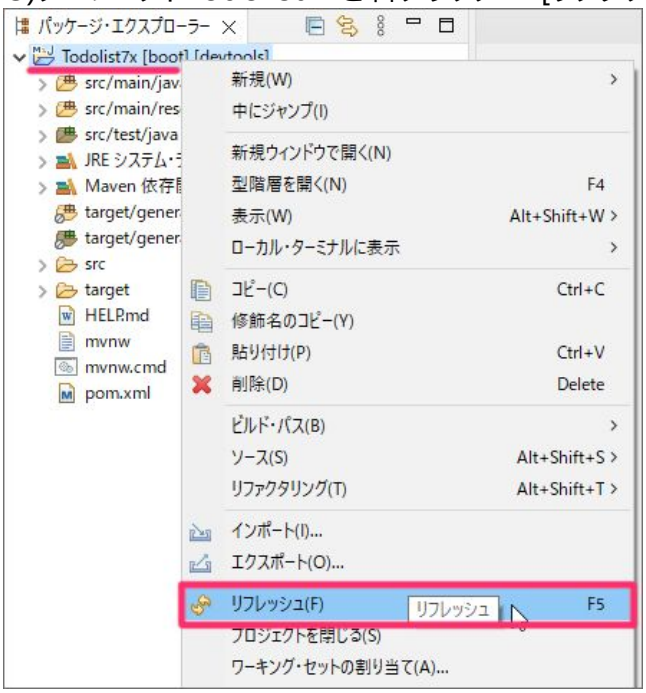


5)ソースコードのコピー

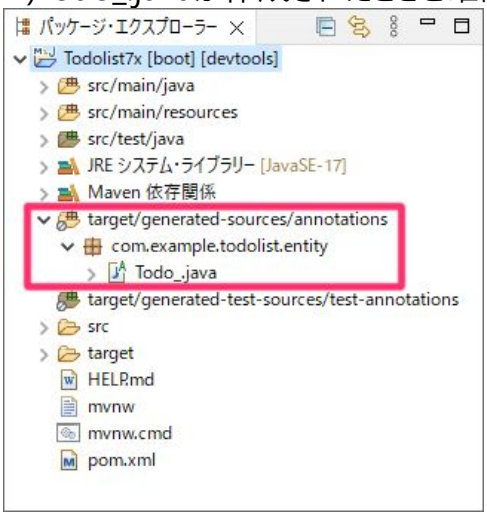
5-1)エクスプローラーで<STS_DIR>¥workspace2¥Todolist7xを開く。

5-2)1)で解凍したTodolist7x下のsrcフォルダを上記5-1)へ上書きコピーする

5-3)プロジェクトTodolist7xを右クリック > [リフレッシュ(F)]をクリックする。



5-4)Todo_.javaが作成されたことを確認する。



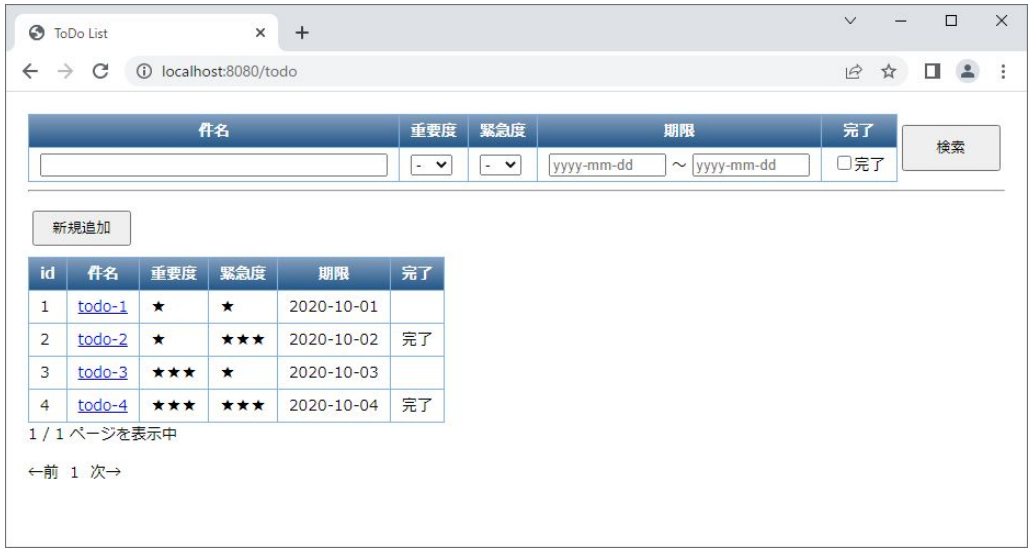
作成されない場合は、STSのメニュー　[プロジェクト(P)] > [クリーン(N)...] > [すべてのプロジェクトをクリーン(A)]をチェックして、[クリーン(C)]ボタンを押してみてください。

6)todoテーブルの作成

todoテーブルを作成していなければ、前書6章最後にある「todoテーブル作成手順」に従いテーブルを作成する。

7)Todolist7xを起動し、http://localhost:8080/todoへアクセスする。

→ToDo一覧が表示されることを確認する。



以上で本書を開始する準備は終わりです。

12. 国際化対応

12.1 国際化とは

前書で作成したToDoアプリは、見出しやエラーメッセージなどを日本語で表示しましたが、本章では英語でも表示できるようにします。このようにメッセージなどを特定の言語に固定せず切り替え可能にすることを「**国際化**」と言います。これには**ロケール(java.util.Locale)**と呼ばれる言語や国・地域を表す情報を利用します。

> 厳密に言えば年月日の表記形式、使用する暦、時差、通貨の表記形式なども国際化の対象となります。しかし本書では「ブラウザの表示言語設定に基づくメッセージ切り替え」を「国際化」として扱います。
> また国際化は「**i18n**」と略されることがあります。これはinternationalization(国際化)の先頭iと語尾nの間が18文字(nternationalizatio)あることに由来します。

12.2 ロケール(Locale)

Webブラウザには「**何語で表示するか？**」という設定ができます。そしてその情報はリクエストと一緒にサーバーへ送られています。ToDo一覧画面(todoList.html)に次の1行を追加すると、実際に確認できます。

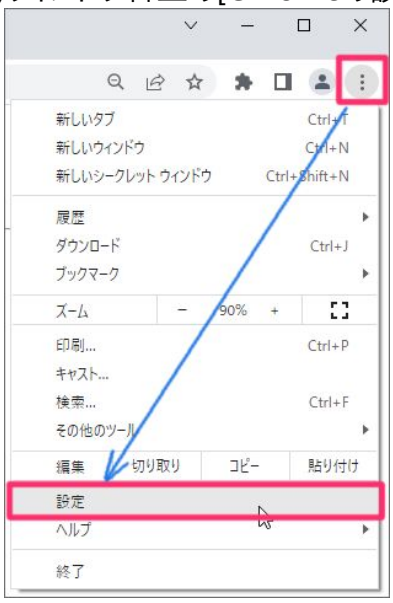
```
 :
<body>
  ${#locale.language}=<span th:text="${#locale.language}"></span>
   <form th:action="@{/}" method="post" th:object="${todoQuery}">
 :
```

#localeはクライアントから送られてきたロケールを表すThymeleafのオブジェクトです。そしてlanguageプロパティは言語名を表します。この状態でブラウザからhttp://localhost:8080/todoをアクセスすると、おそらく以下のようになるでしょう。

表示された**"ja"が言語名**です。つまりこのブラウザは日本語で表示するよう設定されています。

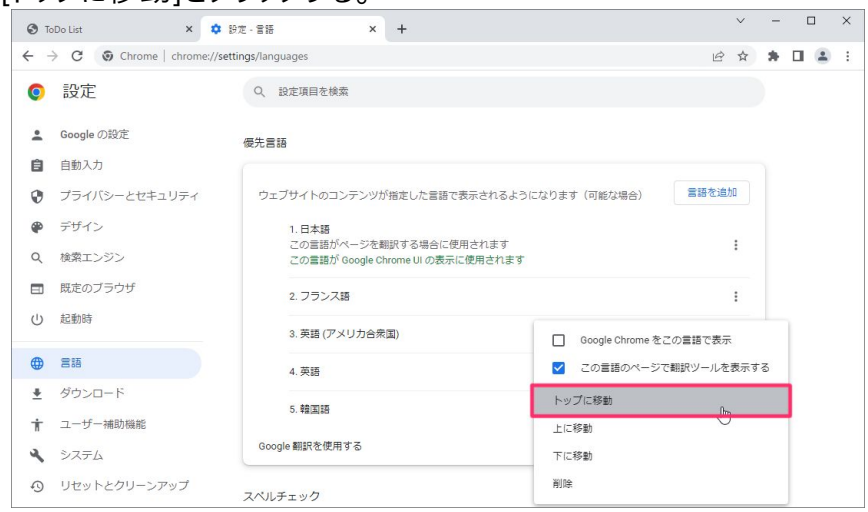
次に言語設定を英語に変更してみます。以下はChromeの場合ですが、Firefox, Edge(Chromium版)も同様の操作で変更可能です。

1)ウィンドウ右上の[Chromeの設定]ボタンをクリック > [設定]をクリックする。

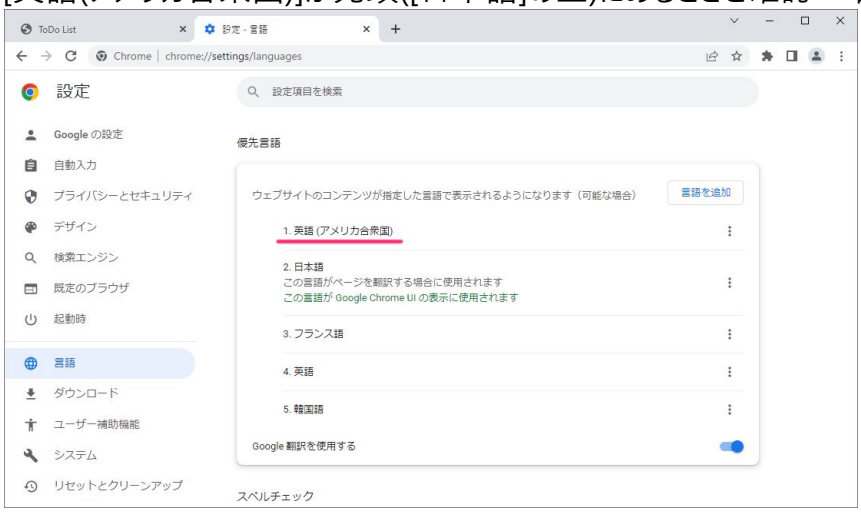


2)左側メニューの[言語] > [英語(アメリカ合衆国)]の右端[その他の操作]をクリックする。

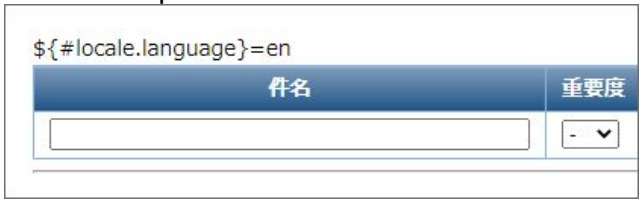
3)[トップに移動]をクリックする。



4) [英語(アメリカ合衆国)]が先頭([日本語]の上)にあることを確認 > 設定タブを閉じる。



これで再びhttp://localhost:8080/todoをアクセスすると、今度は"**en**"と表示されます。



このようにサーバーは、ロケールを通してブラウザの表示言語を判別できます。本章ではこれを利用して以下の項目を国際化(日本語/英語の動的切替え表示)していきます。

(1)見出し、ボタンのキャプション、リンクなど固定表示する文字列

(2)バリデーションエラーのメッセージ

(3)独自チェック(サービス)のエラーメッセージ

12.3 固定表示文字列の国際化

作成するプロジェクトの仕様

プロジェクト名	Todolist8
作成ファイル	src/main/resources/i18n/FixedDisplayStrings.properties src/main/resources/i18n/FixedDisplayStrings_en.properties src/main/resources/i18n/FixedDisplayStrings_ja.properties

```
Todolist8
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist8Application.java
 :    :
 :
 └ src/main/resources
    ├ i18n(★)
    │    ├ FixedDisplayStrings.properties(★)
    │    ├ FixedDisplayStrings_en.properties(★)
    │    └ FixedDisplayStrings_ja.properties(★)
    :
    ├ templates
    │    ├ todoForm.html(▲)
    │    └ todoList.html(▲)
    └application.properties(▲)
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

12.3.1 プロパティファイルの作成

言語によって表示内容を切り替えるには、「**プロパティファイル**」というものを**言語別**に用意します。これは、以下のように「キー名称=文字列」という行の集まりです。

```
#コメント行
キー名称1=文字列1
キー名称2=文字列2
：
#例
label.title=件名
```

最後の行は、キー名称「label.title」に対応する文字列として「件名」を定義しています。

デフォルトのプロパティファイルはsrc/main/resources/messages.propertiesです。しかし本章では日本語用と英語用の2ファイル必要なので、以下命名規則に従ってファイル名を決めます。

(1)ファイル名：**任意の文字列_言語名.properties**

・任意の文字列

messagesとする場合が多いようです。しかしここでは「固定表示文字列」を格納するので、少々違和感があります。そこで**FixedDisplayStrings**とします。

・言語名

前述した#locale.languageで表示されたものです。日本語用は**ja**、英語用は**en**です。

・拡張子

.properties固定です。

(2)格納場所：src/main/resources直下、またはそのサブフォルダ

・後者はプロパティファイル専用フォルダを作成することでresources直下をすっきりさせておきたい、という意図があるように思います。本章でもプロパティファイルは src/main/resources/**i18n**へ格納します。

以上をまとめるとプロパティファイルは次のようになります。

日本語用：src/main/resources/**i18n/FixedDisplayStrings_ja.properties**

英語用：src/main/resources/**i18n/FixedDisplayStrings_en.properties**

この結果本章のプロパティファイルはデフォルト(src/main/resources/messages.properties)とは異なるので、それをapplication.propertiesを通してSpring Bootに知らせます。

【リスト12-1】src/main/resources/application.properties(追加分)

```
#Add next line
spring.messages.basename=i18n/FixedDisplayStrings
```

・**spring.messages.basename**

プロパティファイルの情報を表すキー名称

・**i18n/FixedDisplayStrings**

プロパティファイルが(src/main/resources直下の)i18nフォルダにあり、名前がFixedDisplayStringsで始まる、ということを表している。

ここから実際にプロパティファイルを作成していきます。手順としては、以下のような流れになるかと思います。

1)固定表示文字列を抽出する

・画面を見ながら逐次抽出していきます。

・見出し、ボタンのキャプション、リンク文字列などが対象です。

2)キー名称を定義する

・1)の文字列を表すキー名称を決めます。

・キー名称は半角英数字とし、.(ピリオド)を使ってグループ化、階層化するのが一般的です。

3)言語ごとの文字列(語、文章)を決める

・日本語版は1)で抽出したものがベースになります。

・英語版はこれを翻訳して作成します。

→実務では翻訳家や専門家に依頼、あるはチェックしてもらった方が良いでしょう。

4)プロパティファイルの形にする

下表が(この段階の)ToDoアプリの固定表示文字列データです。キー名称は.(ピリオド)を使いグループ化しています。

【表12-1】固定表示文字列プロパティデータ

分類	キー名称	日本語(ja)	英語(en)
表見出し	label.title	件名	Title
	label.importance	重要度	Importance
	label.urgency	緊急度	Urgency
	label.period	期間	Period
	label.deadline	期限	Deadline
	label.check	チェック	Check
ボタン	button.query	検索	Query
	button.new	新規追加	New
	button.add	登録	Add
	button.update	更新	Update
	button.delete	削除	Delete
	button.cancel	キャンセル	Cancel
選択肢	option.none	選択してください	Select ...
	option.high	高	High
	option.low	低	Low

	radio.high	高	High
	radio.low	低	Low
リンク	link.prev	←前	< Prev
	link.next	次→	Next >
文字列	text.done	完了	Done
	text.paging	{0} / {1} ページを表示中	Showing page {0} / {1}

text.pagingの{0}, {1}は、表示するとき「表示中のページ」「総ページ数」で置き換えます(後述)。

これをプロパティファイルの形にすると以下のようになります。

【リスト12-2】src/main/resources/i18n/FixedDisplayStrings_ja.properties(日本語用)

```
#
label.title=件名
label.importance=重要度
label.urgency=緊急度
label.period=期間
label.deadline=期限
label.check=チェック
#
button.query=検索
button.new=新規追加
button.add=登録
button.update=更新
button.delete=削除
button.cancel=キャンセル
#
option.none=選択してください
option.high=高
option.low=低
radio.high=高い
radio.low=低い
#
link.prev=←前
link.next=次→
#
text.done=完了
text.paging={0} / {1} ページを表示中
```

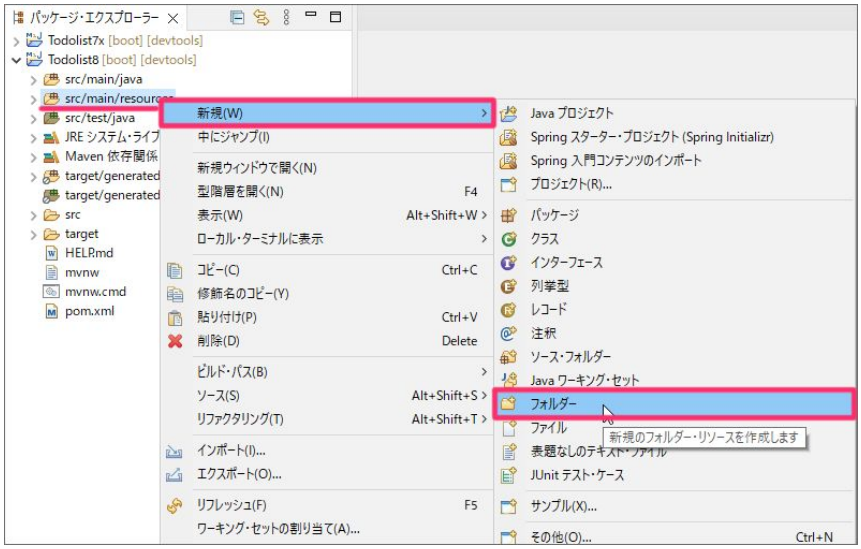
【リスト12-3】src/main/resources/i18n/FixedDisplayStrings_en.properties(英語用)

```
#
label.title=Title
label.importance=Importance
label.urgency=Urgency
```

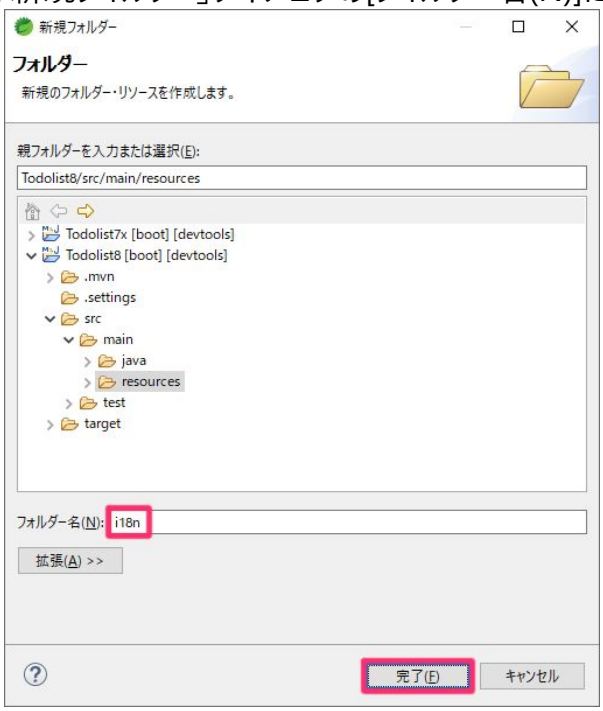
```
label.period=Period
label.deadline=Deadline
label.check=Check
#
button.query=Query
button.new=New
button.add=Add
button.update=Update
button.delete=Delete
button.cancel=Cancel
#
option.none=Select ...
option.high=High
option.low=Low
radio.high=High
radio.low=Low
#
link.prev=< Prev
link.next=Next >
#
text.done=Done
text.paging=Showing page {0} / {1}
```

プロパティファイル作成の操作手順は以下のようになります。

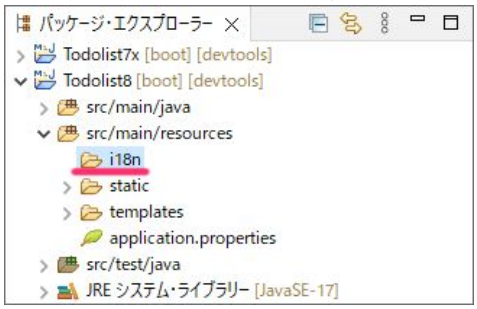
1)パッケージ・エクスプローラーでsrc/main/resourcesを右クリック > [新規(W)] > [フォルダー]をクリックする。



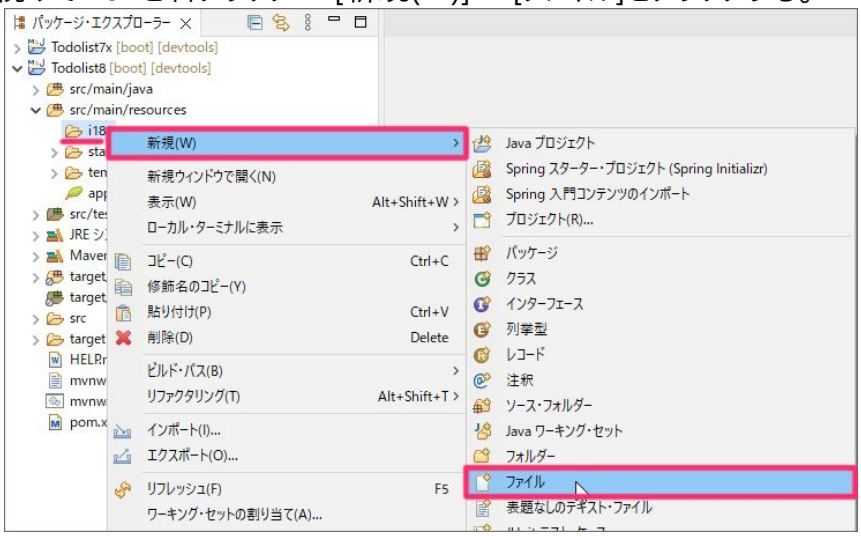
2)「新規フォルダー」ダイアログの[フォルダー名(N)]に i18n を入力 > [完了(F)]をクリックする。



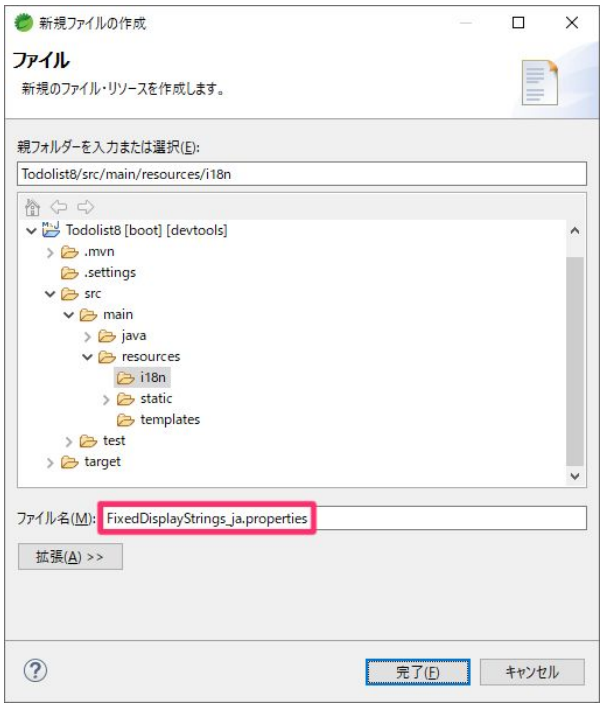
3)フォルダi18nが作成される。



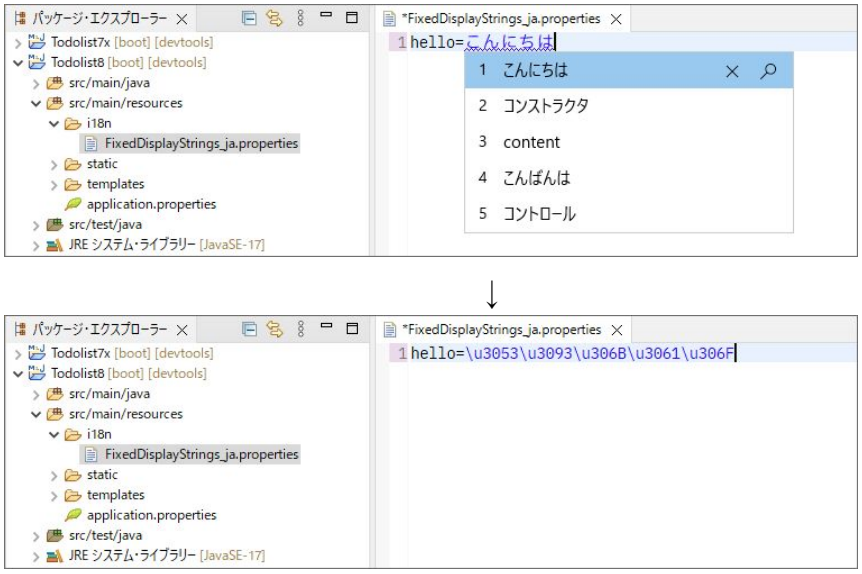
4)続けてi18nを右クリック > [新規(W)] > [ファイル]をクリックする。



5)「新規ファイルの作成」ダイアログで[ファイル名(M)]にFixedDisplayStrings_ja.propertiesを入力 > [完了(F)]をクリックし、日本語用を作成します。



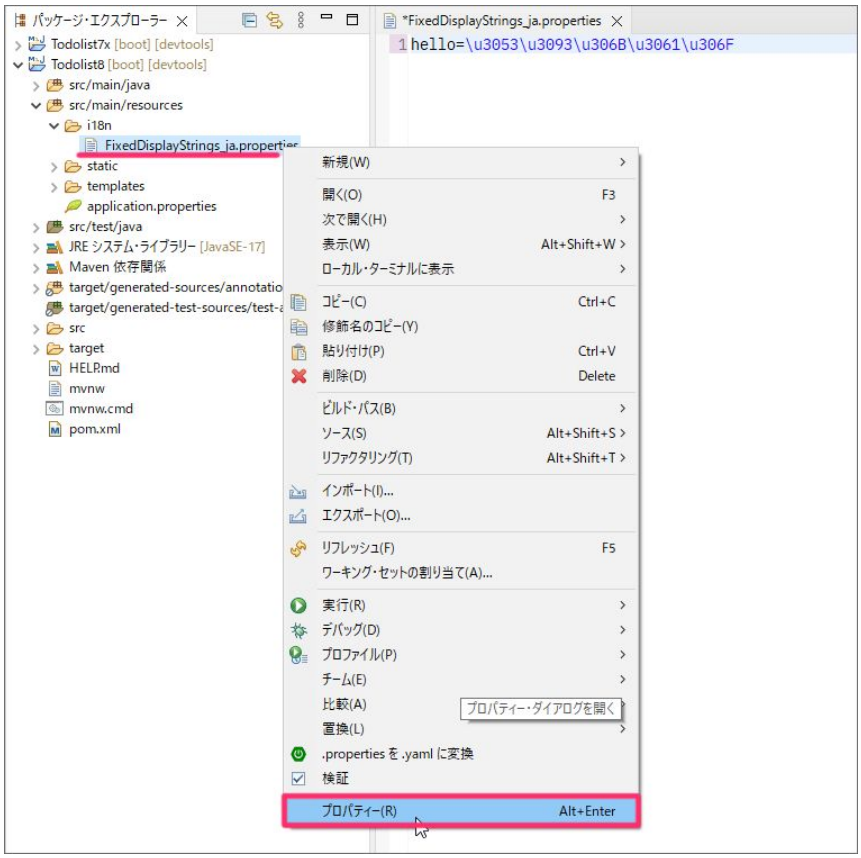
しかしこれだけでは、日本語を入力するとUnicodeにエンコード(¥uXXXX)されてしまいます。たとえば、「hello=こんにちは」を入力すると以下のようになります。



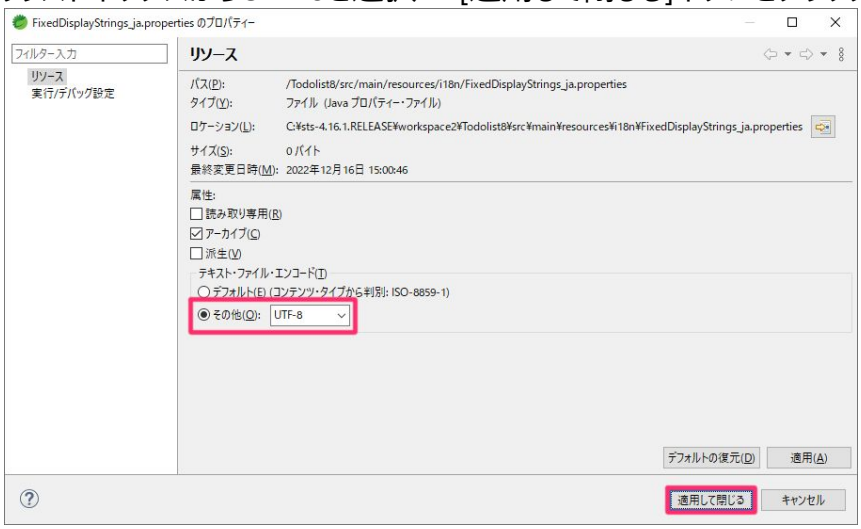
↓

この状態でもプロパティファイルとしては機能します。またエンコードされた部分にマウスカーソルを合わせると、元の日本語が表示されるので、何が書かれているか判別できます。しかしこれを変更するのはかなり面倒です。そこで次の手順で解消します。

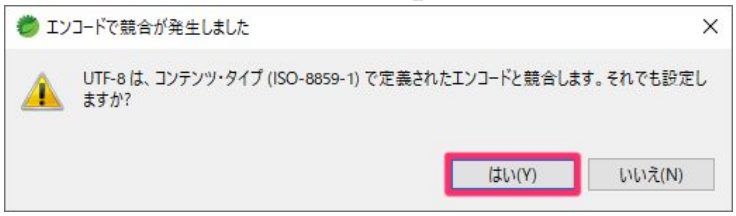
6)FixedDisplayStrings_ja.propertiesを右クリック > [プロパティ(R)]をクリック。


パッケージ・エクスプローラー
Todolist7x [boot] [devtools]
Todolist8 [boot] [devtools]
src/main/java
src/main/resources
i18n
FixedDisplayStrings_ja.properties
static
templates
application.properties
src/test/java
JRE システム・ライブラリー [JavaSE-17]
Maven 依存関係
target/generated-sources/annotatio
target/generated-test-sources/test-
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml
*FixedDisplayStrings_ja.properties
1 hello=\u3053\u3093\u306B\u3061\u306F
新規(W)
開く(O) F3
次で開く(H)
表示(W) Alt+Shift+W
ローカル・ターミナルに表示
コピー(C) Ctrl+C
修飾名のコピー(Y)
貼り付け(P) Ctrl+V
削除(D) Delete
ビルド・パス(B)
ソース(S) Alt+Shift+S
リファクタリング(T) Alt+Shift+T
インポート(I)...
エクスポート(O)...
リフレッシュ(F) F5
ワーキング・セットの割り当て(A)...
実行(R)
デバッグ(D)
プロファイル(P)
チーム(E)
比較(A)
置換(L)
プロパティー・ダイアログを開く
.properties を .yaml に変換
検証
プロパティー(R) Alt+Enter

7)プロパティダイアログで[テキスト・ファイル・エンコーディング(T)]の[その他(O)]を選択 > 右となりのリストボックスからUTF-8を選択 > [適用して閉じる]ボタンをクリックする。



8)「エンコードで競合が発生しました」ダイアログが表示されたら[はい(Y)]をクリックする。



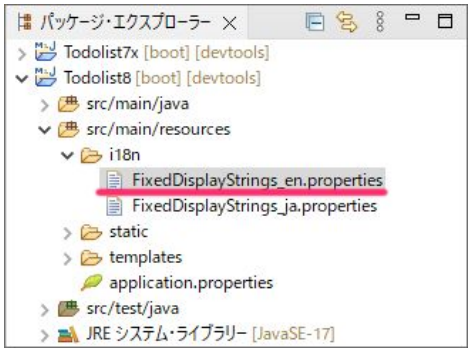
9)これでUnicodeにエンコードされなくなるので、【リスト12-2】の内容を入力します。

保存時以下のようなダイアログが表示されたら[UTF-8として保存(S)]ボタンをクリックしてください。


保存の問題
保存を完了できませんでした。問題が続く場合は ファイル > 別名保存... してください。
理由:
Some characters cannot be mapped using "ISO-8859-1" character encoding for file "/Todolist8/src/main/resources/i18n/FixedDisplayStrings_ja.properties".
エンコードを変更するか、"ISO-8859-1" 文字エンコードによってサポートされていない文字を除去してください。
OK
UTF-8 として保存(S)
最初の文字を選択(F)

10)同様にFixedDisplayStrings_en.propertiesも作成し、【リスト12-3】の内容を入力します。

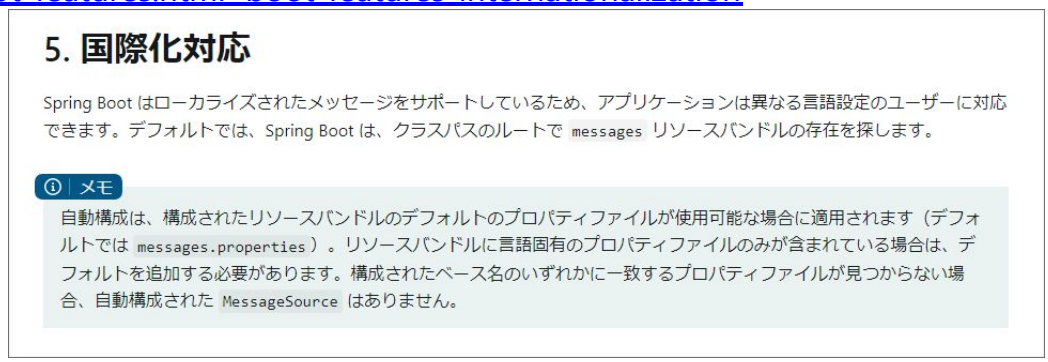
→エンコードの変更は不要です。



最後に「**デフォルトのプロパティファイル**」を作成します。これについては下記に説明があります。

■Spring Boot リファレンスドキュメント > Spring Boot の機能 > 5. 国際化対応

https://spring.pleiades.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-internationalization



5. 国際化対応

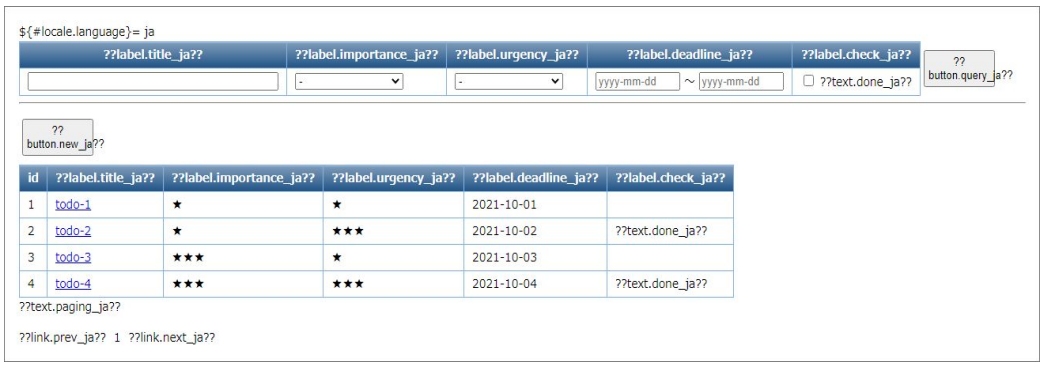
Spring Boot はローカライズされたメッセージをサポートしているため、アプリケーションは異なる言語設定のユーザーに対応できます。デフォルトでは、Spring Boot は、クラスパスのルートで `messages` リソースバンドルの存在を探します。

メモ

自動構成は、構成されたリソースバンドルのデフォルトのプロパティファイルが使用可能な場合に適用されます（デフォルトでは `messages.properties`）。リソースバンドルに言語固有のプロパティファイルのみが含まれている場合は、デフォルトを追加する必要があります。構成されたベース名のいずれかに一致するプロパティファイルが見つからない場合、自動構成された `MessageSource` はありません。

ポイントはメモ(網掛け)の部分です。少々難解ですが「**言語別にプロパティファイルを作成するときは、デフォルトのプロパティファイルも用意せよ**」という意味です。これは本節にも当てはまります。

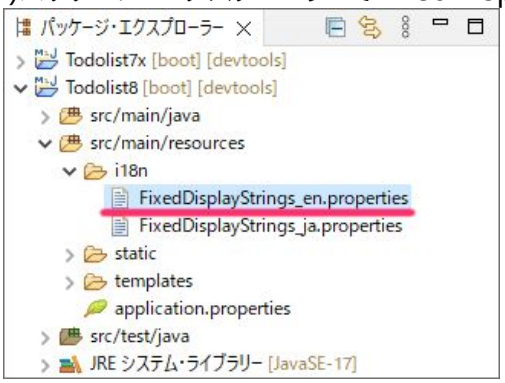
前述したようにデフォルトのプロパティファイルはsrc/main/resources/messages.propertiesです。しかしapplication.propertiesにプロパティファイル名を定義した場合は、**定義した名称から"_言語名"を除いたもの**になります。よってToDoアプリでは i18n/FixedDisplayStrings.propertiesがデフォルトプロパティファイルであり、これを作成しておく必要があります。これが無いと以下のようになってしまうので注意してください。



【図12-1】デフォルトのプロパティファイルが無い場合

デフォルトプロパティファイルは空でも良いのですが、英語版をコピーしたものとします(次節で使います)。

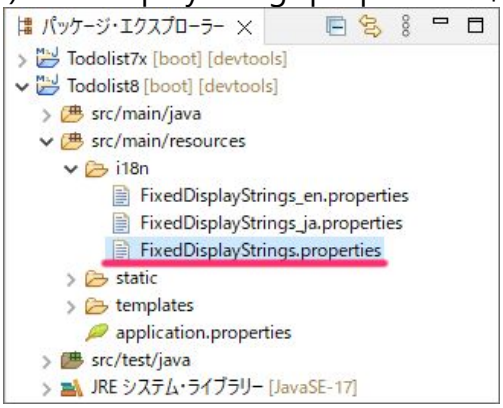
1)パッケージ・エクスプローラーでFixedDisplayStrings_en.propertiesをクリックする。



2)[CTRL] + [C], [CTRL] + [V]を押す(コピー　→　ペースト)。

3)「名前の競合」ダイアログが表示されるので、ファイル名をFixedDisplayStrings.propertiesに変更して[OK]をクリックする。

4)FixedDisplayStrings.propertiesが作成される。



12.3.2 ビューにプロパティファイルの内容を表示する

ではプロパティファイルを使って固定表示文字列を切り替えられるようにします。

Thymeleafではプロパティファイルの文字列を画面(ビュー)へ表示するのに**メッセージ式(#{})**を使います。#{}内にキー名称を書くと、Thymeleafがブラウザの表示言語に一致するプロパティファイルから対応する文字列を探し、これに置き換えます。

たとえば以下のようなspan要素があったとします。

```
<span th:text="#{label.title}"></span>
```

アクセスしてきたブラウザが日本語表示であれば
src/main/resources/i18n/FixedDisplayStrings_ja.properties
が使われます。

この中では「label.title=件名」と定義しているので、Thymeleafは次のようなspan要素にします。

```
<span>件名</span>
```

同様に英語表示のブラウザなら
src/main/resources/i18n/FixedDisplayStrings_en.propertiesが使われます。こちらでは「label.title=Title」としているので次のようになります。

```
<span>Title</span>
```

こういった仕組みによりプログラムを変更することなく固定見出しを切り替えることができます。

以下は固定表示している文字列をメッセージ式に置き換えたToDoリスト一覧画面(todoList.html)です。

【リスト12-4】src/main/resources/templates/todoList.html(一部抜粋)

```
<!DOCTYPE html>
  :
<body>
  ${#locale.language}=<span th:text="${#locale.language}"></span>
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
    <!-- 検索条件入力エリア -->
    <div style="display: flex">
      <table border="1">
        <tr>
          <th th:text="#{label.title}"></th>    <!-- ①-1 -->
          <th th:text="#{label.importance}"></th>
          <th th:text="#{label.urgency}"></th>
          <th th:text="#{label.deadline}"></th>
          <th th:text="#{label.check}"></th>
        </tr>
        <tr>
          <!-- 件名 -->
          <td>
            <input type="text" name="title" size="40" th:value="*{title}">
          </td>
          <!-- 重要度 -->
          <td>
            <select name="importance">
              <option value="-1" th:field="*{importance}">-</option>
              <!-- ①-2 -->
              <option value="1" th:field="*{importance}" th:text="#
{option.high}"></option>
              <option value="0" th:field="*{importance}" th:text="#
{option.low}"></option>
            </select>
          </td>
          <!-- 緊急度 -->
          <td>
            <select name="urgency">
```

```
                <option value="-1" th:field="*{urgency}">-</option>
                <option value="1" th:field="*{urgency}" th:text="#{option.high}">
</option>
                <option value="0" th:field="*{urgency}" th:text="#{option.low}">
</option>
            </select>
        </td>
        <!-- 期限 -->
        <td>
            <input type="text" name="deadlineFrom" th:value="*
{deadlineFrom}" size="10"
                placeholder="yyyy-mm-dd">
            ～
            <input type="text" name="deadlineTo" th:value="*{deadlineTo}"
size="10"
                placeholder="yyyy-mm-dd">
        </td>
        <!-- 完了 -->
        <td>
            <input type="checkbox" value="Y" th:field="*{done}">
            <span th:text="#{text.done}"></span><!-- ①-3 -->
        </td>
    </table>
    <!-- 検索ボタン -->
    <button type="submit" th:formaction="@{/todo/query}"
            th:text="#{button.query}"></button>    <!-- ①-4 -->
</div>
<!-- エラーメッセージエリア -->
<div th:if="${#fields.hasErrors('deadlineFrom')}" th:errors="*{deadlineFrom}"
    th:errorclass="red"></div>
<div th:if="${#fields.hasErrors('deadlineTo')}" th:errors="*{deadlineTo}"
    th:errorclass="red"></div>
<hr>
<!-- 新規追加ボタン -->
```

```
    <button type="submit" th:formaction="@{/todo/create/form}"
            th:text="#{button.new}"></button>
  </form>
  <!-- 検索結果エリア -->
  <table border="1">
    <tr>
      <th>id</th>
      <th th:text="#{label.title}"></th>
      <th th:text="#{label.importance}"></th>
      <th th:text="#{label.urgency}"></th>
      <th th:text="#{label.deadline}"></th>
      <th th:text="#{label.check}"></th>
    </tr>
    <tr th:each="todo:${todoList}">
      <!-- id -->
      <td th:text="${todo.id}"></td>
       <!-- 件名 -->
      <td>
        <a th:href="@{/todo/__${todo.id}__}" th:text="${todo.title}"></a>
      </td>
      <!-- 重要度 -->
      <td th:text="${todo.importance == 1 ? '★★★':'★'}"></td>
      <!-- 緊急度 -->
      <td th:text="${todo.urgency == 1 ? '★★★':'★'}"></td>
      <!-- 期限 -->
      <td th:text="${todo.deadline}"></td>
      <!-- 完了 -->
      <td th:text="${todo.done == 'Y' ? '__#{text.done}__' : ''}"></td>    <!-- ② --
>
    </tr>
  </table>
  <!-- ページリンク -->
  <div th:if="${todoPage != null}">
    <!-- 表示ページ位置 -->
```

```
      <!-- ③ -->
      <span th:text="#{text.paging(${todoPage.getNumber() +
1},${todoPage.getTotalPages()})}">
</span>
      <ul id="nav">
        <!-- ←前 -->
        <li>
          <span th:if="${todoPage.isFirst()}" th:text="#{link.prev}"></span>
          <a th:unless="${todoPage.isFirst()}"
             th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}"
            th:text="#{link.prev}"></a>    <!-- ①-5 -->
        </li>
      :
        <!-- 次→ -->
        <li>
          <span th:if="${todoPage.isLast()}" th:text="#{link.next}"></span>
          <a th:unless="${todoPage.isLast()}"
             th:href="@{/todo/query(page = (${todoPage.getNumber()+ 1}))}"
             th:text="#{link.next}"></a>
        </li>
      </ul>
    </div>
</body>
</html>
```

前述のspan要素と同じように、th:textにメッセージ式を書いています。以下はこのtodoList.htmlで使用しているメッセージ式の例です(ブラウザは日本語表示とします)。

(1)単純な置換

・th要素

```
<th th:text="#{label.title}"></th>    <!-- ①-1 -->
↓
<th>件名</th>
```

・option要素

th:textのメッセージ式は選択肢になります。選択肢が表示言語で変わっても、サーバーへ送信するのはvalue要素の値なのでプログラムの変更は不要です。

```
<!-- ①-2 -->
<option value="1" th:field="*{importance}" th:text="#{option.high}">
</option>
<option value="0" th:field="*{importance}" th:text="#{option.low}"></option>
↓
<option value="1">高</option>
<option value="0">低</option>
```

・span要素

```
<span th:text="#{text.done}"></span><!-- ①-3 -->
↓
<span>完了</span>
```

・button要素

th:textのメッセージ式はボタンのキャプションになります。

```
<button type="submit" th:formaction="@{/todo/query}"
                th:text="#{button.query}"></button>    <!-- ①-4 -->
↓
<button type="submit" formaction="/todo/query">検索</button>
```

・a要素

th:textのメッセージ式はリンク文字列になります。

```
<a th:unless="${todoPage.isFirst()}"
   th:href="@{/todo/query(page = ${todoPage.getNumber() - 1})}"
   th:text="#{link.prev}"></a>    <!-- ①-5 -->
```

```
    ↓
    <a href="/todo/query?page=n">←前</a>
```

(2)事前評価

ToDoリストでは、ToDoが完了(done列='Y')であれば、チェック列に「完了」と表示していました。

```
<td th:text="${todo.done == 'Y' ? '完了' : ''}">
```

これを前述のように置き換えると、#{text.done}という文字列なってしまいます。

```
<td th:text="${todo.done == 'Y' ? '#{text.done}' : ''}">    <!-- NG -->
```

↓

id	件名	重要度	緊急度	期限	チェック
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	#{text.done}
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	#{text.done}
5	todo-5	★	★	2020-10-05	

1 / 8 ページを表示中

←前 1 2 3 次→

【図12-2】メッセージ式が文字リテラルとして扱われている

これは'#{text.done}'の部分が、(メッセージ式ではなく)文字列リテラルと解釈されるためです。こういった場合、メッセージ式の前後を__(アンダースコアを2つ)で囲みます。こうすると三項演算子(?:)を実行する前に、#{text.done}が文字列に置換されるため意図した表示となります。

```
    <!-- ブラウザの表示言語が日本語でtodo.done="Y"の場合 -->
    <td th:text="${todo.done == 'Y' ? '__#{text.done}__' : ''}"></td>    <!-- ② -->
    ↓
    <td th:text="${todo.done == 'Y' ? '完了' : ''}"></td>
    ↓
```

```
    <td>完了</td>
```

このように式を__で囲むことを**事前評価**と言います。

(3)プレースホルダー

【リスト12-2】【リスト12-3】のtext.pagingは、{0}, {1}という記号が含んでいました。これは表示する値と置き換えるための**プレースホルダー**(目印)です。

```
#【リスト12-2】FixedDisplayStrings_ja.properties
text.paging={0} / {1} ページを表示中

#【リスト12-3】FixedDisplayStrings_en.properties
text.paging=Showing page {0} / {1}
```

プレースホルダーと置換する値は#{キー名称({0}の値, {1}の値, ...)}という形式で指定します。実際todoList.htmlでは以下のようにしています。

```
    <!-- ③ -->
    <span th:text="#{text.paging(${todoPage.getNumber() + 1},${todoPage.getTotalPages()})}">
```

これで{0}は表示中のページ、{1}は総ページ数になります。

ToDo入力画面(todoForm.html)も同様に、固定表示文字列をメッセージ式へ置き換えます。

【リスト12-5】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
    :
<body>
    <!-- 入力フォーム -->
    <form th:action="@{/}" method="post" th:object="${todoData}">
        <!-- ToDO入力エリア -->
        <table>
            <!-- id -->
            <tr>
                <th>id</th>
```

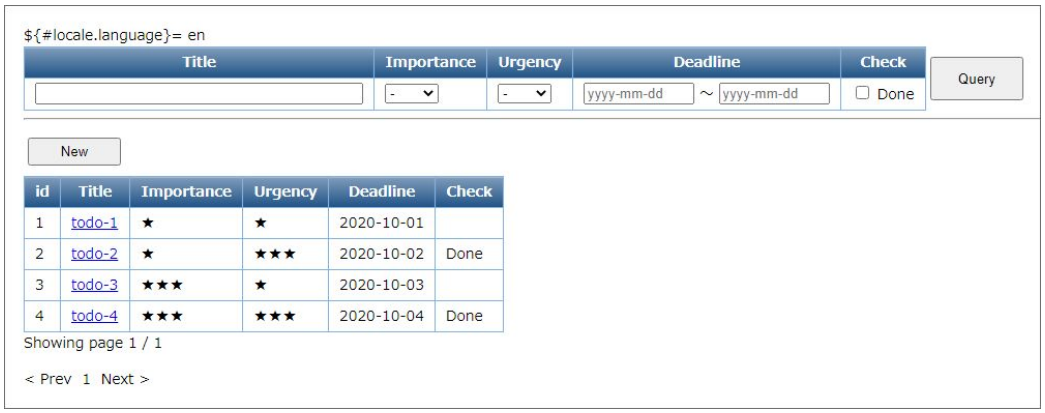
```
            <td>
              <span th:text="*{id}"></span>
              <!-- 更新 のために必要 -->
              <input type="hidden" th:field="*{id}">
            </td>
          </tr>
          <!-- 件名 -->
          <tr>
            <th th:text="#{label.title}"></th>
            <td>
              <input type="text" name="title" size="40" th:value="*{title}">
              <div th:if="${#fields.hasErrors('title')}"
                   th:errors="*{title}" th:errorclass="red"></div>
            </td>
          </tr>
          <!-- 重要度 -->
          <tr>
            <th th:text="#{label.importance}"></th>
            <td>
              <input type="radio" value="1" th:field="*{importance}">
              <span th:text="#{radio.high}"></span>
              <input type="radio" value="0" th:field="*{importance}">
              <span th:text="#{radio.low}"></span>
              <div th:if="${#fields.hasErrors('importance')}"
                   th:errors="*{importance}" th:errorclass="red"></div>
            </td>
          </tr>
          <!-- 緊急度 -->
          <tr>
            <th th:text="#{label.urgency}"></th>
            <td>
              <select name="urgency">
                <option value="-1" th:field="*{urgency}" th:text="#{option.none}">
</option>
```

```
                <option value="1" th:field="*{urgency}" th:text="#{option.high}">
</option>
                <option value="0" th:field="*{urgency}" th:text="#{option.low}">
</option>
            </select>
            <div th:if="${#fields.hasErrors('urgency')}"
                th:errors="*{urgency}" th:errorclass="red"></div>
          </td>
        </tr>
        <!-- 期限 -->
        <tr>
          <th th:text="#{label.deadline}"></th>
          <td>
            <input type="text" name="deadline" th:value="*{deadline}"
placeholder="yyyy-mm-dd">
            <div th:if="${#fields.hasErrors('deadline')}"
                th:errors="*{deadline}" th:errorclass="red"></div>
          </td>
        </tr>
        <!-- チェック -->
        <tr>
          <th th:text="#{label.check}"></th>
          <td>
            <input type="checkbox" value="Y" th:field="*{done}">
            <span th:text="#{text.done}"></span>
            <input type="hidden" name="!done" value="N" />
          </td>
        </tr>
      </table>
      <!-- 更新時の操作ボタン -->
      <div th:if="${session.mode == 'update'}">
        <button type="submit" th:formaction="@{/todo/update}" th:text="#
{button.update}"></button>
```

```
        <button type="submit" th:formaction="@{/todo/delete}" th:text="#
{button.delete}"></button>
        <button type="submit" th:formaction="@{/todo/cancel}" th:text="#
{button.cancel}"></button>
      </div>
      <!-- 新規追加時の操作ボタン -->
      <div th:unless="${session.mode == 'update'}">
        <button type="submit" th:formaction="@{/todo/create/do}" th:text="#
{button.add}"></button>
        <button type="submit" th:formaction="@{/todo/cancel}" th:text="#
{button.cancel}"></button>
      </div>
    </form>
</body>
</html>
```

これで固定表示文字列がブラウザの言語設定(日本語/英語)に応じて切り替わります。

※もし【図12-1】のように文字化けした場合は、一度Todolist8を停止→再実行してみてください。



【図12-3】表示言語を英語にした場合

12.3.3 デフォルトのプロパティファイル

ここまで3種類のプロパティファイル(日本語、英語、デフォルト)を作成しました。
・FixedDisplayStrings_ja.properties
・FixedDisplayStrings_en.properties
・FixedDisplayStrings.properties(内容は英語版のコピー)

ではここで言語設定がフランス語(fr)のブラウザからアクセスしたらどうなるでしょう？フランス語用は作成していないので、デフォルトが使われ英語表記になりそうですが、著者の環境では日本語で表示されます(jaを見ている)。Spring Bootは、言語に対応するプロパティファイルが無い場合、サーバーシステムのローケルで決めます(実際にはもう少し複雑な仕組みになっています)。実際、著者のサーバー環境(=STSをインストールしたPC)は日本語設定なので、上記のようになるわけです。しかしデフォルトのプロパティファイルを表示したいというケースもあると思います。方法はいくつかありますが、もっとも簡単なのはapplication.propertiesに次の1行を追加するものです。

【リスト12-6】src/main/resources/application.properties(追加分)

```
spring.messages.fallback-to-system-locale=false
```

これでデフォルトのプロパティファイルが使われるようになります。以下、実際に試してみます。

デフォルトのプロパティファイル作成

ここまで本書通りに進めてきた方は、デフォルトのプロパティファイルが英語用と同じになっています。そこで判別しやすくするため、デフォルトプロパティファイルのlabel.titleを"Title(default)"へ変更します。

【リスト12-7】src/main/resources/i18n/FixedDisplayStrings.properties(変更箇所)

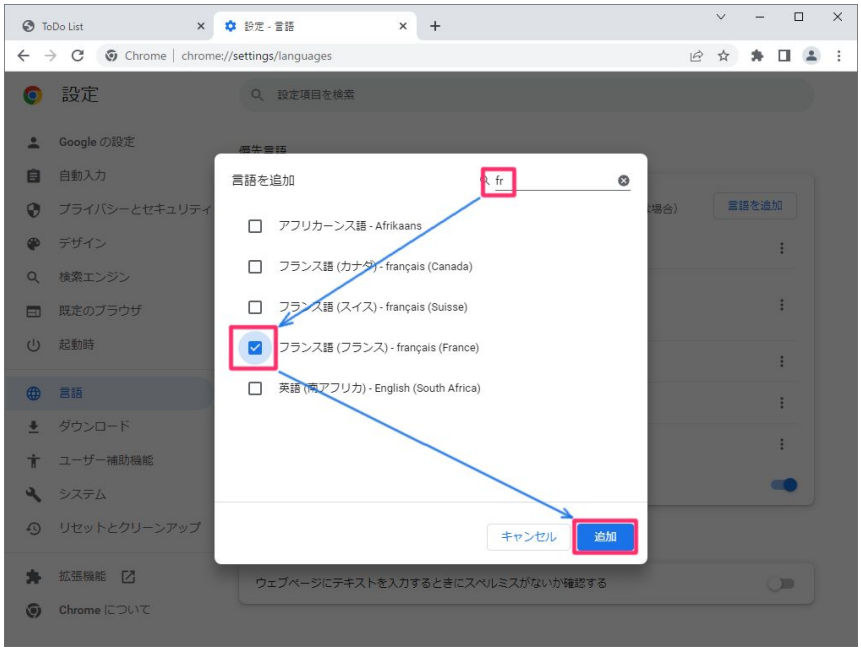
```
#
label.title=Title(default)
label.importance=Importance
label.urgency=Urgency
        :
```

ブラウザの言語設定

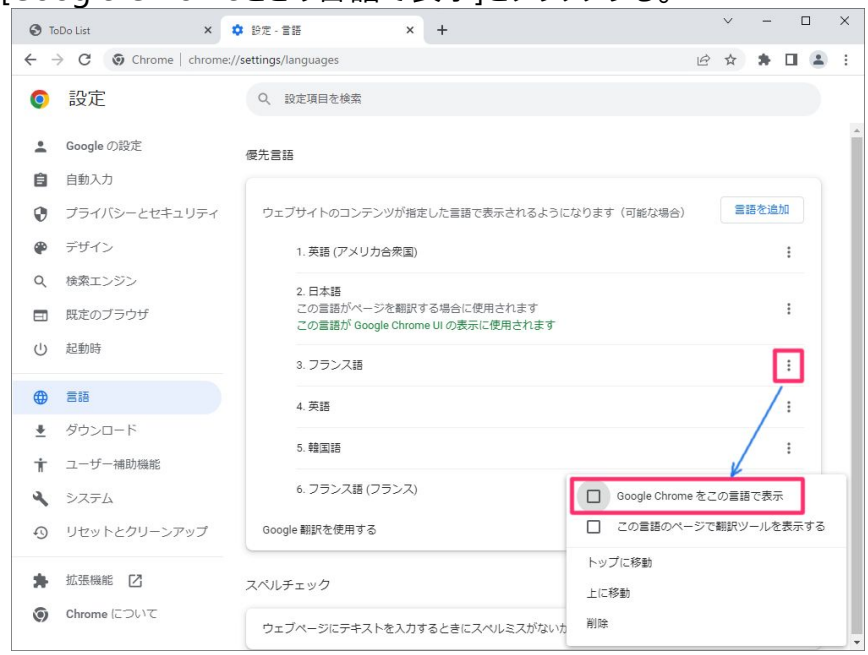
では言語設定をフランス語(fr)に変更して確かめてみます。手順は以下の通りです。

1)ウィンドウ右上の[Chromeの設定]ボタンをクリック > [設定(S)]をクリックする。
2)左側メニューの [言語] > [言語を追加]をクリックする。

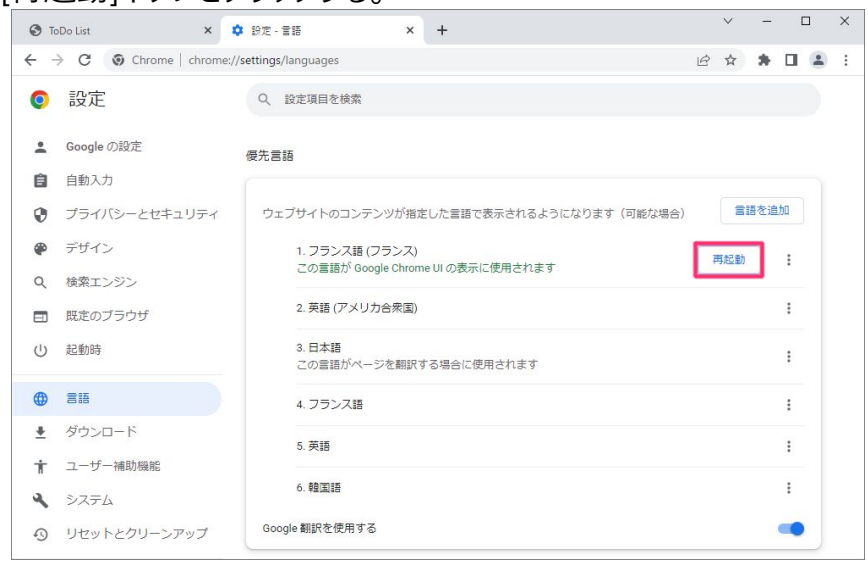
3)「言語を追加」ダイアログの右上に fr を入力 > フランス語(フランス)をチェック > [追加]ボタンをクリックする。



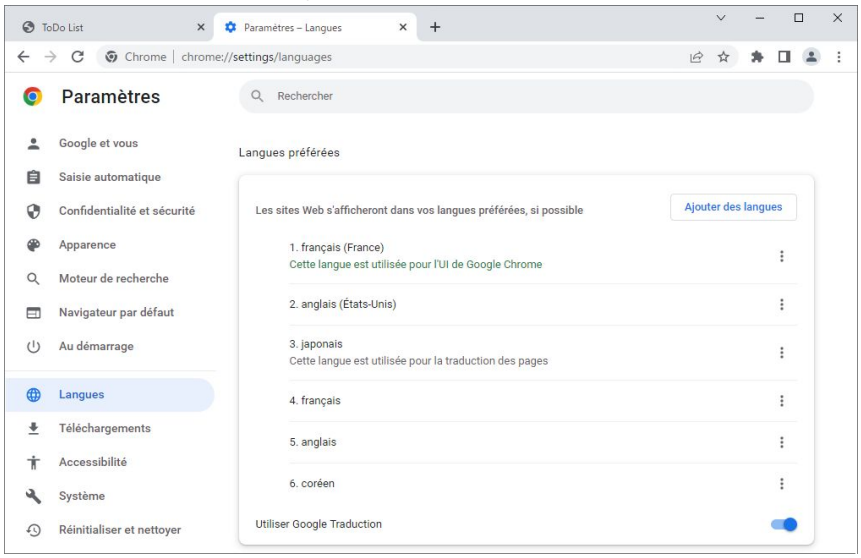
4)追加された[フランス語]の右端にある[その他の操作]をクリックする。
5)[Google Chromeをこの言語で表示]をクリックする。



6)[再起動]ボタンをクリックする。



7)フランス語表示に変わる。



実行

これでhttp://localhost:8080/todoをアクセスします。Title(default)と表示されているので、デフォルトのプロパティファイルを使っている、ということが確認できます。

id	Title(default)	Importance	Urgency	Deadline	Check
1	todo-1	★	★	2020-10-01	
2	todo-2	★	★★★	2020-10-02	Done
3	todo-3	★★★	★	2020-10-03	
4	todo-4	★★★	★★★	2020-10-04	Done
5	todo-5	★	★	2020-10-05	

Showing page 1 / 8

< Prev 1 2 3 Next >

【図12-4】デフォルトのプロパティファイルが使われている

12.4 バリデーションメッセージの国際化

前節では「固定」表示を切り替えられるようにしました。ここからは「動的」に出力するメッセージを国際化していきます。対象は次のものです。

1)@NotBlankなどのバリデーションで出力するメッセージ

2)サービス(com.example.todolist.service.TodoService.java)で出力するメッセージ

本節では1)の国際化を説明します。2)は次節で説明します。

作成するプロジェクトの仕様

プロジェクト名	Todolist8a
作成ファイル	src/main/resources/i18n/ValidationMessages_en.properties src/main/resources/i18n/ValidationMessages_ja.properties

```
Todolist8a
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist8aApplication.java
 │    ├ com.example.todolist.common
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ TodoData.java(▲)
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
```

```
|           └ TodoService.java
└ src/main/resources
    ├ i18n
    |    ├ FixedDisplayStrings.properties
    |    ├ FixedDisplayStrings_en.properties
    |    ├ FixedDisplayStrings_ja.properties
    |    ├ ValidationMessages_en.properties(★)
    |    └ ValidationMessages_ja.properties(★)
    :
    ├ templates
    |    ├ todoForm.html
    |    └ todoList.html
    └application.properties(▲)
```

★：このプロジェクトで追加する
▲：前プロジェクトの内容を一部変更する

バリデーションで表示するエラーメッセージは、以下のようにアノテーションのmessage属性で定義していました。これをプロパティファイルへ移します。

■com.example.todolist.form.TodoData.java(変更前、一部抜粋)

```java
package com.example.todolist.form;
    :
@Data
public class TodoData {
    private Integer id;

    @NotBlank(message = "件名を入力してください")
    private String title;

    @NotNull(message = "重要度を選択してください")
    private Integer importance;

    @Min(value = 0, message = "緊急度を選択してください")
    private Integer urgency;
```

```
    :
}
```

一般にバリデーションのメッセージを定義したプロパティファイルは**ValidationMessages_言語名.properties**といった名前にすることが多いようです。そこで本節もこれにならい、次のようにします。

日本語用：src/main/resources/i18n/ValidationMessages_ja.properties
英語用：src/main/resources/i18n/ValidationMessages_en.properties

そして以下の内容を入力します(次節以降さらに追加していきます)。

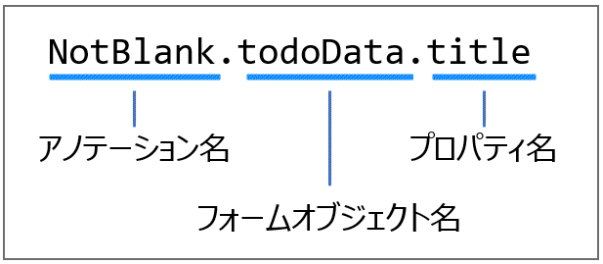
【リスト12-8】src/main/resources/i18n/ValidationMessages_ja.properties

```
#件名
NotBlank.todoData.title=件名を入力してください

#重要度
NotNull.todoData.importance=重要度を選択してください

#緊急度
Min.todoData.urgency=緊急度を選択してください
```

【リスト12-9】src/main/resources/i18n/ValidationMessages_en.properties

```
#Title
NotBlank.todoData.title=Title is required.

#Importance
NotNull.todoData.importance=Select the importance.

#Urgency
Min.todoData.urgency=Select the urgency.
```

定義する文字列はアノテーションのmessage属性で設定していたメッセージそのものです。
ポイントはキー名称です。以下のように . (ピリオド)で区切り3つの情報を記載します。



・1番目：アノテーション名(@は不要)
・2番目：フォームオブジェクト名
フォームクラス名ではなく、バリデーションするフォームオブジェクト名です。TodoDataはToDoの追加処理、更新処理でバリデーションをしますが、そのときのオブジェクト名は以下のようにtodoDataです。2番目に指定するのはこの名前です。

■TodoListController　Todo追加用ハンドラメソッド

```
// ToDo追加処理
@PostMapping("/todo/create/do")
public String createTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result,
                         Model model) { ...
```

■TodoListController　Todo更新用ハンドラメソッド

```
// ToDo更新処理
@PostMapping("/todo/update")
public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result,
                         Model model) { ...
```

・3番目：プロパティ名
バリデーションするフォームオブジェクトのプロパティ名です。

これでtodoDataオブジェクトのtitleプロパティで@NotBlankバリデーションがエラーになったら、「件名を入力してください」(英語設定のブラウザなら「Title is required.」)がBindingResultオブジェクトのresultへ追加されます。

エラーメッセージをプロパティファイルへ移したので、TodoDataからはmessage属性を削除します。

【リスト12-10】com.example.todolist.form.TodoData.java(変更箇所)

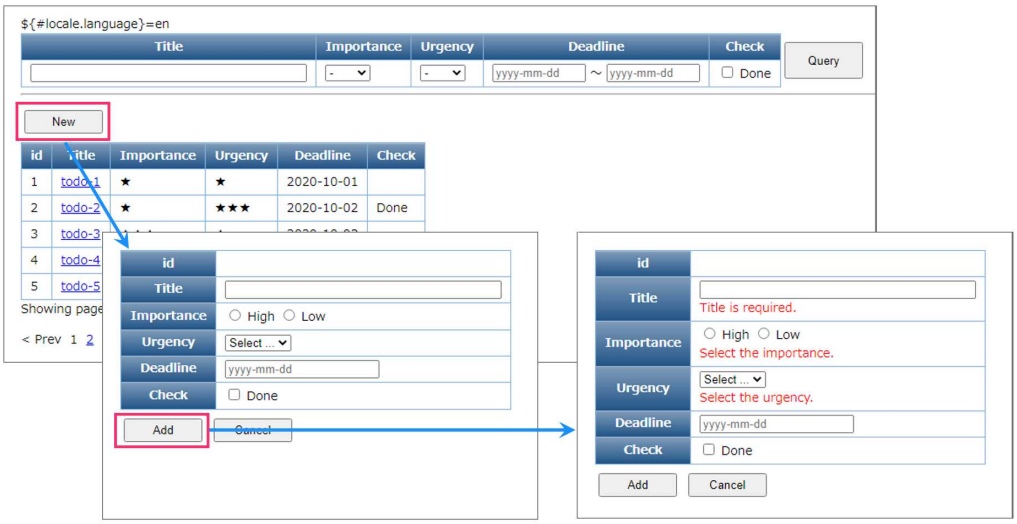
```
package com.example.todolist.form;
   :
@Data
public class TodoData {
	private Integer id;

	@NotBlank    // <- messageを削除
	private String title;

	@NotNull    // <- messageを削除
	private Integer importance;

	@Min(value = 0)    // <- messageを削除
	private Integer urgency;

	private String deadline;
	private String done;
    :
}
```

最後にこのプロパティファイルの存在をSpring Bootに知らせるため、application.propertiesに場所・名前を追記します。

【リスト12-11】src/main/resources/application.properties(追加箇所)

```
spring.messages.basename=i18n/FixedDisplayStrings
        ↓
spring.messages.basename=i18n/FixedDisplayStrings, i18n/ValidationMessages
```

バリデーションメッセージの国際化は以上になります。ブラウザが日本語表示のときは、これまでと変わりません。英語に変更すると、エラーメッセージも英語になることが確認できますので試してみてください。



【図12-5】英語のバリデーションエラーメッセージ

12.5 独自チェックメッセージの国際化

次はサービスクラスTodoServiceで出力している以下のエラーメッセージを国際化します。
・件名が全角スペースです
・期限を設定するときは今日以降にしてください
・期限を設定するときはyyyy-mm-dd形式で入力してください
・期限:開始を設定するときはyyyy-mm-dd形式で入力してください
・期限:終了を設定するときはyyyy-mm-dd形式で入力してください

作成するプロジェクトの仕様

プロジェクト名	Todolist8b

```
Todolist8b
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist8bApplication.java
 │    ├ com.example.todolist.common
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java(▲)
```

```
└ src/main/resources
    ├ i18n
    │      ├ FixedDisplayStrings.properties
    │      ├ FixedDisplayStrings_en.properties
    │      ├ FixedDisplayStrings_ja.properties
    │      ├ ValidationMessages_en.properties(▲)
    │      └ ValidationMessages_ja.properties(▲)
    :
    ├ templates
    │      ├ todoForm.html
    │      └ todoList.html
    └application.properties
```

★：このプロジェクトで追加する
▲：前プロジェクトの内容を一部変更する

前述のエラーメッセージは、TodoService内でFieldErrorオブジェクトに設定することでブラウザに表示されます。よって国際化するには、このメッセージを言語設定で変える必要があります。

■現在のメッセージ設定処理

```
FieldError fieldError = new FieldError(
        result.getObjectName(),
        "title",
        "件名が全角スペースです");
result.addError(fieldError);
```

国際化の方法はこれまでと同様に次の2ステップからなります。
(1)プロパティファイルにエラーメッセージを定義する
(2)(1)を取得して表示する。

(1)エラーメッセージの定義

エラーメッセージごとにキー名称を作成し、プロパティファイルへ追加します。キー名称は自由ですが、前節と同様に「エラー種別.フォームオブジェクト名.プロパティ名」という形にします。また追加先も前節のバリデーション用のプロパティファイルとします。

【リスト12-12】src/main/resources/i18n/ValidationMessages_ja.properties(追加分)

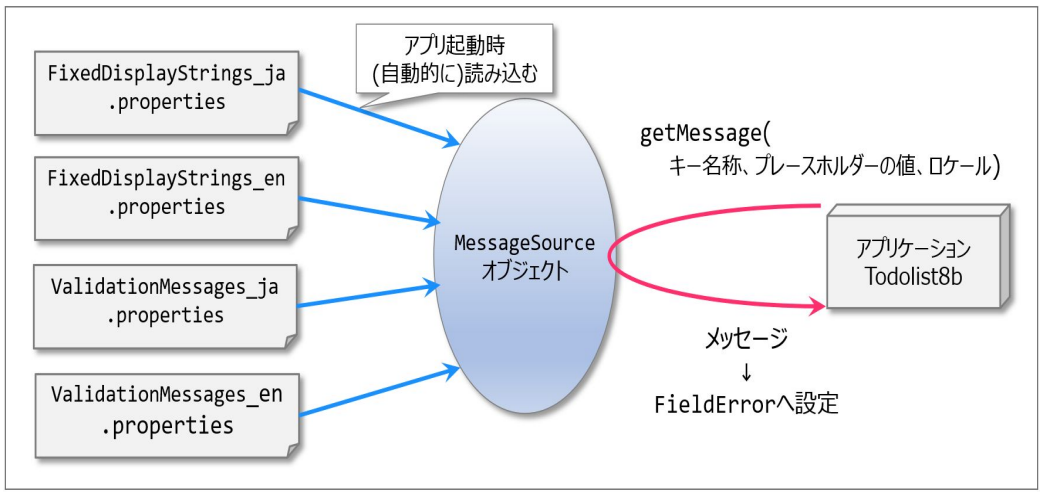
```
#件名
DoubleSpace.todoData.title=件名が全角スペースです

#期限
Past.todoData.deadline=期限を設定するときは今日以降にしてください
InvalidFormat.todoData.deadline=期限を設定するときはyyyy-mm-dd形式で入力してください
InvalidFormat.todoQuery.deadlineFrom=期限:開始を設定するときはyyyy-mm-dd形式で入力してください
InvalidFormat.todoQuery.deadlineTo=期限:終了を設定するときはyyyy-mm-dd形式で入力してください
```

【リスト12-13】src/main/resources/i18n/ValidationMessages_en.properties(追加分)

```
#Title
DoubleSpace.todoData.title=Title is a double space.

#Deadline
Past.todoData.deadline=Deadline is past.
InvalidFormat.todoData.deadline=Deadline in the yyyy-mm-dd format.
InvalidFormat.todoQuery.deadlineFrom=Deadline_from in the yyyy-mm-dd format.
InvalidFormat.todoQuery.deadlineTo=Deadline_to in the yyyy-mm-dd format.
```

(2)エラーメッセージの取得

プロパティファイルで定義された文字列をJavaプログラムから取得するには**MessageSourceオブジェクト(org.springframework.context.MessageSource)**を使います。このオブジェクトはアプリケーション起動時、プロパティファイルの内容を自動的に読み込み、その内容を保持しています。TodoServiceでエラーを見つけたら、ここからgetMessage()で対応するエラーメッセージを取得しFieldErrorへ設定します。図で表すと以下のようなイメージです。



【図12-6】MessageSourceによるメッセージの取得

以下はTodoServiceで「件名が全角スペースだけで構成されていたらエラーメッセージを設定する」という処理に関連する部分です。

【リスト12-14】com.example.todolist.service.TodoService#isValid()

```
package com.example.todolist.service;
            :
import java.util.Locale;
import lombok.AllArgsConstructor;
import org.springframework.context.MessageSource;
            :
@Service
@AllArgsConstructor //   ②
public class TodoService {
    private final MessageSource messageSource;   //   ①

    // Todoのチェック
        public boolean isValid(TodoData todoData,
                               BindingResult result,
                               boolean isCreate,
                               Locale locale) {   //   ④
        boolean ans = true;

        // 件名が全角スペースだけで構成されていたらエラー
```

```
                    :
                FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "title",
                    messageSource.getMessage("DoubleSpace.todoData.title",
null, locale));//③
                result.addError(fieldError);
                ans = false;
                    :
}
```

MessageSource型変数を定義し(①)、@AllArgsConstructor(②)でコンストラクターインジェクションします。

エラーメッセージを生成するFieldErrorのコンストラクターは3番目の引数を変更します(③)。

■変更前

```
        FieldError fieldError = new FieldError(
            result.getObjectName(),
            "title",
            "件名が全角スペースです");
```

■変更後

```
        FieldError fieldError = new FieldError(
            result.getObjectName(),
            "title",
            messageSource.getMessage("DoubleSpace.todoData.title", null,
locale));//③
```

getMessage()には3つの引数を渡しています。

■getMessageのメソッドシグネチャ

```
String getMessage(String code,    Object[] args,    Locale locale)
```

・**code**
取得するプロパティファイルのキー名称
ここでは"DoubleSpace.todoData.title"に対応するエラーメッセージを取得する。

・**args**
エラーメッセージのプレースホルダーを置換する値
たとえば以下のような定義があったとします。
InvalidLength.password=パスワードは{0}文字以上、{1}文字以下としてください

これを次のように呼び出すと
getMessage("InvalidLength.password", new Integer[]{8,16}, locale));
↓
"パスワードは8文字以上、16文字以下としてください"
となります。

ただしDoubleSpace.todoData.titleにはプレースホルダーが無いためnullを渡します。

・**locale**
どの言語のプロパティファイルを使用するか指定するLocale(java.util.Locale)オブジェクト
ハンドラメソッドにjava.util.Locale型引数を追加すると、これを受け取れる。

そこでコントローラーTodoListControllerのハンドラメソッドcreateTodo()にLocale型の引数を追加します。

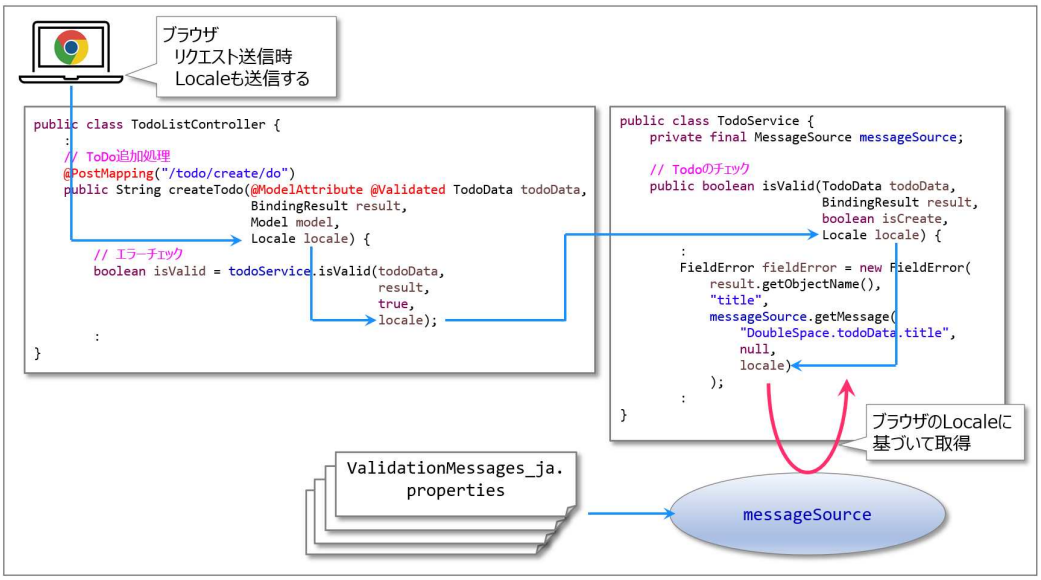
■Localeの取得

```
// ToDo追加処理
@PostMapping("/todo/create/do")
public String createTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result,
                         Model model,
                         Locale locale) {
    :
}
```

これをTodoService#isValid()に渡し、getMessage()の引数とします(④)。

```
    // Todoのチェック
    public boolean isValid(TodoData todoData,
                           BindingResult result,
                           boolean isCreate,
                           Locale locale) {    //    ④
```

このLocaleオブジェクトの受け渡しを図で表すと次のようになります。



【図12-7】ブラウザのLocaleに応じたメッセージの取得

よってTodoListControllerは以下の箇所を変更します。

【リスト12-15】com.example.todolist.controller.TodoListController.java(変更箇所)

```
package com.example.todolist.controller;

import java.util.Locale;
 :
@Controller
@RequiredArgsConstructor
public class TodoListController {
      :
    // ToDo追加処理
    @PostMapping("/todo/create/do")
```

```
    public String createTodo(@ModelAttribute @Validated TodoData todoData,
                             BindingResult result,
                             Model model,
                             Locale locale) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result, true, locale);
        :
    }

    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                             BindingResult result,
                             Model model,
                             Locale locale) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result, false, locale);
        :
    }

    // ToDo検索処理
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
                                  BindingResult result,
                                  @PageableDefault(page = 0, size = 5,
sort = "id")
                                  Pageable pageable,
                                  ModelAndView mv,
                                  Locale locale) {
        mv.setViewName("todoList");

        Page<Todo> todoPage = null;
        if (todoService.isValid(todoQuery, result, locale)) {
            // エラーがなければ検索
```

```
                :
    }

}
```

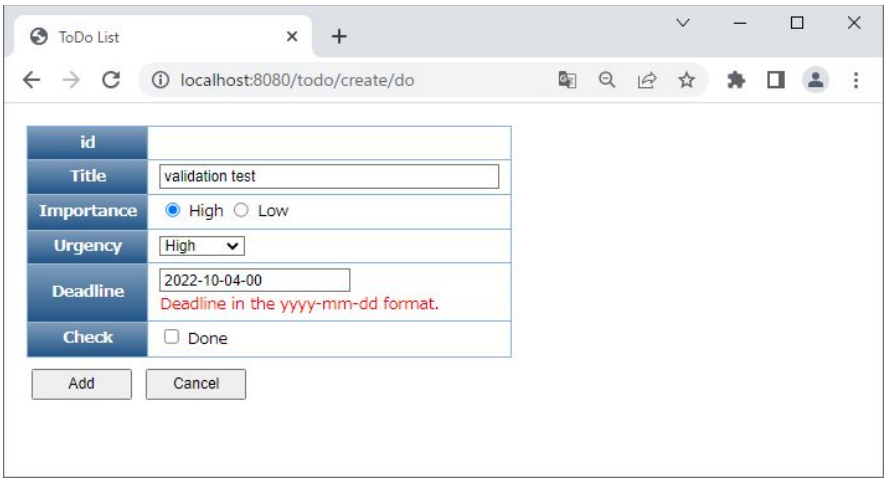
またTodoServiceの他のエラーメッセージも、受け取ったLocaleを使いgetMessage()で取得します。

【リスト12-16】com.example.todolist.service.TodoService.java(変更箇所)

```
package com.example.todolist.service;
            :
import java.util.Locale;
import org.springframework.context.MessageSource;
import lombok.AllArgsConstructor;
            :
@Service
@AllArgsConstructor
public class TodoService {
    private final MessageSource messageSource;

        :
        // yyyy-mm-dd形式チェック
            :
            FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "deadline",
                    messageSource.getMessage(
                        "InvalidFormat.todoData.deadline", null, locale));
        :
        // 過去日付チェックは新規登録の場合のみ
            :
        FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "deadline",
                    messageSource.getMessage(
```

```
                    "Past.todoData.deadline", null, locale));
            :
    }

    // 検索条件のチェック
    public boolean isValid(TodoQuery todoQuery,
                           BindingResult result,
                           Locale locale) {
        :
      // 期限:開始の形式をチェック
          :
          FieldError fieldError = new FieldError(
                result.getObjectName(),
                "deadlineFrom",
                messageSource.getMessage(
                    "InvalidFormat.todoQuery.deadlineFrom", null, locale));
          :
        // 期限:終了の形式をチェック
          :
            FieldError fieldError = new FieldError(
                  result.getObjectName(),
                  "deadlineTo",
                  messageSource.getMessage(
                        "InvalidFormat.todoQuery.deadlineTo", null, locale));
 :
}
```

以上で独自チェックの場合も言語表示に応じたメッセージが表示されるようになります。


ToDo List
localhost:8080/todo/create/do
id
Title
validation test
Importance
High
Low
Urgency
High
Deadline
2022-10-04-00
Deadline in the yyyy-mm-dd format.
Check
Done
Add
Cancel

13 操作メッセージ

プログラムで出力(表示)するメッセージは、エラーや異常に関するものだけではありません。操作が正常に完了したことを知らせるためにも使われます。ToDoアプリで言えば、ToDoを登録した後の一覧画面に「ToDoを作成しました」といったメッセージを表示できれば、ユーザーにとって親切かもしれません。

【図13-1】操作メッセージ(Information)

本章ではこういったユーザーの操作に対するメッセージを表示できるようにします。

13.1 操作メッセージの表示仕様

操作メッセージは次の 3 種類とします。

①Information(情報)

・前述のように処理が正常に終わったことを伝えるメッセージ。

・アイコン(i)を含め全体的に青色系の表示とする。

③Warning(警告)

・エラーではないが、ユーザーに何らかの注意を伝えるメッセージ。

・例えば、下図のように条件に一致するToDoが見つからなかった場合などが該当する。

・黄色系の表示とする。

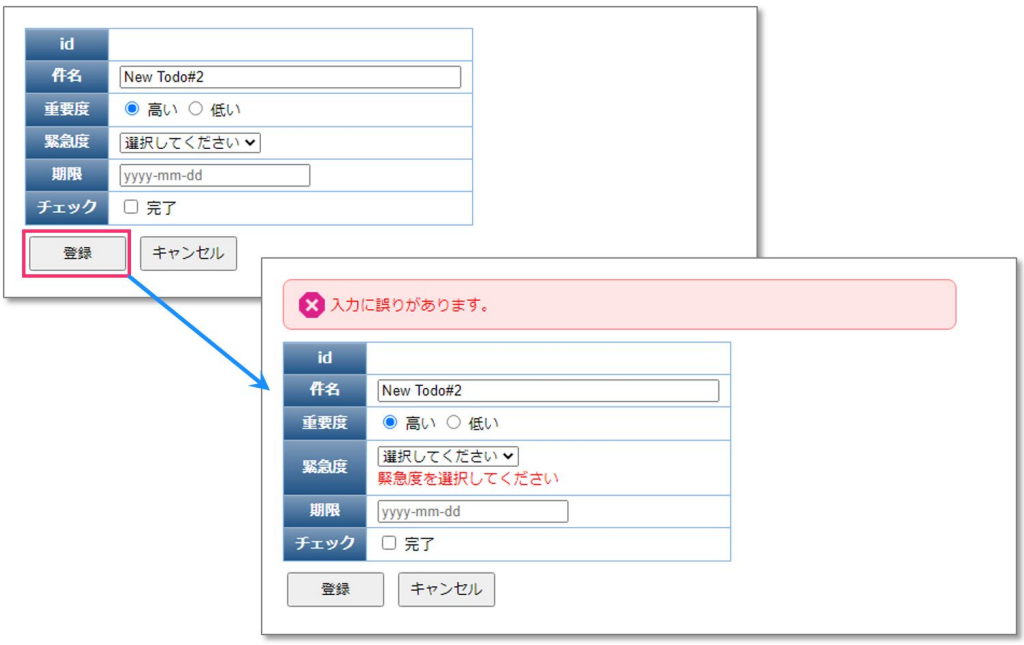
【図13-2】操作メッセージ(Warning)

③Error(エラー)

・ユーザーの操作に何らかの問題があったことを伝えるメッセージ。

・すでにエラーメッセージを表示するようになっているが、画面上部にエラー発生を知らせる表示を追加する。

・赤色系の表示とする。



【図13-3】操作メッセージ(Error)

作成するプロジェクトの仕様

プロジェクト名	Todolist9
作成ファイル	com.example.todolist.common.OpMsg.java src/main/resources/i18n/OperationMessages_en.properties src/main/resources/i18n/OperationMessages_ja.properties src/main/resources/static/icons/error_icon.png src/main/resources/static/icons/info_icon.png src/main/resources/static/icons/warning_icon.png src/main/resources/templates/fragments.html

```
Todolist9
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist9Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java(★)
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java
 └ src/main/resources
```

```
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties
    │    ├ FixedDisplayStrings_ja.properties
    │    ├ OperationMessages_en.properties(★)
    │    ├ OperationMessages_ja.properties(★)
    │    ├ ValidationMessages_en.properties
    │    └ ValidationMessages_ja.properties
    ├ static
    │    ├ css
    │    │   └ style.css(▲)
    │    └ icons(★)
    │        ├ error_icon.png(★)
    │        ├ info_icon.png(★)
    │        └ warning_icon.png(★)
    ├ templates
    │    ├ fragments.html(★)
    │    ├ todoForm.html(▲)
    │    └ todoList.html(▲)
    └application.properties(▲)
```

★：このプロジェクトで追加する
▲：前プロジェクトの内容を一部変更する

最初にメッセージの種別(Information/Warning/Error)とメッセージを保持するクラスOpMsgを作成します。

【リスト13-1】com.example.todolist.common.OpMsg.java

```java
package com.example.todolist.common;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class OpMsg {
```

```
    private String msgType;    // "I":Information, "W":Warning, "E":Error
    private String msgText;    //   メッセージ
}
```

msgTypeにメッセージの種別、msgTextにメッセージを格納します。種別はコメントのように"I", "W", "E"でInformation,Warning,Errorを表します。

> こういった種別は文字列リテラルではなくJavaのEnum型で表すこともできます。興味がある方は調べてみてください。

表示するメッセージは前章で説明したように言語別のプロパティファイルへ定義します。ここまで作成したFixedDisplayMessages, ValidationMessagesとは用途が異なるので、新たに**OperationMessages**として追加します。またキー名称は"msg." + メッセージの種別 + "." + メッセージ名 とします。

【リスト13-2】src/main/resources/i18n/OperationMessages_ja.properties

```
#
msg.i.todo_created=ToDoを作成しました。
msg.i.todo_updated=ToDoを更新しました。
msg.i.todo_deleted=ToDoを削除しました。
#
msg.w.todo_not_found=該当するToDoが見つかりませんでした。
#
msg.e.input_something_wrong=入力に誤りがあります。
```

【リスト13-3】src/main/resources/i18n/OperationMessages_en.properties

```
#
msg.i.todo_created=ToDo has been created.
msg.i.todo_updated=ToDo has been updated.
msg.i.todo_deleted=ToDo has been deleted.
#
msg.w.todo_not_found=The specified todo was not found.
#
msg.e.input_something_wrong=There is an error in the input.
```

プロパティファイルを追加したのでapplication.propertiesにもその情報を追加します。

【リスト13-4】src/main/resources/application.properties(追加箇所)

```
#
spring.messages.basename=i18n/FixedDisplayStrings, i18n/ValidationMessages,
i18n/OperationMessages
```

※レイアウトの都合上2行に分かれていますが、STSには改行せずに入力してください。

メッセージの準備は以上です。次はこのメッセージをセットするコントローラークラスを変更します。
まず前述【図13-3】ToDo追加時にエラーがあった場合です。

【リスト13-5】
com.example.todolist.controller.TodoListController#createTodo()

```
package com.example.todolist.controller;
    :
import org.springframework.context.MessageSource;
import com.example.todolist.common.OpMsg;
    :
public class TodoListController {
    :
  private final MessageSource messageSource;   // 追加
    :
  // ToDo追加処理
  @PostMapping("/todo/create/do")
  public String createTodo(@ModelAttribute @Validated TodoData todoData,
                           BindingResult result, Model model,
                           Locale locale) {
      // エラーチェック
      boolean isValid = todoService.isValid(todoData, result, true, locale);
      if (!result.hasErrors() && isValid) {
          :
      } else {
          // ①エラーあり -> エラーメッセージをセット
          String msg =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
```

```
            model.addAttribute("msg", new OpMsg("E", msg));
            return "todoForm";
        }
    }
 :
}
```

ここではToDo入力画面を再表示(return "todoForm")する前に、プロパティファイルから"msg.e.input_something_wrong"をキーにしてメッセージを取得し、OpMsgオブジェクトを作成します(①)。メッセージ種別はErrorなので"E"です。これをaddAttribute()で"msg"という名称でmodelにセットし、ToDo入力画面へ渡します。

ToDo入力画面には、このmsgから操作メッセージを組み立てる定義を追加します。

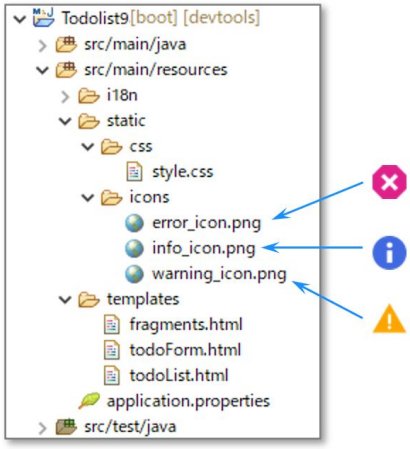
【リスト13-6】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
    :
<body>
  <!-- 操作メッセージエリア ここから　↓ ↓ ↓ -->
  <div th:if="${msg != null}">　<!-- ① -->
    <div th:if="${msg.msgType=='E'}">　<!-- ② -->
      <div class="err_msg">
        <img th:src="@{/icons/error_icon.png}" class="va"> <span
class="va"
          th:text="${msg.msgText}"></span>
      </div>
    </div>
    <div th:if="${msg.msgType=='W'}">　<!-- ③ -->
      <div class="warn_msg">
        <img th:src="@{/icons/warning_icon.png}" class="va"> <span
class="va"
          th:text="${msg.msgText}"></span>
      </div>
    </div>
    <div th:if="${msg.msgType=='I'}">　<!-- ④ -->
      <div class="info_msg">
        <img th:src="@{/icons/info_icon.png}" class="va"> <span
class="va"
          th:text="${msg.msgText}"></span>
      </div>
    </div>
  </div>
  <!-- 操作メッセージエリア ここまで　↑ ↑ ↑ -->
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
      :
  </form>
```

```
</body>
</html>
```

最初にコントローラークラスからmsgが渡されたかチェックします(①)。あればメッセージ種別で振り分けます(②③④)。ここではmsgTypeが"E"なので②の部分が作成されます。

メッセージはそのままspan要素で表示しますが、メッセージ種別はimg要素で対応するアイコンに変えます。アイコンの場所はth:srcのURL式で指定していますが、このように記述すると(CSSファイルの指定と同様に)src/main/resources/staticからの相対位置となります。よって各アイコンは下図の場所へ格納しておきます。



【図13-4】アイコンの配置

iconsフォルダは以下のように作成してください。

(1)サポートサイトからソースファイルをダウンロードし、解凍する。

(2)(1)で作成されたTodolist9¥src¥main¥resources¥static¥iconフォルダをエクスプローラーで右クリック > [コピー(C)]をクリックする。

(3)STSのパッケージ・エクスプローラーでTodolist9のsrc/main/resources/static右クリック > [貼り付け(V)]をクリックする。

(4)プロジェクトTodolist9を右クリック > [リフレッシュ(F)]をクリックする。

またstyle.cssには、これらのclass定義を追加します。

【リスト13-7】src/main/resources/static/css/style.css(追加分)

```
.va {
vertical-align: middle;
}
```

```
/* information message */
.info_msg {
margin-bottom: 0.75em; /*下余白*/
padding: 0.75em 1em; /*文字余白*/
width: 50%; /*幅*/
color: blue; /*文字色*/
background-color: #dfefff; /*背景色*/
border: 1px solid #379bff; /*線の太さ・色*/
border-radius: 8px; /*角の丸み*/
}

/* warning message */
.warn_msg {
margin-bottom: 0.75em; /*下余白*/
padding: 0.75em 1em; /*文字余白*/
width: 50%; /*幅*/
color: #bd3e00; /*文字色*/
background-color: #ffffc6; /*背景色*/
border: 1px solid #ff8040; /*線の太さ・色*/
border-radius: 8px; /*角の丸み*/
}

/* error message */
.err_msg {
margin-bottom: 0.75em; /*下余白*/
padding: 0.75em 1em; /*文字余白*/
width: 50%; /*幅*/
color: red; /*文字色*/
background-color: #ffe6e6; /*背景色*/
border: 1px solid #ff8080; /*線の太さ・色*/
border-radius: 8px; /*角の丸み*/
}
```

これでToDo入力画面に操作メッセージを(ToDo追加時にエラーがあった場合だけですが)表示できるようになります。

13.2 フラグメント(th:fragment/th:insert)

次はToDo一覧画面にも操作メッセージを表示します。
【リスト13-6】と同様の記述を追加してもいいのが、これだとメッセージ表示方法を変更する場合、両方のhtmlファイルを直すことになるため、あまり良い方法と言えません。Thymeleafではこのように画面間で共通に使いたい部分を**フラグメント**(fragment;断片)として定義すると、簡単に共有できます。

フラグメントにする部分は**th:fragment**属性で名前を付けます。ここでは【リスト13-6】todoForm.htmlの操作メッセージエリアをdiv要素で囲み、名前を"msg_area"とします。

```
<!-- 操作メッセージエリア -->
<div th:fragment="msg_area">
  <div th:if="${msg != null}"><!-- ※1 -->
      :
  </div>
</div>
```

一方todoList.htmlは以下のようにします。

```
<!DOCTYPE html>
    :
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="~{todoForm :: msg_area}"></div><!-- ※2 -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
    <!-- 検索条件入力エリア -->
    :
```

th:replace属性には、この要素を置き換えるフラグメントを「**~{ビュー名::th:fragmentで付けた名前}**」という形式で指定します(ビュー名は.htmlが付かない)。これでToDo一覧画面を表示するとき、※2の部分は「ビューtodoFormのフラグメントmsg_area」、つまり※1のdiv要素以下の内容になります。

これでToDo一覧画面にもToDo入力画面と同じ操作メッセージエリアを定義できたわけですが、一般的にフラグメントは元の画面から独立させることが多いようです。そこでこれに沿ったやり方に変えてみます。

まずfragments.htmlというファイルを作成し、todoForm.htmlから操作メッセージエリアの定義を移します。

【リスト13-8】src/main/resources/templates/fragments.html(一部抜粋)

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
</head>
<body>
  <!-- 操作メッセージエリアのフラグメント ここから ↓ ↓ ↓ -->
  <div th:fragment="msg_area">
    <div th:if="${msg != null}">
      <div th:if="${msg.msgType=='E'}">
        :
      </div>
      <div th:if="${msg.msgType=='W'}">
        :
      </div>
      <div th:if="${msg.msgType=='I'}">
        :
      </div>
    </div>
  </div>
  <!-- 操作メッセージエリアのフラグメント ここまで ↑ ↑ ↑ -->
</body>
</html>
```

フラグメントを使う側のtodoList.html/todoForm.htmlは以下のように変更します。これも「**ビュー名::th:fragmentで付けた名前**」という形式になっています。

【リスト13-9】src/main/resources/templates/todoList.html(変更箇所)

```
<!DOCTYPE html>
```

```
    :
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="~{fragments :: msg_area}"></div>
  <!-- 検索条件入力エリア -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
    :
```

【リスト13-10】src/main/resources/templates/todoForm.html(変更箇所)

```
<!DOCTYPE html>
    :
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="~{fragments :: msg_area}"></div>
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
    :
```

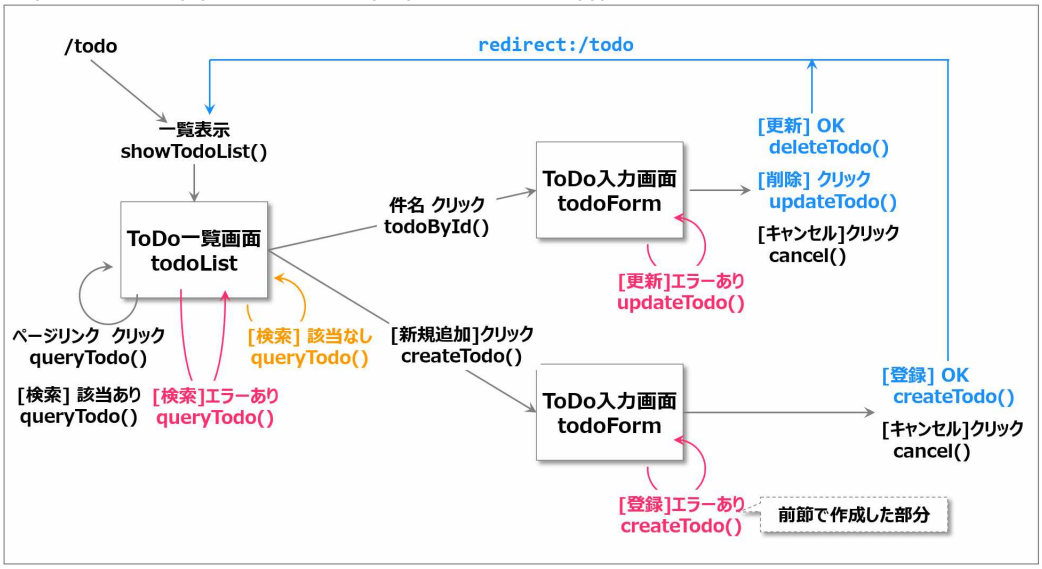
これで同じフラグメントを2つの画面(ビュー)で共有できます。

> th:replaceはフラグメントで置き換えましたが、他に子要素として「挿入」する**th:insert**属性も用意されています。興味がある方は以下のサイトなどを調べてみてください。
>
> Tutorial: Using Thymeleaf (ja) > 8 テンプレートレイアウト
>
> https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf_ja.html#テンプレートレイアウト

13.3 RedirectAttributes

操作メッセージの表示方法がわかったので、他のパターンも追加します。

下図はToDoアプリの画面と操作をまとめたものですが、色のついた部分は操作メッセージを表しています(赤：エラー(E)、黄：警告(W)、青：情報(I))。



【図13-5】ToDoアプリで操作メッセージを表示するタイミング

例えば右下の「[登録]エラーあり」は、前節で作成した『[登録]ボタンをクリックしたときエラーがあれば操作メッセージをToDo入力画面へ表示する』ことを表しています。つまり矢印の先がメッセージを表示する画面です。表にすると以下のようになります。

【表13-1】ToDoアプリの操作メッセージ表示仕様

#	発生元/タイミング		表示先	表示内容	
①	ToDo一覧画面 todoList	[検索]エラーあり queryTodo()	ToDo一覧画面 todoList	エラー(E)	msg.e.input_something_wrong
②		[検索]該当なし queryTodo()		警告(W)	msg.w.todo_not_found
③	ToDo入力画面	[登録]エラーあり	ToDo入力画面	エラー	msg.e.input_something_wrong

	todoForm	createTodo()	todoForm	(E)	
④		[更新]エラーあり updateTodo()		エラー(E)	msg.e.input_something_wrong
⑤		[登録]OK createTodo()	ToDo一覧画面 todoList	情報(I)	msg.i.todo_created
⑥		[更新]OK updateTodo()		情報(I)	msg.i.todo_updated
⑦		[削除]OK deleteTodo()		情報(I)	msg.i.todo_deleted

この表に沿って操作メッセージを表示できるようにします。

①[検索]エラーあり、②[検索]該当なし

どちらもToDo一覧画面からToDo一覧画面、つまり自画面への遷移で、ともにqueryTodo()で処理しています。ここに以下のようなコードを追加します。

【リスト13-11】

com.example.todolist.controller.TodoListController#queryTodo()

```
    // ToDo検索処理
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
BindingResult result,
                                  @PageableDefault(page = 0, size = 5, sort
= "id")
                                  Pageable pageable, ModelAndView mv,
Locale locale) {
        mv.setViewName("todoList");

        Page<Todo> todoPage = null;
        if (todoService.isValid(todoQuery, result, locale)) {
```

```
            // エラーがなければ検索
            todoPage = todoDaoImpl.findByCriteria(todoQuery, pageable);
                :
            // ②該当なかったらメッセージを表示
            if (todoPage.getContent().size() == 0) {
                String msg =
messageSource.getMessage("msg.w.todo_not_found", null, locale);
                mv.addObject("msg", new OpMsg("W", msg));
            }
        } else {
            // ①検索条件エラーあり -> エラーメッセージをセット
            String msg =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
            mv.addObject("msg", new OpMsg("E", msg));

            mv.addObject("todoPage", null);
            mv.addObject("todoList", null);
        }

        return mv;
    }
```

①の「検索条件エラー」は、前節の[登録]でエラーがあった場合と同じ考え方です。
②は検索結果、つまりtodoPage.getContent()の要素数が0であれば「該当なし」と判断し、メッセージをセットします。

③[登録]エラーあり

前節で解説したので省略します。

④[更新]エラーあり

考え方は①③と同じです。

【リスト13-12】

com.example.todolist.controller.TodoListController#updateTodo() (step1)

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                                        BindingResult result, Model model, Locale locale)
{
        // エラーチェック
          boolean isValid = todoService.isValid(todoData, result, false, locale);
          if (!result.hasErrors() && isValid) {
                    // エラーなし -> 更新
                    Todo todo = todoData.toEntity();
                    todoRepository.saveAndFlush(todo);
                    return "redirect:/todo";
          } else {
                    // ④エラーあり -> エラーメッセージをセット
                    String msg =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
                    model.addAttribute("msg", new OpMsg("E", msg));
                    return "todoForm";
          }
    }
```

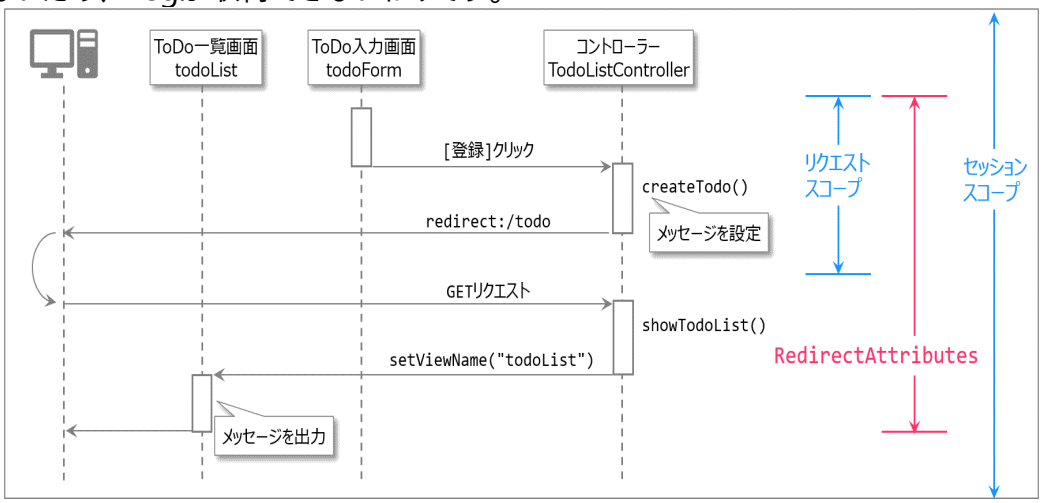
⑥[更新]OK

表の順番と異なりますが、【リスト13-12】に続けて⑥の更新成功メッセージを追加します。以下のコードは、ここまでと同じようにメッセージをModelオブジェクトへセットしていますが、これではTodo一覧画面に**メッセージが表示されません。**

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                                        BindingResult result, Model model,
```

```
                                        Locale locale) {
        // エラーチェック
          boolean isValid = todoService.isValid(todoData, result, false, locale);
          if (!result.hasErrors() && isValid) {
                    // エラーなし -> 更新
                    Todo todo = todoData.toEntity();
                    todoRepository.saveAndFlush(todo);
                    // 更新完了メッセージをセットしてリダイレクト　→　このメッセージは表示されない(NG)
                    String msg = messageSource.getMessage("msg.i.todo_updated", null, locale);
                    model.addAttribute("msg", new OpMsg("I", msg));

                    return "redirect:/todo";
          } else {
                // ④エラーあり -> エラーメッセージをセット
           :
          }
    }
```

これはオブジェクトの「**スコープ(scope;有効範囲)**」が影響しています。上記コードでメッセージをセットしたmodelは、リクエストがクライアント～サーバーを一往復する間だけ存在します。この範囲を「**リクエストスコープ**」と言います。しかしリダイレクト("redirect:/todo")は、下図のようにクライアント～サーバー間を(自動的に)二往復しています。このためtodoListではmodelがスコープ外で存在しないため、msgが取得できないわけです。



【図13-6】リクエストスコープではリダイレクト先にデータを渡せない

解決策としては以下のようなものがあります。

1)Session(セッション)にメッセージデータを格納する

セッションはサーバーへ最初にアクセスしたとき作成され、リクエスト間でデータを共有する仕組みを提供します(**セッションスコープ**)。しかしセッションに格納したデータはタイムアウトするまで存在します。このため操作メッセージを表示しないときは、セッションからデータを明示的に削除する(removeAttribute())、あるいは空のデータで上書きするような処理が必要です。

2)RedirectAttributesを使う

Spring Bootにはリダイレクト先へパラメータ(データ)を渡す仕組みとして**RedirectAttributes**というものが用意されています。スコープは上図のようにセッションスコープとリクエストスコープの間というイメージです(「**フラッシュスコープ**」と呼ばれることもあります)。

ToDoアプリでは2)の方法を使います。

【リスト13-13】
com.example.todolist.controller.TodoListController#updateTodo() (step2)

```
        :
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
```

```
    :
// ToDo更新処理
@PostMapping("/todo/update")
public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result, Model model,
                         RedirectAttributes redirectAttributes,    // ※ 1
                         Locale locale) {
    // エラーチェック
    boolean isValid = todoService.isValid(todoData, result, false, locale);
    if (!result.hasErrors() && isValid) {
        // エラーなし -> 更新
        Todo todo = todoData.toEntity();
        todoRepository.saveAndFlush(todo);

        // ⑥更新完了メッセージをセットしてリダイレクト
        String msg = messageSource.getMessage("msg.i.todo_updated", null,
locale);
        redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));    //
※2
        return "redirect:/todo";

    } else {
        // ④エラーあり -> エラーメッセージをセット
         :
    }
}
```

RedirectAttributesを使うときは、ハンドラーメソッドに引数を追加します(※１)。これにaddFlashAttribute()でメッセージデータmsgを追加(※2)すると、リダイレクト先のtodoListまで有効になります。

todoListは"msg"という名前のデータを取得して操作メッセージを作成するので、ここで設定した内容が表示されるわけです。

⑤[登録]OK

これも(6)と同じパターンです。RedirectAttributes型の引数を追加し、メッセージデータをセットします。

【リスト13-14】

com.example.todolist.controller.TodoListController#createTodo()

```
    // ToDo追加処理
    @PostMapping("/todo/create/do")
    public String createTodo(@ModelAttribute @Validated TodoData todoData,
                             BindingResult result,    Model model,
                             RedirectAttributes redirectAttributes, Locale
locale) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result, true, locale);
        if (!result.hasErrors() && isValid) {
            // エラーなし -> 追加
            Todo todo = todoData.toEntity();
            todoRepository.saveAndFlush(todo);

            // ⑤追加完了メッセージをセットしてリダイレクト
            String msg = messageSource.getMessage("msg.i.todo_created",
null, locale);
            redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
            return "redirect:/todo";

        } else {
            // ③エラーあり -> エラーメッセージをセット
          :
        }
    }
```

⑦[削除]OK

⑤⑥と同じくRedirectAttributesを使います。

【リスト13-15】

com.example.todolist.controller.TodoListController#deleteTodo()

```
    // ToDo削除処理
    @PostMapping("/todo/delete")
    public String deleteTodo(@ModelAttribute TodoData todoData,
                             RedirectAttributes redirectAttributes,
                             Locale locale) {
        // 削除
        todoRepository.deleteById(todoData.getId());

        // ⑦削除完了メッセージをセットしてリダイレクト
        String msg = messageSource.getMessage("msg.i.todo_deleted", null,
locale);
        redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
        return "redirect:/todo";
    }
```

これでユーザーの操作に応じたメッセージを表示できるようになりました。

次章からは実務システムでも利用頻度が高い、関連エンティティの扱いについて解説していきます。

14. エンティティの関連

ここまでToDoアプリが使用するテーブル(エンティティ)はtodoテーブル 1 つだけでした。しかしほとんどの業務システムは、複数のテーブルを使うようになっています(テーブル数が100を超えるシステムも珍しくありません)。

そこでToDoアプリにもタスク(task)を表す**taskテーブル**を追加します。ToDoが「やるべきこと」なら、タスクはその「具体的な内容」です。例えばToDoが「新機能Xのリリース」なら、タスクは(1)新機能Xの要件確認、(2)設計、(3)開発・テスト、(4)リリース作業、といった感じになるでしょう。

本章では複数のテーブルを扱う際必要な「**関連**」「**多重度**」という考え方について説明した後、ToDoアプリにtaskテーブルを追加し、アクセスできるようプログラムを変更していきます。

14.1 関連

上述したように業務システムはテーブルを複数持つのが一般的です。そしてそれらテーブルの間には「関係」あるいは「関連」というものが存在します。この考え方はプログラムを作るうえで非常に重要なので、テーブル設計に近い部分から掘り下げて説明します。

例えば、あるシステムで使うため以下の「従業員名簿」をテーブルにしたいのですが、どのような形にすればよいでしょう？

■従業員名簿

従業員番号	従業員名	部署コード	部署名
3425	安藤 健児	200	営業部
3429	中川 覚	400	設計部
3675	土田 理香	800	経理部
3796	岡崎 正征	400	設計部
5293	青山 彰揮	400	設計部
5784	富田 俊彰	400	設計部
7538	高島 雅栄	900	総務部
8934	小谷 さゆみ	400	設計部
9587	望月 浩治	800	経理部
9657	神田 玲子	400	設計部

【図14-1】従業員名簿

もっとも簡単なのは「そのままテーブルにする」です。しかし問題になりそうなことがあります。

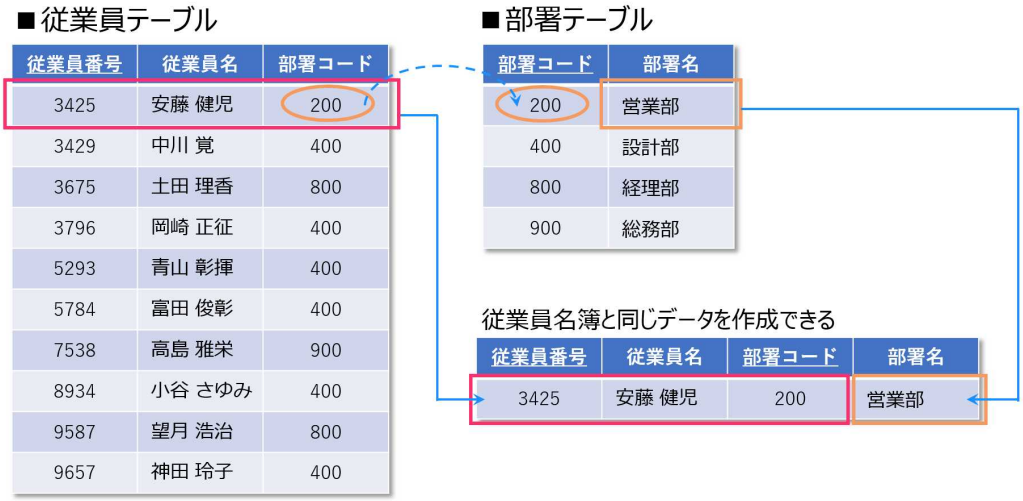
1)更新時不整合を引き起こす可能性がある

「設計部」を「開発部」へ名称変更する組織改正があったとします。従業員名簿の形では、５人の部署名を(漏れなく)修正しなければなりません。変更箇所は少ない方が、ミス(修正もれ)を防ぎやすいものです。

2)従業員が所属していない部署を載せられない

最近「企画室」という部署が発足しました。しかしまだ誰も配属されていません。従業員名簿の形では、こういった「従業員が所属していない部署」を表せません。

そこで従業員名簿を「従業員」、「部署」に分けます。そして従業員テーブルには部署コードを残します。



【図14-2】従業員テーブルと部署テーブルに分割

こうすると上記1), 2)を解決できます。

1)部署名の変更は、部署テーブルの１箇所だけで済む。

2)従業員が所属していない部署も部署テーブルに登録できる。

もう１つ重要なのは、このようにテーブルを分けても元の従業員名簿と同じ形のデータを作成できることです(テーブルの「**結合**」)。つまり「従業員テーブル」と「部署テーブル」は、従業員名簿を表すための「**関係**」を持っています(元が1つなのですから、関係があって当然という見方もできます)。

本書ではPostgreSQLというRDB(リレーショナル・データベース)を利用していますが、RDBのRは「relational(関係)」の頭文字ということからも、関係はRDBの本質的な部分と言えます。

> ここでは従業員名簿を 2 つのテーブルへ分けましたが、こういった作業を「**テーブルの正規化**」と言います。正規化の考え方で言えば、従業員名簿は「第二正規形」と呼ばれる形をしています。この形には本文中で触れたような懸念があります。そこで従業員テーブル、部署テーブルに分割しましたが、これらは「**第三正規形**」と呼ばれる形をしています。システム設計では、ここまで分解することを 1 つのゴールとします。
> **業務システムを設計する上で、正規化は非常に重要な考え方**です。しかし本書の枠を超えていますので説明は割愛します。興味がある方は、是非調べてみてください。

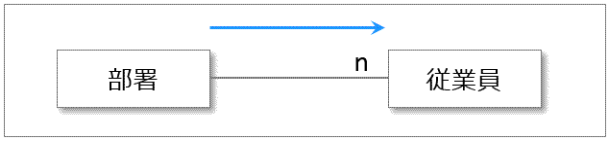
14.2 多重度

テーブルの関係には様々なタイプがあります。それを表すのが「**多重度(cardinality;カーディナリティ)**」です。これはテーブルのレコード同士の**数量的な関係**に着目した考え方です。前節の「部署」と「従業員」で言えば

・「部署」には「従業員」が何人配属されているか
・「従業員」はいくつの「部署」に所属しているか

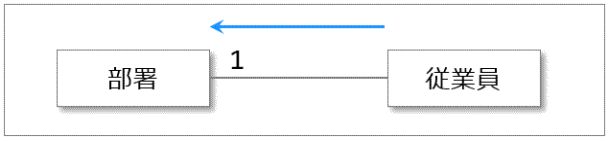
を表現したものです。このように多重度は「部署→従業員」「部署←従業員」の双方向で考えます。

まず部署から見て行きます。通常 **1 つ**の部署には、従業員が**複数人**配属されています。また前述の「企画室」のように、従業員のいない部署なら0人です。そこで下図のように、従業員側に「**n**」と書いておきます。これは「**0人以上**」という意味です。



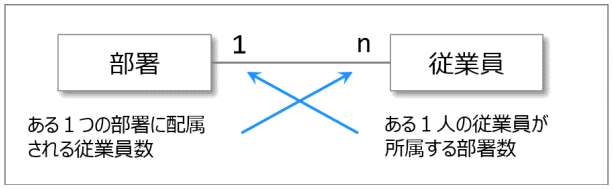
【図14-3】部署から見た従業員

次に従業員から見ます。**1 人**の従業員が所属する部署が **1 つ**なら、部署側を「1」とします。



【図14-4】従業員から見た部署

2 つを合体します。この場合(部署側を基準に考えると)多重度は**1：n**となります。



【図14-5】部署と従業員の多重度(1:n)

ではToDoとタスクはどのような多重度になるでしょう？これも「ToDo→タスク」「ToDo←タスク」の双方向で考えます。

■ToDo→タスク

・１つのToDoは複数のタスクを持つ可能性があります。

・しかし分割するまでもない単純なToDoはタスクが無いでしょう。この場合は 0 です。

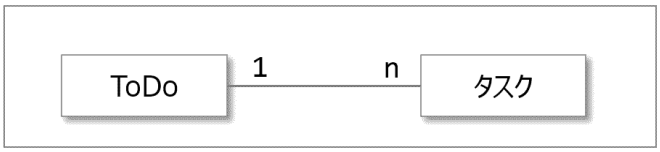
→n(0個以上)

■ToDo←タスク

・「タスクはToDoを構成するもの」なので、関係するToDoが最低１つあるはずです。

・「ToDoで共通なタスク」を作ればnになりますが、ToDoアプリでは考えないことにします。

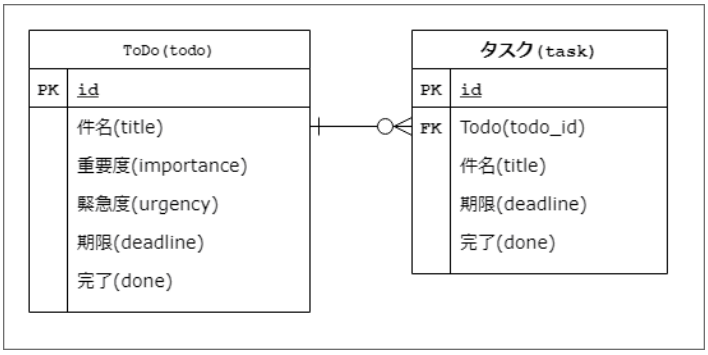
→１



【図14-6】Todoとタスクの多重度

よってToDoとタスクの多重度は**1:n**です。この関係が姿を変え、プログラム中に何度も現れるので覚えておいてください。

ここで説明した内容は、RDBを使ったシステムの設計に欠かせない**ER図(entity relationship diagram)**につながります。ER図の書き方はいくつかありますが、そのうちの1つである**IE記法**で表すと以下のようになります。



【図14-7】本章で使用するテーブルのER図(IE記法)

> 従業員が複数の部署に所属できる場合はどうなるでしょうか？例えば
> ・「設計部」「新製品開発プロジェクト」の２つに所属している人がいる

> ・経理部長は総務部長も兼務している
> といった場合です。こういう場合は、部署と従業員の多重度は**n:n**になります。こういったケースでは本書で解説している方法ではうまく扱えません。**発展編**で解説していますので興味がある方は御覧ください。

14.3 Taskテーブルの追加

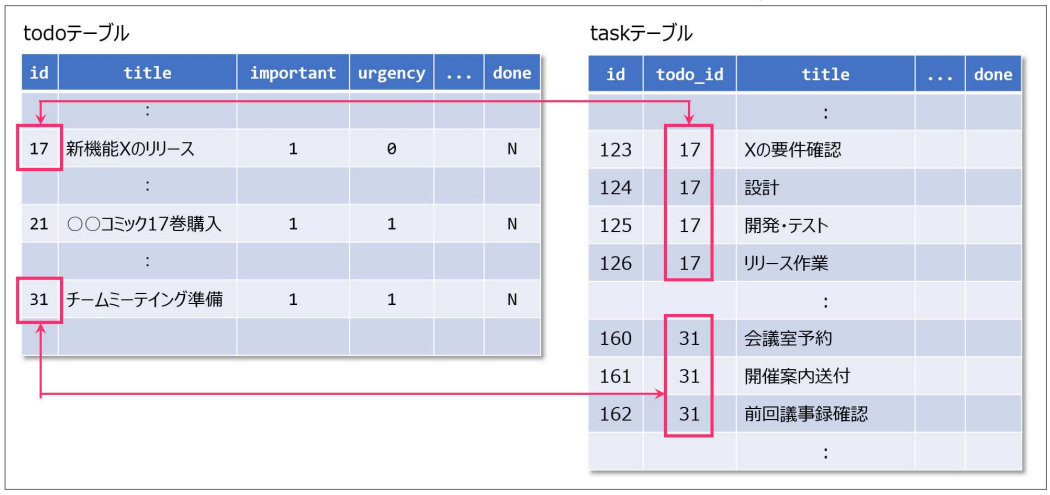
ToDoアプリで使用するtaskテーブルは以下のような形とします。

【表14-1】taskテーブルの形式

列名	内容	データ型	制約	備考
id	1～	SERIAL	PRIMARY KEY	連番(自動採番)
todo_id	対応するToDoのid	INTEGER		
title	件名	TEXT		
deadline	期限	DATE		
done	完了	TEXT		'Y':完了, 'N':未完了

ポイントはtodo_id列です。ここには「このタスクはどのToDoのものか？」を表すためtodoテーブルのid列(以下、todo.id)の値を設定します。

例えばtodoテーブルとtaskテーブルが以下のような状態だとします。



【図14-8】todoテーブルとtaskテーブルの関連付け

taskテーブルのid(=task.id)が123～126のレコードは、todo_idが17です。これはToDo「新機能Xのリリース」のタスクであることを表しています。同様にToDo「チームミーティング準備」のタスクは、task.idが160～162のレコードです。一方、ToDo「○○コミック17巻購入」のid(=21)に対応するレコードはtaskテーブルにありません。これはタスクがないToDoです。

このように前節で説明した**テーブルの関係は、テーブルのキー値を保持することで表します。**

> todoテーブルとtaskテーブルは多重度が1:nです。そしてn側(=taskテーブル)に1側(=todoテーブル)のキー(id)を格納する列(todo_id)があります。【図14-2】の従業員テーブル、部署テーブルでもn側(=従業員テーブル)が、1側(=部署テーブル)のキーを持ちます。こういった形にするのは、**テーブル正規化のセオリーの１つ**です。

> **外部キー(Foreign key)**を定義することで、テーブル間に制約を持たせることもできます。興味がある方は調べてみてください。

では実際にtaskテーブルを作成します。taskとの対応がわかりやすいデータを登録するため、todoテーブルも再作成します。以下のSQLファイルを前書6章の手順を参考にpsqlから実行してください。

【リスト14-1】create_todo_task.sql

```
DROP TABLE todo;
CREATE TABLE todo
(
   id                SERIAL PRIMARY KEY,
   title             TEXT,
   importance    INTEGER,
   urgency         INTEGER,
   deadline       DATE,
   done             TEXT
);

INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-1', 0, 0, '2021-10-01', 'N');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-2', 0, 1, '2021-10-02', 'Y');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-3', 1, 0, '2021-10-03', 'N');
INSERT INTO todo(title, importance, urgency, deadline, done)
VALUES('todo-4', 1, 1, '2021-10-04', 'Y');

DROP TABLE task;
```

```
CREATE TABLE task
(
   id                SERIAL PRIMARY KEY,
   todo_id           INTEGER,
   title             TEXT,
   deadline          DATE,
   done              TEXT
);

INSERT INTO task(todo_id,title,deadline,done) VALUES(1, 'task1-1','2021-09-30','N');
INSERT INTO task(todo_id,title,deadline,done) VALUES(2, 'task2-1',null,'N');
INSERT INTO task(todo_id,title,deadline,done) VALUES(2, 'task2-2',null,'Y');
INSERT INTO task(todo_id,title,deadline,done) VALUES(3, 'task3-1','2021-10-02','N');
INSERT INTO task(todo_id,title,deadline,done) VALUES(3, 'task3-2',null,'N');
INSERT INTO task(todo_id,title,deadline,done) VALUES(3, 'task3-3',null,'N');
```

> DROP TABLEは「**テーブルを削除**」する命令です(DELETEは「テーブルのレコード」を削除」)。上記SQLを初めて実行するとき、taskテーブルはまだ存在しないためDROP TABLE taskでエラーになりますが問題ありません。

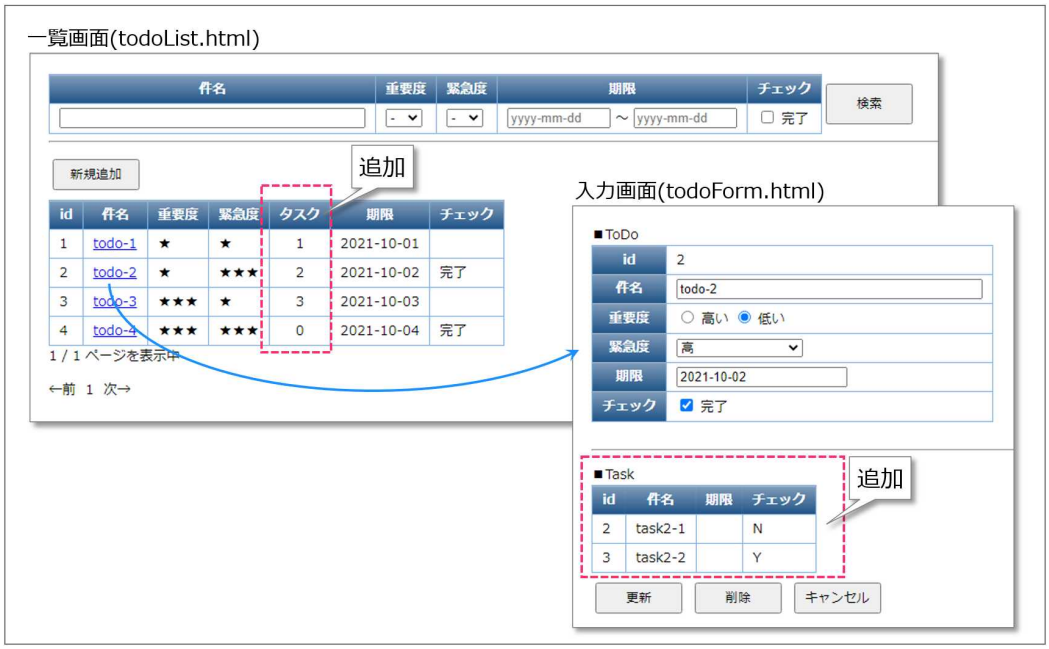
これでtodo, taskには以下のようなデータが登録されます。

【表14-2】todo/taskサンプルデータ

todo		task		
id	title	id	todo_id	title
1	todo-1	1	1	task1-1
2	todo-2	2	2	task2-1
		3	2	task2-2
3	todo-3	4	3	task3-1
		5	3	task3-2
		6	3	task3-3
4	todo-4			

todo-1,2,3はタスクをそれぞれ1,2,3個持ちますが、todo-4にはありません。

ではToDoアプリからtaskテーブルもアクセスできるようにします。この後本章では、一覧画面に**各ToDoのタスク数**を表示できるようにします。またToDo入力画面下部には**タスク一覧**を追加します。



【図14-9】タスク数とタスクの一覧表示

作成するプロジェクトの仕様

プロジェクト名	Todolist10
作成ファイル	com.example.todolist.entity.Task.java

```
Todolist10
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist10Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ Task.java(★)
 │    │    └ Todo.java(▲)
 │    ├ com.example.todolist.form
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java
 └ src/main/resources
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties(▲)
    │    ├ FixedDisplayStrings_ja.properties(▲)
    │    ├ OperationMessages_en.properties
```

```
    │    ├ OperationMessages_ja.properties
    │    ├ ValidationMessages_en.properties
    │    └ ValidationMessages_ja.properties
    :
    ├ templates
    │    ├ fragments.html
    │    ├ todoForm.html(▲)
    │    └ todoList.html(▲)
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

14.4 関連を持つエンティティの定義

まずtaskテーブルを追加したので対応するエンティティTaskを定義します。

【リスト14-2】com.example.todolist.entity.Task.java

```
package com.example.todolist.entity;

import java.sql.Date;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.Data;

@Entity
@Table(name = "task")
@Data
public class Task {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Integer id;

	@ManyToOne	//	①
	@JoinColumn(name = "todo_id")	//②
	private Todo todo;	//③

	@Column(name = "title")
	private String title;
```

```
    @Column(name = "deadline")
    private Date deadline;

    @Column(name = "done")
    private String done;

}
```

基本的には前書6章で説明したtodoテーブルに対するエンティティTodoと同様に、taskテーブルの列に対応するプロパティを定義します。ただしtodoプロパティは例外です(taskテーブルにtodoという列は無い)。これはtaskエンティティ(テーブル)からtodoエンティティ(テーブル)を見たときの関係を表しています。

①**@ManyToOne**

todoとtaskの多重度は**1:n**です。

@ManyToOneアノテーションは、taskが**n側**であることを表します。

②**@JoinColumn(name = "todo_id")**

taskに関連するtodoは、taskテーブル上のtodo_id列の値で決まることを表します(【図14-8】)。

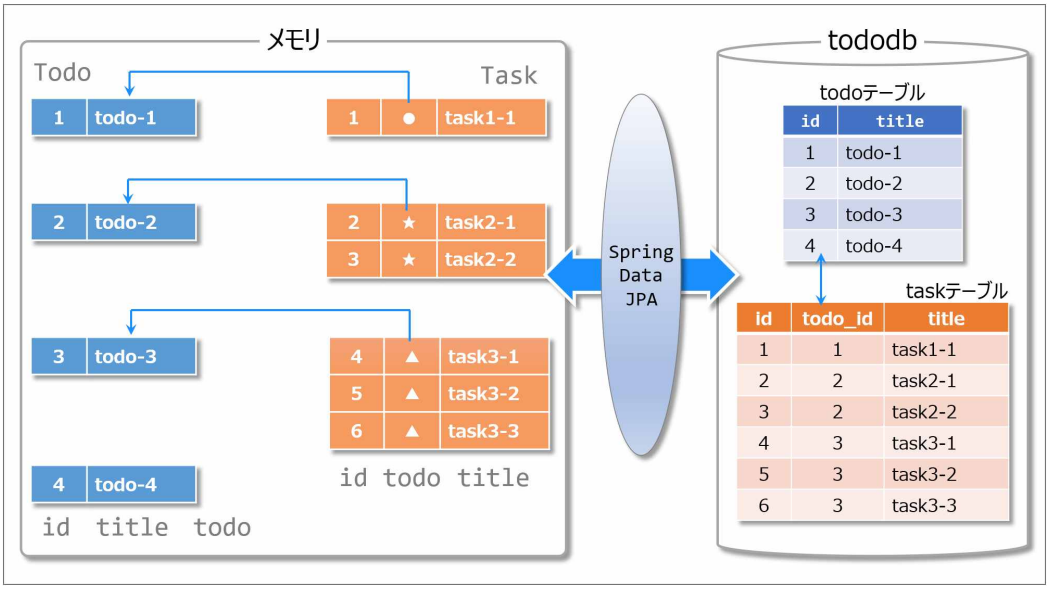
"Join"は「結合」と言う意味です。

③**private Todo toto;**

taskに関連するtodoを表します。

taskから見ると関連するtodoは 1 個なので、Todo型の変数とします。

ここは少々理解しにくい部分ですが、図にすると次のようなイメージになります。



【図14-10】エンティティの関係(task→todo)

図右側のtododbは【表14-2】のデータが格納されている状態を表しています。task.todo_idが関連するtodo.idの値を持っています。

左側のメモリは、これらのデータを検索した結果の状態です。テーブルの内容がTodoオブジェクト、Taskオブジェクトになっていますが、task.todo_idの値で表していた関連は、Todoオブジェクトへの参照に変わります(●★▲は前書5章の補足で説明したメモリ上の格納場所を示す特別な値の意)。このようにSpring Boot(Spring Data JPA)では、**参照によってエンティティ(テーブル)の関連を表します。**

しかしこれだけではToDoから関連するタスクをアクセスできません。そこでTodoクラスにも定義を追加します。

【リスト14-3】com.example.todolist.entity.Todo.java(一部抜粋)

```
package com.example.todolist.entity;

import java.util.List;
import jakarta.persistence.CascadeType;
import jakarta.persistence.OneToMany;
import lombok.ToString;
      :
@Entity
@Table(name = "todo")
@Data
//   ③追加しないとtoString()で循環参照が起こりStackOverFlowを引き起こす
@ToString(exclude="taskList")
public class Todo {
      :
      @Column(name = "done")
      private String done;

      //   taskListプロパティを追加
      @OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)   //   ①
      private List<Task> taskList;   //   ②
}
```

追加したのは最後のtaskListプロパティです。

①@OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)

・@OneToManyアノテーションは、Todoが**1：nの1側**であることを表します。

・mappedByには、このTodoを参照するn側のプロパティ名(【リスト14-2】③todo)を指定します。

もしTaskがプロパティxyzでTodoを参照するなら、mappedByはxyzとします。

```
      @ManyToOne
      @JoinColumn(name = "todo_id")
      private Todo xyz;
```

```
        ↓
    @OneToMany(mappedBy = "xyz", cascade = CascadeType.ALL)
```

・cascadeは、このTodoに対する処理(登録、更新、削除など)を参照しているTaskにも適用するかどうかの指定です。CascadeType.ALLは、すべての処理をTaskにも適用することを表します。(後述)

②private List<Task> taskList;

・関連するTaskを表すプロパティです。

TodoはTaskを0個以上持つのでList<Task>とします。

→関連するTaskが無いの場合、taskList.size()=0となる(nullではない)。

このtaskListに対応する列はtodoテーブルに無いので**@Columnは指定しません。**

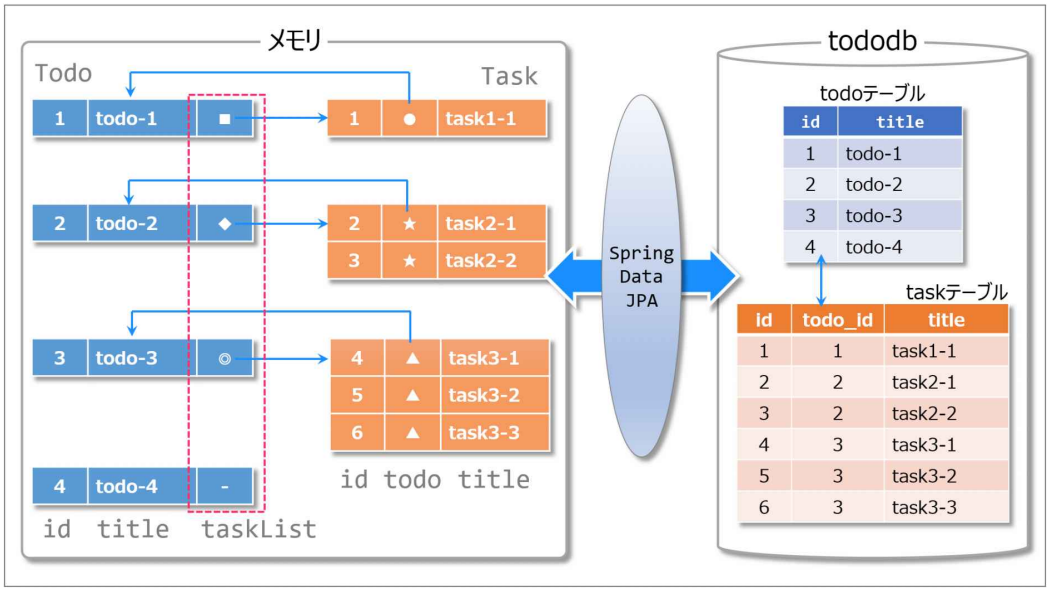
③@ToString(exclude="taskList")

・Lombokが生成するtoString()から、taskListを除外する指定です。

@Dataはsetter/getterの他にtoString()も生成します。デフォルトでは全プロパティが対象です。しかしtaskListの参照しているTaskオブジェクトは、自Todoオブジェクトを参照しています(下図のようにループしている)。このためデフォルトで生成されたtoString()を呼び出すと実行例外StackOverFlowが発生します。

これを避けるにはtoString()からtaskListを除外する必要があります。

前述の【図14-10】にTodo→Taskの関連を加えると以下のようになります。これでTask←→Todoの双方向に関連付けできました。



【図14-11】エンティティの関係(Task←→Todo)

14.5 関連するエンティティの取得

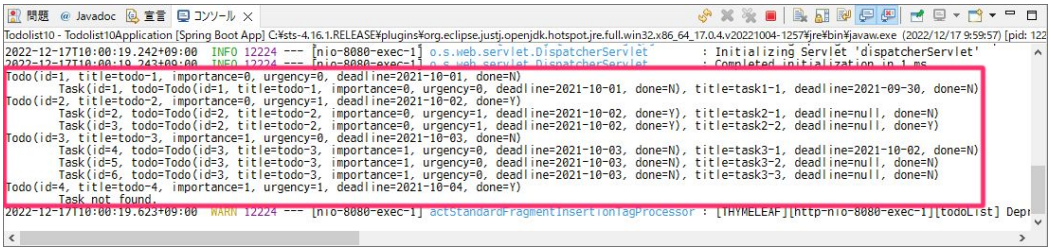
TodoとTaskの関連を定義できたのでアクセスしてみます。
まずコントローラーTodoListControllerのshowTodoList()に以下のコードを追加してください。

【リスト14-4】
com.example.todolist.controller.TodoListController#showTodoList()

```
package com.example.todolist.controller;
    :
import java.util.List;
import com.example.todolist.entity.Task;
    :
public class TodoListController {
    :
  // ToDo一覧表示
  @GetMapping("/todo")
  public ModelAndView showTodoList(ModelAndView mv,
                                   @PageableDefault(page = 0, size = 5,
sort = "id")
                                   Pageable pageable) {
      :
      mv.addObject("todoList", todoPage.getContent());

      // ----- 追加　ここから ↓ ↓ ↓ -----
      List<Todo> todoList = todoRepository.findAll();
      List<Task> taskList;
      for (Todo todo : todoList) {
          System.out.println(todo);
          taskList = todo.getTaskList();
          if (taskList.size() == 0) {
              System.out.println("¥tTask not found.");
          } else {
              for (Task task : taskList) {
```

```
                        System.out.println("¥t" + task);
                    }
                }
            }
        // ----- 追加　ここまで ↑ ↑ ↑ -----

        return mv;
    }
    :
}
```

この状態でhttp://localhost:8080/todoへアクセスすると、todoテーブルに加えtaskテーブルの内容もSTSのコンソールウィンドウへ出力されます。



追加したコードのtodoRepository.findAll()は、todoテーブルだけ検索しているように見えますが、taskテーブルの内容も取得しています。これは【リスト14-2】【リスト14-3】でTodoとTaskを関連付けた結果、【図14-11】のようにtodoを検索すると関連するtaskがtaskListプロパティへ格納されるためです。このように関連を定義すると、複数のテーブルを同時に操作できるようになります。

14.5.1 Task数の取得

ではtaskListプロパティを使って【図14-9】のように表示を追加して行きます。まずToDo一覧画面にタスク数を表示させます。

【リスト14-5】src/main/resources/templates/todoList.html(一部抜粋)

```
<!DOCTYPE html>
  :
<body>
```

```
    :
    <!-- 検索結果エリア -->
    <table border="1">
      <tr>
        <th>id</th>
        <th th:text="#{label.title}"></th>
        <th th:text="#{label.importance}"></th>
        <th th:text="#{label.urgency}"></th>
        <th th:text="#{label.task}"></th>    <!-- ① -->
        <th th:text="#{label.deadline}"></th>
        <th th:text="#{label.check}"></th>
      </tr>
      <tr th:each="todo:${todoList}">
        <!-- id -->
        <td th:text="${todo.id}"></td>
        <!-- 件名 -->
        <td>
          <a th:href="@{/todo/__${todo.id}__}" th:text="${todo.title}"></a>
        </td>
        <!-- 重要度 -->
        <td th:text="${todo.importance == 1 ? '★★★':'★'}"></td>
        <!-- 緊急度 -->
        <td th:text="${todo.urgency == 1 ? '★★★':'★'}"></td>
        <!-- タスク数 -->
        <td th:text="${#lists.size(todo.taskList)}" style="text-align: center"></td>
<!-- ② -->
        <!-- 期限 -->
        <td th:text="${todo.deadline}"></td>
        <!-- 完了 -->
        <td th:text="${todo.done == 'Y' ? '__#{text.done}__' : ''}"></td>
      </tr>
    </table>
    :
</body>
```

```
</html>
```

【リスト14-6】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
label.task=タスク
```

【リスト14-7】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
label.task=Task
```

①は見出しで、タスク数を出力しているのは②の行です。
#listsはThymeleafのリスト用ユーティリティオブジェクトで、size()は引数の要素数を返します。ToDoに関連するタスクはtodo.taskListに格納されているので、その要素数がタスク数です。

なおここで[新規登録]ボタンをクリックすると、実行時エラーになります。これは必要な処理をまだ作成していないためです(17章で解説します)。

14.5.2 Taskの取得

次にToDo入力画面にタスクを一覧します。この画面に表示するデータを取得しているのは、TodoListControllerのtodoById()です。

```
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int id,
ModelAndView mv) {
        mv.setViewName("todoForm");
        Todo todo = todoRepository.findById(id).get();
        mv.addObject("todoData", todo);
        session.setAttribute("mode", "update");
        return mv;
```

```
    }
```

これもfindById()でtodoテーブルを検索したとき、関連するタスクがtodo.taskListへセットされます。そのため変更は不要です。todoForm.htmlにタスク一覧表示の定義を追加するだけです。

【リスト14-8】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
      :
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">    <!-- ① --
>
      ■ToDo
      <!-- ToDo入力エリア -->
      <table>
      :
      </table>
      <!--   追加   ここから ↓ ↓ ↓ -->
      <hr style="margin-top: 2em; margin-bottom: 1em;">
      <!-- タスク一覧 -->
      ■Task
      <div th:if="${#lists.size(todoData.taskList)} > 0">    <!-- ② -->
        <table>
          <tr>
            <th>id</th>
            <th th:text="#{label.title}"></th>
            <th th:text="#{label.deadline}"></th>
            <th th:text="#{label.check}"></th>
          </tr>
          <!-- 登録済Task -->
          <tr th:each="task:*{taskList}">    <!-- ③ -->
```

```
                <td th:text="${task.id}"></td>
                <td th:text="${task.title}"></td>
                <td th:text="${task.deadline}"></td>
                <td th:text="${task.done}"></td>
            </tr>
        </table>
    </div>
    <!--　追加　ここまで ↑ ↑ ↑ -->
    <!-- 更新時の操作ボタン -->
    <div th:if="${session.mode == 'update'}">
    :
    </div>
  </form>
</body>
</html>
```

表示方法は基本的にTodo一覧と同じです。

まずtodoById()でセットしたtodoDataをth:objectで指定します(①)。このtodoDataのtaskListが要素(=タスク)を持っていたら一覧表示のdiv要素を生成します(②)。

一覧の表示にはth:eachを使います(③)。変数選択式*{taskList}は、todoData.taskListのことであり、List<Task>型です。これを変数taskに取り出しながらtr要素、td要素を生成します。

これで入力画面にタスク一覧が表示されるようになります。

ここでも[更新]ボタンをクリックすると、必要な処理が無いため実行時エラーになります。これは次章で処理を追加します。

[削除]ボタンは機能します。しかもToDoだけでなく、関連するタスクも削除します。これについては16章で解説します。

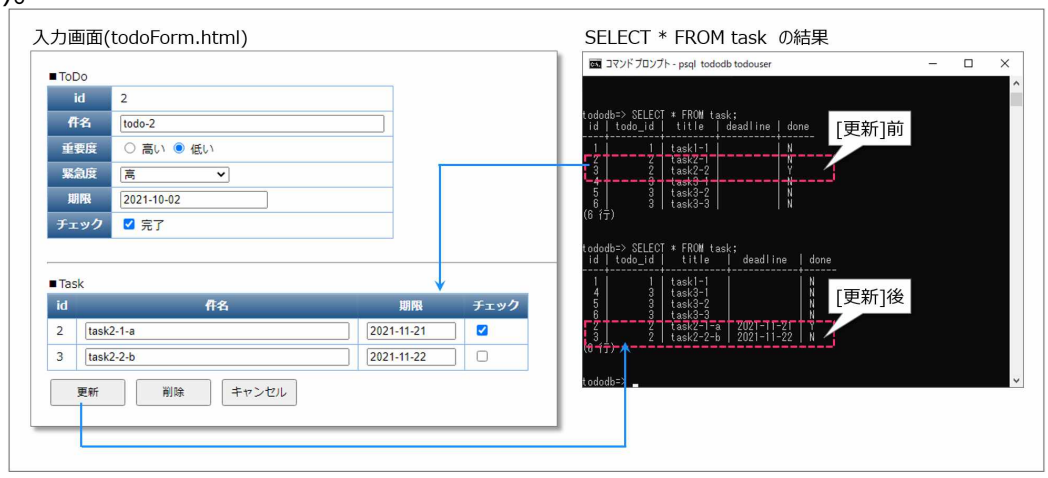
15. 関連するエンティティの更新

本章では登録済みのタスクを変更できるようにします。

前章ではToDo入力画面(todoForm.html)にタスクを一覧表示しましたが、その部分をinput要素などに置き換え、更新処理を追加していきます。

【図15-1】タスク更新エリア

[更新]ボタンをクリックすると、変更後の内容をtaskテーブルに反映します。psqlでtaskテーブルを検索すると、更新前後の様子を確認できます(レコードの表示順が変わっていることに注意してください)。



【図15-2】taskテーブルの更新

また件名と期限はToDoと同様のチェックを行います。

【図15-3】タスクデータのエラーチェック

なおToDoとタスクは同時に更新します。ToDoの内容が変更されていれば、それもtodoテーブルへ反映します。

【図15-4】ToDoデータの更新

作成するプロジェクトの仕様

プロジェクト名	Todolist10a
作成ファイル	com.example.todolist.form.TaskData.java

```
Todolist10a
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist10aApplication.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ Task.java(▲)
 │    │    └ Todo.java(▲)
 │    ├ com.example.todolist.form
 │    │    ├ TaskData.java(★)
 │    │    ├ TodoData.java(▲)
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java(▲)
 └ src/main/resources
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties(▲)
    │    ├ FixedDisplayStrings_ja.properties(▲)
    │    ├ OperationMessages_en.properties
```

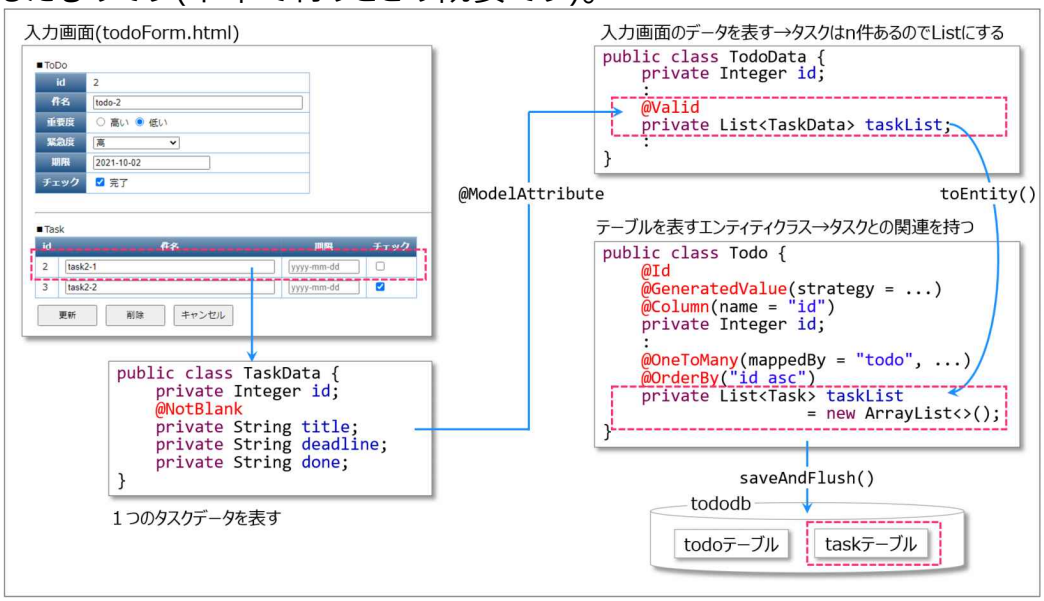
```
    │      ├ OperationMessages_ja.properties
    │      ├ ValidationMessages_en.properties(▲)
    │      └ ValidationMessages_ja.properties(▲)
    :
    ├ templates
    │      ├ fragments.html
    │      ├ todoForm.html(▲)
    │      └ todoList.html
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

15.1 フォームクラスの追加

ToDoアプリでは、ToDo入力画面のinput要素の値をTodoDataオブジェクトのプロパティで表しています。そのため画面にタスク更新エリアを追加するなら、TodoDataにも対応するプロパティが必要です。下図は入力されたタスクデータをどのようにtaskテーブルへ反映するか表したものです(本章で行うことの概要です)。



【図15-5】タスクデータ用の定義追加

まず更新エリアの1行分を表すフォームクラスTaskDataを作成します。前章で説明したように、タスクはToDoに対してn件あるので、TodoDataではList<TaskData>型のプロパティとし、タスク更新エリア全体を表します。

そしてこれをエンティティTodoのtaskListプロパティへセットするようTodoData#toEntity()を拡張します。あとはsaveAndFlush()を実行すればtaskテーブルが変更されます。

以下、詳細に見ていきます。最初はTaskDataです。

【リスト15-1】com.example.todolist.form.TaskData.java

```
package com.example.todolist.form;

import jakarta.validation.constraints.NotBlank;
import lombok.AllArgsConstructor;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class TaskData {
	private Integer id;

	@NotBlank
	private String title;

	private String deadline;
	private String done;

}
```

件名(title)は@NotBlankでバリデーションします。全角スペースのチェック、期限チェックはサービスクラスTodoServiceで行います(後述)。

これをTodoDataへ追加します。

【リスト15-2】com.example.todolist.form.TodoData.java(一部抜粋)

```
package com.example.todolist.form;
	:
import java.util.List;
import jakarta.validation.Valid;
import com.example.todolist.common.Utils;
import com.example.todolist.entity.Task;
	:
@Data
public class TodoData {
	:
	private String done;

	@Valid	//	②
	private List<TaskData> taskList;	//	①
```

```
    // 入力データからTodo Entityを生成して返す
    public Todo toEntity() {
        Todo todo = new Todo();

        // ToDo部分
            :
         todo.setDone(done);

        // ③Task部分追加　ここから ↓ ↓ ↓
        Date date;
        Task task;
       if (taskList != null) {
            for (TaskData taskData : taskList) {
                date = Utils.str2dateOrNull(taskData.getDeadline());
                task = new Task(
                        taskData.getId(), null, taskData.getTitle(), date,
taskData.getDone());
                todo.addTask(task);
            }
        }
        // ③Task部分追加　ここまで ↑ ↑ ↑

        return todo;
    }
}
```

①タスクデータをList<TaskData>型のプロパティとして追加する。
TaskDataをListでラップし、タスク更新エリア全体を表します。
ここには後述するように@ModelAttributeでタスク更新エリアの内容がバインドされます。
また反対に、ここに設定されている値はタスク更新エリアの各input要素へセットされます。
②taskListに**@Valid**アノテーションを付与する。

@Validは、そのクラス内で定義されているバリデーションを有効にします。
ここでは(taskListではなく)TaskDataのtitleに付与された@NotBlankを実行する、という意味になります。

③バインドされたtaskListからTaskオブジェクトを作成しTodoオブジェクトへセットする処理を追加する。
ここでTaskの全プロパティを対象とするコンストラクタを使うので、Taskクラスには@AllArgsConstructorを追加します。またデフォルトコンストラクタ(引数のないコンストラクタ)を生成する**@NoArgsConstructorアノテーション**も追加します。

※addTask()は【リスト15-4】で説明します。

【リスト15-3】com.example.todolist.entity.Task.java(追加個所)

```
package com.example.todolist.entity;
        :
import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;
      :
@Entity
@Table(name = "task")
@Data
@AllArgsConstructor   //   追加
@NoArgsConstructor   //   追加
public class Task {
        :
}
```

前章のTaskクラスにはコンストラクタ定義がありませんでした。この場合、Javaコンパイラが自動的にデフォルトコンストラクタを生成します。これを内部でSpring Bootが使用していました。

しかし@AllArgsConstructorを指定すると(コンストラクタが作成されるので)、Javaコンパイラは**デフォルトコンストラクタを生成しません**。よってこのままではSpring Bootが実行時エラーを起こします。そこで@NoArgsConstructorにより、デフォルトコンストラクタも生成されるようにします。

また【リスト15-2】ではコンストラクタの 2 番目の引数にnullを渡していることに注意してください。

```
new Task(task.getId(), null, task.getTitle(), date, task.getDone())
```

これはTaskと関連付いているTodoへの参照を表すtodoプロパティにセットする値です。

```
@ManyToOne
@JoinColumn(name = "todo_id")
private Todo todo;
```

ここにはTodo#addTask()で、このTodoへの参照を設定します(①)。

【リスト15-4】com.example.todolist.entity.Todo#addTask()

```
package com.example.todolist.entity;
    :
import java.util.ArrayList;
import jakarta.persistence.OrderBy;
    :
@Entity
@Table(name = "todo")
@Data
@ToString(exclude = "taskList")
public class Todo {
    :
    @OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)
    @OrderBy("id asc")   //   ②
    private List<Task> taskList = new ArrayList<>(); //   ③

    // Todoへの参照設定
    public void addTask(Task task) {   //   ①
        task.setTodo(this);
        taskList.add(task);
    }
}
```

> ①のaddTask()がどうして必要なのかわからない、何をやっているのかわからない、という方向けにパラパラ漫画風の解説を章末に載せています。そちらを参考にしてください。

また**@OrderByアノテーション**を付与し、タスクをidの昇順(asc)で取得します(②)。これは本章冒頭で触れたように、レコードを更新すると検索結果の並び順が変わるためです。タスクの表示順は一定にしたいので、このアノテーションで並び順を指定します。

なおtaskListはArrayList<Task>オブジェクトで初期化します(③)。これはtaskList.add(task)でNullPointerExceptionにならないようにするためです。

15.2 関連エンティティの更新用フィールド追加

次に前章でToDo入力画面に作成したタスク一覧を更新エリアに変更します。

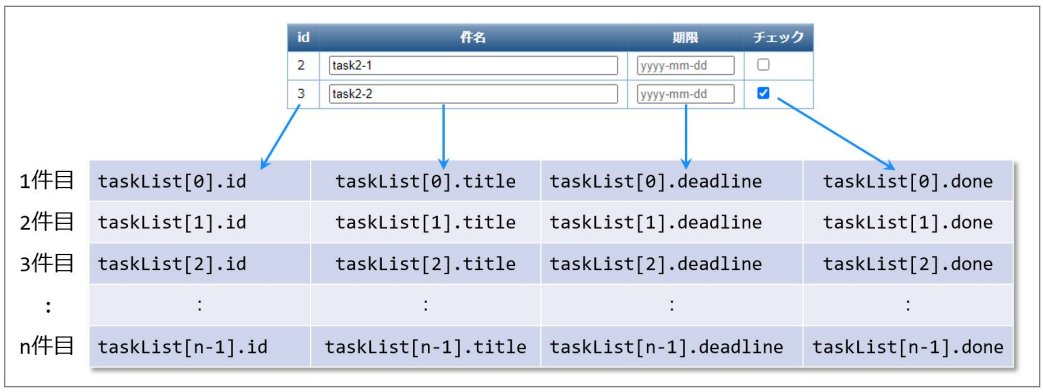
【リスト15-5】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
    ■ToDo
    <!-- ToDO入力エリア -->
    <table>
    :
    </table>

    <hr style="margin-top: 2em; margin-bottom: 1em;">
    ■Task
    <!-- 変更　ここから ↓ ↓ ↓ -->
    <!-- タスク更新エリア -->
    <div th:if="${#lists.size(todoData.taskList)} != 0">
      <table>
        <tr>
          <th>id</th>
          <th th:text="#{label.title}"></th>
          <th th:text="#{label.deadline}"></th>
          <th th:text="#{label.check}"></th>
        </tr>
        <!-- 登録済みTask -->
        <tr th:each="task,stat:*{taskList}">
          <!-- id -->
          <td>
```

```
                <span th:text="${task.id}"></span>
                <!-- 更新 のために必要 -->
                <input type="hidden" th:name="${'taskList[' + stat.index + '].id'}"
                  th:value="${task.id}" />
              </td>
              <!-- 件名 -->
              <td>
                <input type="text" th:name="${'taskList[' + stat.index + '].title'}"
size="40"
                  th:value="${task.title}">
                <div th:if="${#fields.hasErrors('taskList[' + stat.index + '].title')}"
                  th:errors="*{taskList[__${stat.index}__].title}" th:errorclass="red">
</div>
              </td>
              <!-- 期限 -->
              <td>
                <input type="text" th:name="${'taskList[' + stat.index +
'].deadline'}" size="10"
                  th:value="${task.deadline}" placeholder="yyyy-mm-dd">
                <div th:if="${#fields.hasErrors('taskList[' + stat.index + '].deadline')}"
                  th:errors="*{taskList[__${stat.index}__].deadline}"
th:errorclass="red"></div>
              </td>
              <!-- チェック -->
              <td>
                <input type="checkbox" th:name="${'taskList[' + stat.index +
'].done'}" value="Y"
                  th:checked="*{taskList[__${stat.index}__].done=='Y'}" />
                <input type="hidden"
th:name="${'!taskList[__${stat.index}__].done'}" value="N" />
              </td>
            </tr>
          </table>
        </div>
```

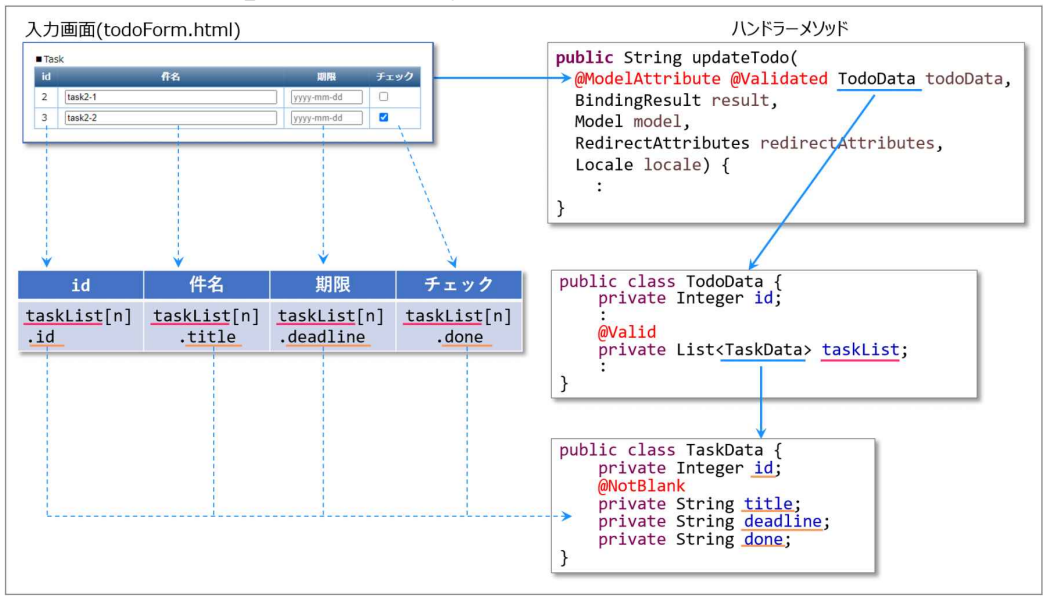
```
    <!-- 変更　ここまで ↑ ↑ ↑ -->
    :
</html>
```

ここでのポイントはth:nameで指定する名前を「**taskList[n].TaskDataのプロパティ名**」とすることです。



【図15-6】タスクデータ更新エリアの名前

これは画面に入力されたデータを@ModelAttributeでTodoDataオブジェクトへバインドするとき、「何をどこへセットするか」を表しています。



【図15-7】繰り返し構造を持つ関連エンティティの取得

この図が示すようにTodoDataオブジェクトは、taskListプロパティを持っています。そしてtaskListはList<TaskData>型なので、複数のTaskDataオブジェクトから成ります。さらにTaskDataは画

面で指定されたid, title, deadline, doneという名前のプロパティを持っています。
こういった場合Spring Bootは以下のようにデータをバインドします。
①TaskDataオブジェクトを生成する
②①のid, title, deadline, doneプロパティへ画面の対応するフィールドの値をセットする
③②をtaskListへ追加する
④①～③を行ごとに繰り返す。

また前述の定義には「taskListのn番目」という情報が必要になるためth:eachに**ステータス変数**を追加します。

```
<tr th:each="task,stat:*{taskList}">
```

ステータス変数はth:each="各要素を表す変数, ステータス変数 : 繰り返し対象"のように指定します(名前は自由ですがstat, あるいはstatusとすることが多いようです)。ステータス変数からは下表のプロパティを介して各種情報を取得できます。

【表15-1】ステータス変数のプロパティ

プロパティ	内容
index	現在処理中の要素番号(0始まり)
count	現在処理中の要素番号(1始まり)
size	繰り返し対象の全要素数
current	現在の要素オブジェクト
even	現在の要素が偶数番目ならtrue
odd	現在の要素が奇数番目ならtrue
first	現在の要素が先頭要素ならtrue
last	現在の要素が最終要素ならtrue

こうするとstat.indexでtaskListの何番目(0始まり)か取得できるので、これを使いth:nameを作成します。

```
<input type="hidden" th:name="${'taskList[' + stat.index + '].id'}"
  th:value="${task.id}" />
```

エラー表示は少々複雑なので、まず同じ画面にあるToDoの件名の場合、どのように定義したか確認してみます。

```
<td>
  <input type="text" name="title" size="40" th:value="*{title}">
  <div th:if="${#fields.hasErrors('title')}" th:errors="*{title}" th:errorclass="red">
</div>
</td>
```

このようにエラー有無を判断する#fields.hasErrors()の引数はname値を''で囲んだもの、エラーメッセージを取得するth:errosはname値と同じです。

タスクの件名の場合、name値はtaskList[n].titleです。これを同じように指定します。

```
<!-- 件名 -->
<td>
  <input type="text" th:name="${'taskList[' + stat.index + '].title'}" size="40"
    th:value="${task.title}">
  <div th:if="${#fields.hasErrors('taskList[' + stat.index + '].title')}"
    th:errors="*{taskList[__${stat.index}__].title}" th:errorclass="red"></div>
</td>
```

th:errorsは文字列ではない(Springの式言語で指定する)ため、事前評価していることに注意してください。

期限もこれと同じようにtaskList[n].deadlineをもとに指定します。

15.3 関連エンティティの更新処理

最後はコントローラーTodoListControllerの処理です。といっても、実際に追加するのはタスクのチェック処理だけです。

ToDo入力画面を表示するtodoById()は、前章でも説明したようにfindById()でToDoと関連するタスクを取得し、ToDo入力画面へ渡しています。よって変更不要です。

```
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int id,
ModelAndView mv) {
        mv.setViewName("todoForm");
        Todo todo = todoRepository.findById(id).get();// <-ToDoとタスクを取得している
        mv.addObject("todoData", todo);
        session.setAttribute("mode", "update");
        return mv;
    }
```

更新処理はupdateTodo()です。

【リスト15-6】

com.example.todolist.controller.TodoListController#updateTodo()

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                             BindingResult result,
                             Model model,
                             RedirectAttributes redirectAttributes,
                             Locale locale) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result, false, locale);
        if (!result.hasErrors() && isValid) {
            // エラーなし -> 更新
            Todo todo = todoData.toEntity();// <-タスクもセットしている
```

```
            todo = todoRepository.saveAndFlush(todo);// <- ToDoとタスクを更新する
                    :
            return "redirect:/todo/" + todo.getId();// <- 再表示する(①)
        } else {
            // エラーあり -> エラーメッセージをセット
                    :
        }
    }
```

@ModelAttributeでバインドされたtodoDataからTodoData#toEntity()でTodoエンティティを作成するとき、関連するTaskエンティティも作成しています(【リスト15-2】)。この状態でsaveAndFlush()を実行するとtodoテーブルだけでなくtaskテーブルも更新されます。

よってupdateTodo()も変更不要なのですが、次に表示する画面を変更します(①)。これまでは更新が成功したらToDo一覧へ戻りましたが、再度入力画面を表示させます。これはタスクが更新されたことをその場で確認できるようにするためです。

また入力画面から一覧画面に戻るボタンのキャプションを「キャンセル」から[戻る]へ変更します(処理に変更はありません)。ここまで「入力画面で操作するのを止めて(=キャンセルして)一覧画面へ戻るという意味合いだったのですが、これが変わったためです。

【リスト15-7】src/main/resources/i18n/FixedDisplayStrings_ja.properties(変更箇所)

```
button.cancel=戻る
```

【リスト15-8】src/main/resources/i18n/FixedDisplayStrings_en.properties(変更箇所)

```
button.cancel=Back
```

新たに必要となるのは、入力されたタスクデータのバリデーションだけです。【リスト15-1】でタスクの件名(title)に@NotBlankを付与したので、このエラーメッセージをプロパティファイルへ追加します。キー名称は「アノテーション名.フォームオブジェクト名.プロパティ名」です。

【リスト15-9】src/main/resources/i18n/ValidationMessages_ja.properties(追加分)

```
NotBlank.taskList.title=件名を入力してください
```

【リスト15-10】src/main/resources/i18n/ValidationMessages_en.properties(追加分)

```
NotBlank.taskList.title=Title is required.
```

ToDo同様、チェックはサービスクラスTodoService で行います。

【リスト15-11】com.example.todolist.service.TodoService#isValid()

```
package com.example.todolist.service;
    :
import java.util.List;
import com.example.todolist.form.TaskData;
    :
@Service
@AllArgsConstructor
public class TodoService {
    private final MessageSource messageSource;

    // Todo + Taskのチェック
    public boolean isValid(TodoData todoData, BindingResult result, boolean
isCreate,
                                        Locale locale) {
        boolean ans = true;

        // Todo部分のチェック
        // 件名が全角スペースだけで構成されていたらエラー
        if (!Utils.isBlank(todoData.getTitle())) {
            :
        }
            :
```

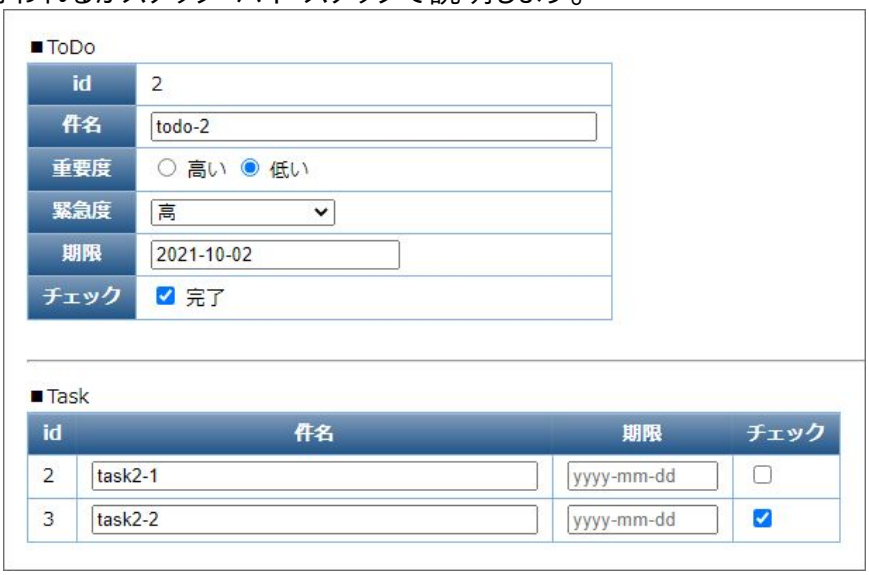
```
            // Taskのチェック(追加　ここから↓ ↓ ↓)
            List<TaskData> taskList = todoData.getTaskList();
            if (taskList != null) {
                // すべてのタスクに対して以下を実行する
                //「タスクのn番目」という情報が必要なので(拡張for文でなく)for文を使用する
                for (int n = 0; n < taskList.size(); n++) {
                    TaskData taskData = taskList.get(n);

                    // タスクの件名が全角スペースだけで構成されていたらエラー
                    if (!Utils.isBlank(taskData.getTitle())) {
                        if (Utils.isAllDoubleSpace(taskData.getTitle())) {
                            FieldError fieldError = new FieldError(
                                    result.getObjectName(),
                                    "taskList[" + n + "].title", //　①
                                    messageSource.getMessage(
                                            "DoubleSpace.todoData.title", null, locale));
                            result.addError(fieldError);
                            ans = false;
                        }
                    }

                    // タスク期限のyyyy-mm-dd形式チェック
                    String taskDeadline = taskData.getDeadline();
                    if (!taskDeadline.equals("") && !Utils.isValidDateFormat(taskDeadline)) {
                        FieldError fieldError = new FieldError(
                                result.getObjectName(),
                                "taskList[" + n + "].deadline",　//　②
                                messageSource.getMessage(
                                        "InvalidFormat.todoData.deadline", null, locale));
                        result.addError(fieldError);
                        ans = false;
                    }
```

```
                }
            }
            // Taskのチェック(追加　ここまで↑ ↑ ↑)

            return ans;
        }
}
```

ポイントはFieldErrorオブジェクトを作成するときのフィールド名です(①②)。
ToDoのフィールド名は"title"といったものでしたが、タスクは【図15-6】のように「taskList[n].TaskDataのプロパティ名」という形式なのでこれに合わせます。この際、プログラム中にコメントで記載しているように拡張for文ではなくfor文を使います。これは「いま何番目の要素か？」という情報が拡張for文では取得できないためです。

補足：addTask()詳説

ここではToDo入力画面が以下の状態だったとき、TodoData#toEntity()でどのような処理が行われるかステップ・バイ・ステップで説明します。




```
public class TodoData {
                :
    // 入力データからTodo Entityを生成して返す
    public Todo toEntity() {
        Todo todo = new Todo();   //   ①
        // ToDo部分
                :
        todo.setDone(done);   //   ②

        // Task部分
        Date date;
        Task task;
        for (TaskData taskData : taskList) {
            date = Utils.str2dateOrNull(taskData.getDeadline());
            //   ③
            task = new Task(
```

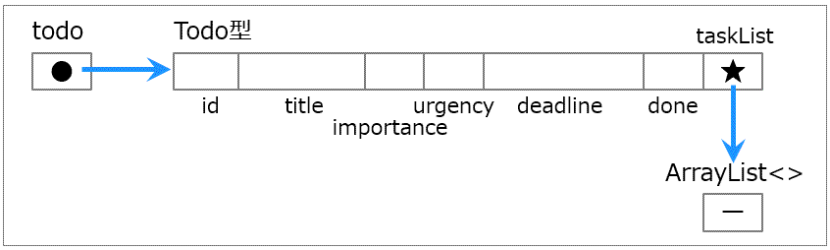
```
                        taskData.getId(), null, taskData.getTitle(), date,
taskData.getDone());
                todo.addTask(task);    //    ④
            }
            return todo;
        }
}
```

```
public class Todo {
        :
        public void addTask(Task task) {
                task.setTodo(this);    //    ⑤
                taskList.add(task);    //    ⑥
        }
}
```

なお、●★▲などの記号は前書5章補足と同じように「参照型データが格納されている場所を示す特別な値」、青矢印は「参照」、赤点線は「参照の代入」を表します。

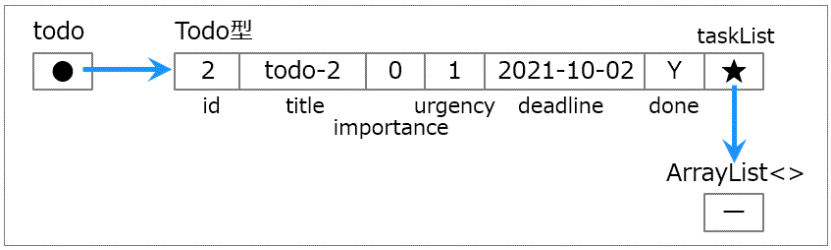
①Todo todo = new Todo();

右辺のnew演算子でTodoオブジェクトの領域を作成し、その場所を左辺のtodoへ代入します。この時点ではid～doneはnullです。taskListはArrayList<Task>型の領域を参照しています。



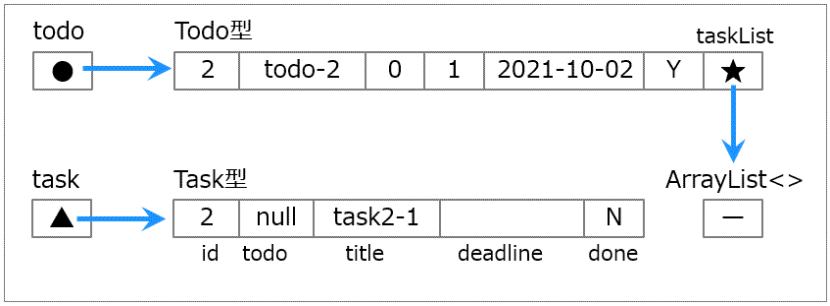
②todo.setDone(done)実行時点

バインドされたデータがtodoへセットされます。id～doneも参照型(Integer, String, Date)なので青矢印で表すべきですが、図が煩雑になるのでプリミティブ型と同じ書き方にしています。



③task = new Task(taskData.getId(), null, taskData.getTitle(), date,
taskData.getDone());

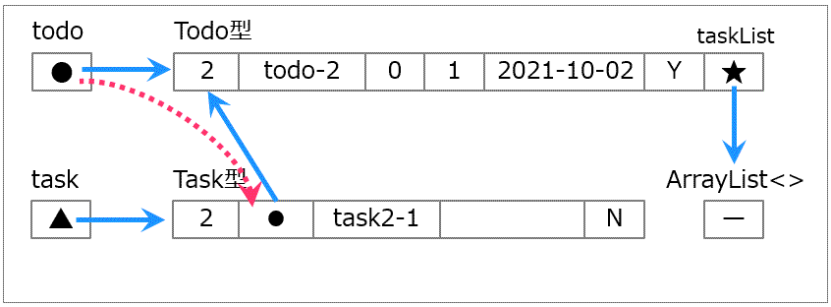
右辺でTaskオブジェクトの領域を作成すると同時にプロパティをセットし、その場所を左辺へ代入します。
この時点ではTaskオブジェクトのtodoプロパティはnullです。



④todo.addTask(task)→⑤task.setTodo(this);

このthisは④のtodoを表しています。よってtodoの値(●)がtaskのtodoプロパティに代入されます。

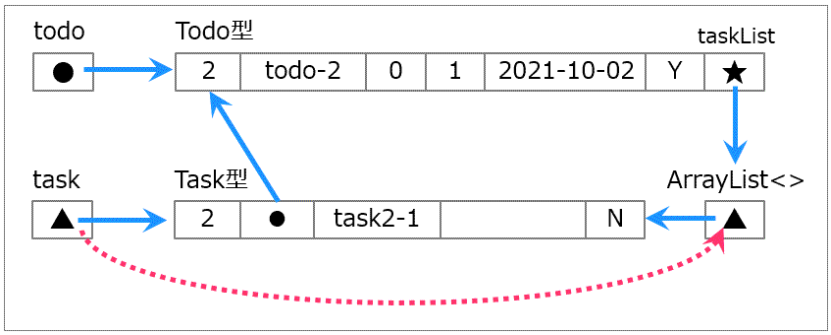
これでtaskからtodoを参照できるようになりました。



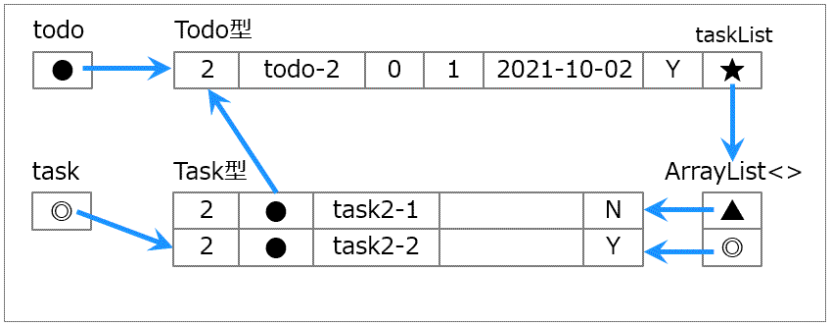
⑥taskList.add(task);

taskの値(▲)をtaskListへ追加します。

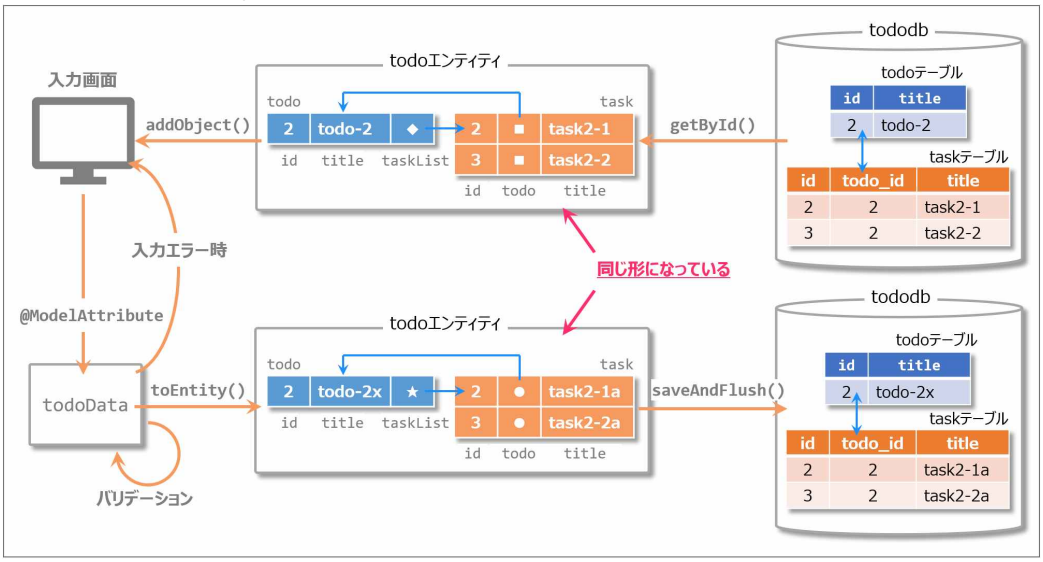
これでtodoからもtaskを参照できるようになりました。



⑦タスク更新エリアの2行目も③～⑥と同様にします。



この結果は何を表しているのでしょうか？実はfindAll()やfindById()でtodoを検索したときと同じ状態を作成しています。



【図15-8】タスクデータのライフサイクル

todoを検索すると関連するtaskも取得します。このとき関連は参照に置き換わります。これはSpring Data JPAが自動的に行います。

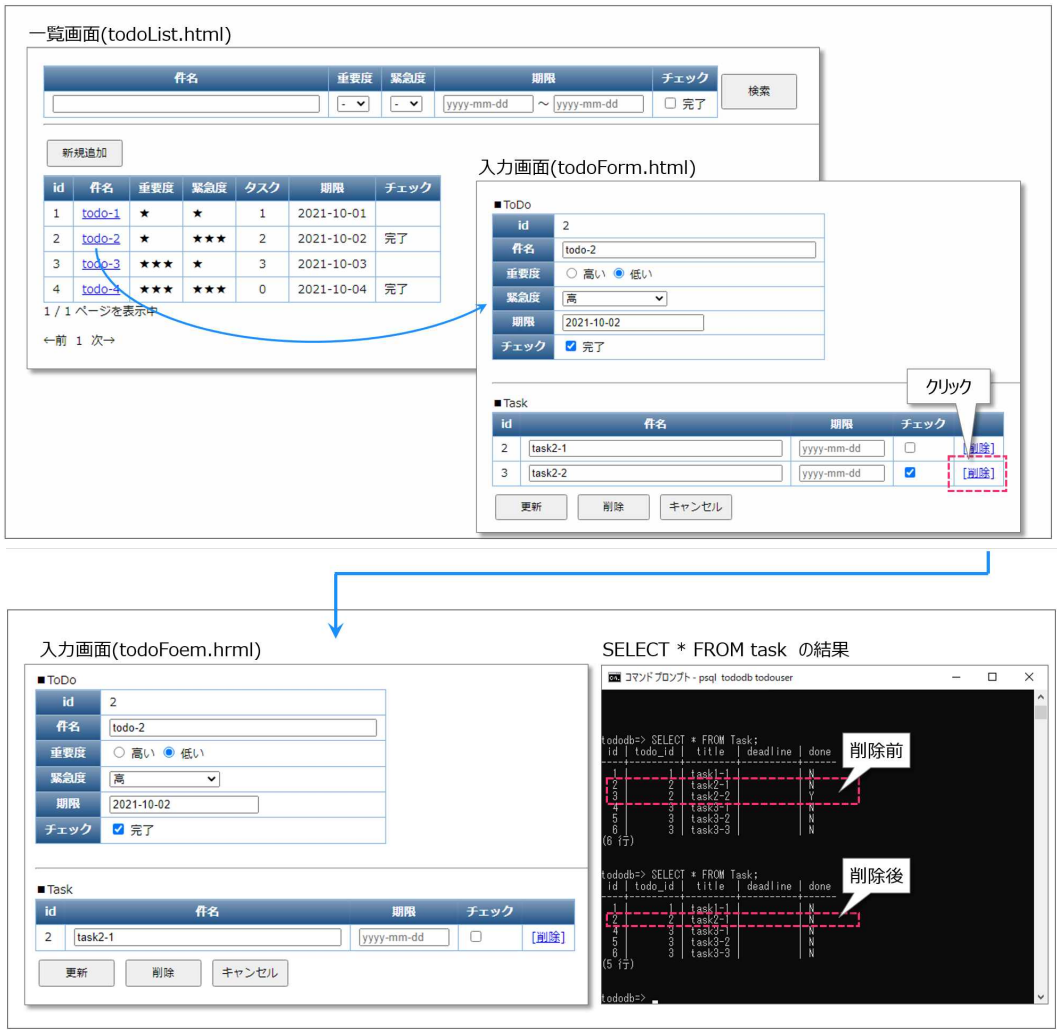
saveAndFlush()で更新するときも、これと同じようにtodoとtaskを関連付けた状態にする必要があります。つまり@ModelAttributeでバインドされたデータからtodoオブジェクト、taskオブジェクトを作成するだけでなく、それら同士を参照できるようにしなければなりません。これがここで説明した処理の目的です。

> なぜこういう(複雑な)処理が必要かというと、ToDo入力画面の[更新]ボタンが「ToDoとタスクを同時に更新する」という仕様だからです。これを[ToDo更新][タスク更新]ボタンに分け、ToDoとタスクを別々に更新するならもっと簡単にできます。

16. 関連するエンティティの削除

本章ではタスクの削除機能を追加します。

14章の最後で触れたように、ToDo入力画面(todoList.html)の[削除]ボタンをクリックすると、ToDoと関連するタスクが削除されます。一方、本章ではタスク更新エリアに削除リンクを設け、当該タスクのみ対象とします。そして画面はToDo一覧画面(todoList.html)へ戻らず、同じ入力画面に残りのタスクを表示します。



【図16-1】タスクの削除

作成するプロジェクトの仕様

プロジェクト名	Todolist10b
作成ファイル	com.example.todolist.repository.TaskRepository.java

```
Todolist10b
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist10bApplication.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ Task.java
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ TaskData.java
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    ├ TaskRepository.java(★)
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java
 └ src/main/resources
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties(▲)
```

```
    │    ├ FixedDisplayStrings_ja.properties(▲)
    │    ├ OperationMessages_en.properties(▲)
    │    ├ OperationMessages_ja.properties(▲)
    │    ├ ValidationMessages_en.properties
    │    └ ValidationMessages_ja.properties
    :
    ├ templates
    │    ├ fragments.html
    │    ├ todoForm.html(▲)
    │    └ todoList.html
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

タスクの削除リンクを作成する前に、すでにある[削除]ボタンの処理を確認します。上述したように、このボタンはToDoと関連するタスクを削除します。処理はコントローラTodoListControllerのdeleteTodo()で行っています。

```
// ToDo削除処理
@PostMapping("/todo/delete")
public String deleteTodo(@ModelAttribute TodoData todoData) {
        todoRepository.deleteById(todoData.getId());
                :
        return "redirect:/todo";
}
```

ここでブラウザから送られてきたidをキーにdeleteById()でTodoを削除しますが、このTodoのtaskListプロパティは関連するTaskを参照しています。そしてTodoに行ったデータ操作(登録、更新、削除など)はTaskにも適用する指定しています。

```
//    Todoへの操作をTaskにも適用する
@OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)
private List<Task> taskList;
```

よってTodoを削除すると関連する(=参照している)Taskも同時に削除されるわけです。

16.1 Taskリポジトリの追加

ではタスクのみを削除するリンクを作成していきます。

まずTaskの削除処理ですが、前述のTodo削除は、リポジトリで生成されたdeleteById()で行っていました。Taskも同じようにidを指定して削除するので、このdeleteById()が必要です。そこでTaskエンティティのリポジトリを追加します。

【リスト16-1】com.example.todolist.repository.TaskRepository .java

```
package com.example.todolist.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.Task;

@Repository
public interface TaskRepository extends JpaRepository<Task, Integer> {
}
```

これで前書「【表6-5】自動実装されるメソッド」で示した一連のメソッドが使えるようになります。この中にdeleteById()も含まれています。

16.2 Taskの削除

次に入力画面へ削除リンクを追加します。

【リスト16-2】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
  :
<body>
  <!-- メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
    <!-- ToDO入力エリア -->
    ■ToDo
    :
    ■Task
    <!-- タスク更新エリア -->
    <div th:if="${#lists.size(todoData.taskList)} != 0">
      <table>
        <tr>
          <th>id</th>
          <th th:text="#{label.title}"></th>
          <th th:text="#{label.deadline}"></th>
          <th th:text="#{label.check}"></th>
          <th></th><!-- 削除リンクのために1列追加 -->
        </tr>
        <!-- 登録済Task -->
        <tr th:each="task,stat:*{taskList}">
            :
          <!-- 削除リンクを追加する　ここから ↓ ↓ ↓ -->
          <!-- 削除リンク -->
          <td>
            <a th:href="@{/task/delete(task_id=${task.id},todo_id=*{id})}"
              th:text="#{link.delete}"></a>
```

```
            </td>
            <!-- 削除リンクを追加する　ここまで ↑ ↑ ↑ -->
          </tr>
        </table>
          :
  </form>
</body>
</html>
```

【リスト16-3】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
link.delete=[削除]
```

【リスト16-4】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
link.delete=[Del]
```

ポイントは削除リンクのURLを作成するth:hrefです。

```
th:href="@{/task/delete(task_id=${task.id},todo_id=*{id})}"
```

【リスト16-2】ように${task.id}はこのタスクのid、*{id}は関連するToDoのidを表しています(ToDoのidが必要な理由は後述)。例えば【図16-1】のようにid=2のTodoに関連付いているid=3のTaskの場合、URLは次のようになります。

```
th:href="@{/task/delete(task_id=${task.id},todo_id=*{id})}"
 ↓
th:href="@{/task/delete(task_id=3,todo_id=2)}"
 ↓
href="/task/delete?task_id=3&todo_id=2"
```

これに対応するタスク削除用のハンドラーメソッドdeleteTask()をコントローラに追加します。

【リスト16-5】
com.example.todolist.controller.TodoListController#deleteTask()

```
package com.example.todolist.controller;
    :
import com.example.todolist.repository.TaskRepository;
import org.springframework.web.bind.annotation.RequestParam;
    :
@Controller
@RequiredArgsConstructor
public class TodoListController {
    :
    private final TaskRepository taskRepository;    //    ①
    :
    // Task削除処理
    @GetMapping("/task/delete")
    public String deleteTask(@RequestParam(name = "task_id") int taskId, //
②
                             @RequestParam(name = "todo_id") int
todoId, //    ②
                             RedirectAttributes redirectAttributes, Locale
locale) {
        // ③Taskを削除
        taskRepository.deleteById(taskId);

        // 削除完了メッセージをセットしてリダイレクト
        String msg = messageSource.getMessage("msg.i.task_deleted", null,
locale);
        redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
        return "redirect:/todo/" + todoId;    // ④
    }
}
```

【リスト16-6】src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.task_deleted=Taskを削除しました。
```

【リスト16-7】src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.task_deleted=Task has been deleted.
```

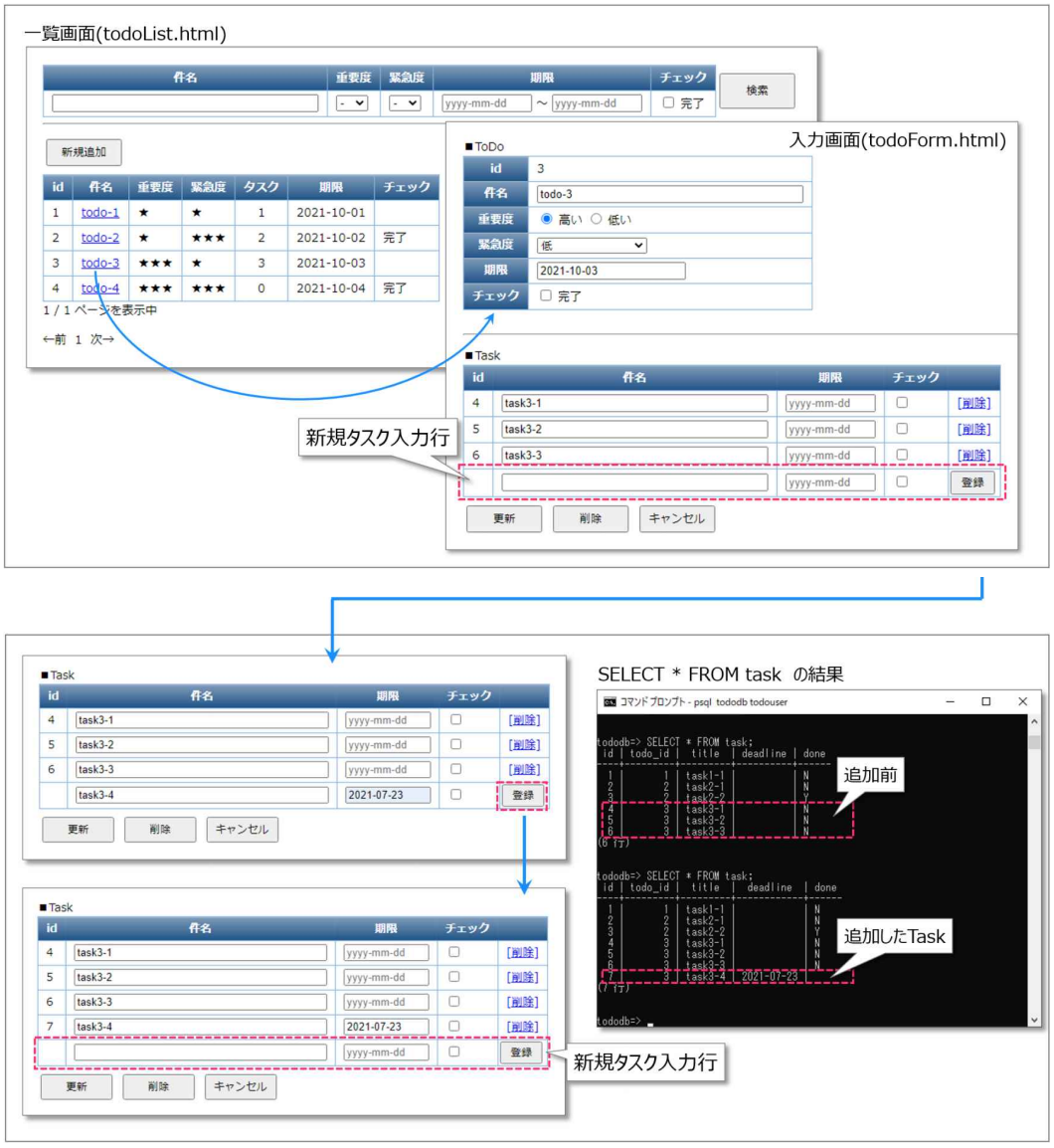
まず【リスト16-1】のTaskRepository型変数を定義し、コンストラクターインジェクションされるようにします(①)。

リンクのURLパラメーターで指定されたタスクのid, ToDoのidは、それぞれtaskId, todoIdへバインドします(②)。そしてtaskIdをTaskRepositoryで自動実装されたdeleteById()の引数にしてタスクを削除します(③)。

todoIdは、削除後のリダイレクト先指定に使います(④)。例えばtodoId=2ならリダイレクト先は"/todo/2"となり、タスク削除後に入力画面が再表示されます。

17. 関連するエンティティの追加

本章ではタスクを登録できるようにします。
入力エリアはタスク更新エリアの直後に追加します。



【図17-1】タスクの登録

件名と期限についてはToDoと同様にチェックします。

【図17-2】登録タスクの入力チェック

作成するプロジェクトの仕様

プロジェクト名	Todolist10c

```
Todolist10c
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist10cApplication.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ Task.java
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ TaskData.java
 │    │    ├ TodoData.java(▲)
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    ├ TaskRepository.java
 │    │    └ TodoRepository.java
 │    └ com.example.todolist.service
 │         └ TodoService.java(▲)
 └ src/main/resources
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties
    │    ├ FixedDisplayStrings_ja.properties
```

```
    │      ├ OperationMessages_en.properties(▲)
    │      ├ OperationMessages_ja.properties(▲)
    │      ├ ValidationMessages_en.properties(▲)
    │      └ ValidationMessages_ja.properties(▲)
    :
    ├ templates
    │      ├ fragments.html
    │      ├ todoForm.html(▲)
    │      └ todoList.html
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

17.1 新規タスク入力行の表示

15章でも説明したようにToDo入力画面のinput要素の値はTodoDataで保持します。新規タスク入力行に対応するプロパティもここに追加します。

【リスト17-1】com.example.todolist.form.TodoData.java(追加個所)

```
public class TodoData {
    :
    @Valid
    private List<TaskData> taskList;

    private TaskData newTask;    //  ←   (追加)新規タスク入力行用
    :
}
```

追加したnewTaskには@Validを付与しません。付与すると**[更新]ボタン**をクリックしたとき、新規タスクの件名が入力されていないと@NotBlankバリデーションでエラーになるためです。そこでこれに相当するチェックは、サービスクラスで行ないます。

【図17-3】@Validがあると[更新]クリックでエラーメッセージが表示される

そして新規タスク入力行(newTaskプロパティ)に対応するinput要素を入力画面に作成します。

【リスト17-2】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
    :
<body>
  <form th:action="@{/}" method="post" th:object="${todoData}">
    ■ToDo
    <!-- ToDO入力エリア -->
    <table>
      :
    </table>

    <div th:if="${session.mode == 'update'}"><!-- ②更新の場合、Task一覧を表示する -->
      <hr style="margin-top: 2em; margin-bottom: 1em;">
      ■Task
      <div th:if="${#lists.size(todoData.taskList)} != 0">
      <table>
        <tr>
          <th>id</th>
          <th th:text="#{label.title}"></th>
          <th th:text="#{label.deadline}"></th>
          <th th:text="#{label.check}"></th>
          <th></th>
        </tr>
        <!-- 登録済Task -->
        <tr th:each="task,stat:*{taskList}">
          :
        </tr>
        <!-- ①新規タスク入力行追加　ここから ↓ ↓ ↓ -->
        <tr>
          <!-- id -->
          <td></td>
          <!-- 件名 -->
          <td>
```

```
                <input type="text" name="newTask.title" size="40" th:value="*
{newTask.title}">
                <div th:if="${#fields.hasErrors('newTask.title')}" th:errors="*
{newTask.title}"
                  th:errorclass="red"></div>
              </td>
              <!-- 期限 -->
              <td>
                <input type="text" name="newTask.deadline" size="10"
                  th:value="*{newTask.deadline}"    placeholder="yyyy-mm-dd">
                <div th:if="${#fields.hasErrors('newTask.deadline')}"
                  th:errors="*{newTask.deadline}"
                  th:errorclass="red"></div>
              </td>
              <!-- チェック -->
              <td>
                <input type="checkbox" name="newTask.done" value="Y"
                  th:checked="*{newTask.done=='Y'}" />
                <input type="hidden" name="!newTask.done" value="N" />
              </td>
              <!-- 追加ボタン    -->
              <td style="padding: 0px;">
                <button type="submit" th:formaction="@{/task/create}"
th:text="#{button.add}"
                  style="margin: 2px; padding: 2px; width: 4em;"></button>
              </td>
            </tr>
            <!-- ①新規タスク入力行追加    ここまで ↑ ↑ ↑ -->
          </table>
          </div>
        </div><!-- ② 追加 -->
        :
</body>
</html>
```

登録済タスク(タスク更新エリア)に続けて、新規タスク入力行を追加します(①)。ここでのポイントはinput要素のnameが「**newTask.プロパティ名**」(newTask.title, newTask.deadline. newTask.done)になっていることです。こうすると@ModelAttributeでTodoDataオブジェクトへバインドするとき、タスク入力行の各項目が【リスト17-1】で定義したnewTaskのtitle, deadline, doneプロパティへセットされます。

またToDoの新規登録時には、タスクを表示しないようth:ifで制御します(②)。これはToDoを先に作ってからタスクを入力できるようにするためです。

17.2 タスクの入力処理

タスク入力処理の前に、一覧画面でToDoの件名リンクをクリックしたときの処理を変更します。

【リスト17-3】
com.example.todolist.controller.TodoListController#todoById()

```
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int id,
ModelAndView mv) {
        mv.setViewName("todoForm");
        Todo todo = todoRepository.findById(id).get();

        //mv.addObject("todoData", todo);
       //    ↓TodoDataを渡すように変更する
        mv.addObject("todoData", new TodoData(todo));

        session.setAttribute("mode", "update");
        return mv;
    }
```

ここまではクリックされたToDoのidをキーにtodoテーブルを検索した結果(=Todoオブジェクト)を入力画面へ渡していました。しかし【リスト17-2】で新規タスク入力行に対応するinput要素を追加したため、TodoDataオブジェクトが必要です(Todoオブジェクトのままでは、newTaskが無いので実行時エラーになる)。そこでTodoDataクラスにTodoを引数とするコンストラクターを定義し、その結果を入力画面へセットします。

【リスト17-4】com.example.todolist.form.TodoData#TodoData()

```
package com.example.todolist.form;
        :
import java.util.ArrayList;
import lombok.NoArgsConstructor;
        :
```

```
@Data
@NoArgsConstructor   //   ←   追加する
public class TodoData {
    :
    //   Todoの内容から入力画面へ渡すTodoDataを生成する   ここから ↓ ↓ ↓
    public TodoData(Todo todo) {
        // Todo部分
        this.id = todo.getId();
        this.title = todo.getTitle();
        this.importance = todo.getImportance();
        this.urgency = todo.getUrgency();
        this.deadline = Utils.date2str(todo.getDeadline());
        this.done = todo.getDone();

        // 登録済Task
        this.taskList = new ArrayList<>();
        String dt;
        for (Task task : todo.getTaskList()) {
            dt = Utils.date2str(task.getDeadline());
            this.taskList.add(
                new TaskData(task.getId(), task.getTitle(), dt, task.getDone()));
        }

        // 追加用Task
        newTask = new TaskData();
    }
    //   Todoの内容から入力画面へ渡すTodoDataを生成する   ここまで ↑ ↑ ↑
}
```

このとき@NoArgsConstructorも追加し、デフォルトコンストラクターが生成されるようにします(createTodo()でデフォルトコンストラクターを使っているため)。

そして入力画面でタスクの[登録]ボタンをクリックしたとき、タスクを追加します。【リスト17-2】はボタンは以下のように定義しています。

```
<button type="submit" th:formaction="@{/task/create}" th:text="#{button.add}"
```

対応するハンドラーメソッドは次のようにします。

【リスト17-5】
com.example.todolist.controller.TodoListController#createTask()

```
      :
import com.example.todolist.entity.Task;
      :
   // Task追加処理
   @PostMapping("/task/create")
   public String createTask(@ModelAttribute TodoData todoData,
                                    BindingResult result, Model model,
                                    RedirectAttributes redirectAttributes, Locale
locale) {
         // エラーチェック
         boolean isValid = todoService.isValid(todoData.getNewTask(), result,
locale);
         if (isValid) {
               // エラーなし
               Todo todo = todoData.toEntity();
               Task task = todoData.toTaskEntity();    // ①
               task.setTodo(todo);    // ②
               taskRepository.saveAndFlush(task);

               // 追加完了メッセージをセットしてリダイレクト
               String msg = messageSource.getMessage("msg.i.task_created", null,
locale);
               redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
               return "redirect:/todo/" + todo.getId();

         } else {
```

```
            // エラーあり -> エラーメッセージをセット
            String msg =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
            model.addAttribute("msg", new OpMsg("E", msg));
            return "todoForm";
        }
    }
```

【リスト17-6】src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.task_created=Taskを作成しました。
```

【リスト17-7】src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.task_created=Task has been created.
```

入力画面の内容は@ModelAttributeによりtodoDataへバインドされます。ここからTodoData#getNewTask()で新規タスク入力行の内容を取得しチェックします(isValid()については後述)。エラーが無ければtoTaskEntity(後述)でTaskオブジェクトを作成し(①)、saveAndFlush()でtaskテーブルへ追加します。

このとき関連するTodoオブジェクトへの参照を設定していることに注意してください(②)。この参照が無いと、タスクをToDoに関連付けられません。

```
public class Task {
        :
        @ManyToOne
        @JoinColumn(name = "todo_id")
        private Todo todo;
        :
}
```

なおTodoData#toTaskEntity()は、新規タスク入力行(newTask)からTaskオブジェクトを作成して返すものです。

【リスト17-8】com.example.todolist.form.TodoData#toTaskEntity()

```
//   TodoForm画面の新規タスク入力行の内容からTaskオブジェクトを生成して返す
public Task toTaskEntity() {
        Task task = new Task();
        task.setId(newTask.getId());
        task.setTitle(newTask.getTitle());
        task.setDone(newTask.getDone());
```

```
        task.setDeadline(Utils.str2date(newTask.getDeadline()));

        return task;
    }
```

Taskをチェックするis Valid()は次のようになっています。

【リスト17-9】com.example.todolist.service.TodoService#isValid()

```
    // 新規Taskのチェック
    public boolean isValid(TaskData taskData, BindingResult result, Locale locale) {
        boolean ans = true;

            // タスクの件名が半角スペースだけ or "" ならエラー
            if (Utils.isBlank(taskData.getTitle())) {
                FieldError fieldError = new FieldError(
                        result.getObjectName(),
                        "newTask.title",
                        messageSource.getMessage("NotBlank.taskData.title", null,
locale));
                result.addError(fieldError);
                ans = false;

            } else {
                // タスクの件名が全角スペースだけで構成されていたらエラー
                if (Utils.isAllDoubleSpace(taskData.getTitle())) {
                    FieldError fieldError = new FieldError(
                            result.getObjectName(),
                            "newTask.title",
                            messageSource.getMessage(
                                    "DoubleSpace.todoData.title", null, locale));
                    result.addError(fieldError);
                    ans = false;
                }
            }
```

```
        // 期限""ならチェックしない
        String deadline = taskData.getDeadline();
        if (deadline.equals("")) {
            return ans;
        }

        // 期限のyyyy-mm-dd形式チェック
        if (!Utils.isValidDateFormat(deadline)) {
            FieldError fieldError = new FieldError(
                result.getObjectName(),
                "newTask.deadline",
                messageSource.getMessage(
                    "InvalidFormat.todoData.deadline", null, locale));
            result.addError(fieldError);
            ans = false;

        } else {
            // 過去日付ならエラー
            if (!Utils.isTodayOrFurtureDate(deadline)) {
                FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "newTask.deadline",
                    messageSource.getMessage("Past.todoData.deadline", null,
locale));
                result.addError(fieldError);
                ans = false;
            }
        }

        return ans;
    }
```

【リスト17-10】src/main/resources/i18n/ValidationMessages_ja.properties(追加分)

NotBlank.taskData.title=件名を入力してください

【リスト17-11】src/main/resources/i18n/ValidationMessages_en.properties(追加分)

```
NotBlank.taskData.title=Title is required.
```

これで冒頭【図17-1】のように、タスクを追加できるようになりました。

しかしまだ新規にToDoを作成した場合の処理に変更が必要です。まずToDo入力画面のタスク一覧の表示/非表示制御を削除します(【リスト17-2】では削除済みです)。

【リスト17-12】src/main/resources/templates/todoForm.html(変更箇所)

```
<!DOCTYPE html>
  :
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
    ■ToDo
    <!-- ToDo入力エリア -->
      :
    <!-- タスク更新エリア -->
    <div th:if="${session.mode == 'update'}">
      <hr style="margin-top: 2em; margin-bottom: 1em;">
      ■Task
      <!-- <div th:if="${#lists.size(todoData.taskList)} > 0"> -->
        <table>
          :
          <!-- 登録済Task -->
            :
          <!-- 新規タスク入力行 -->
          </tr>
```

```
                :
            </table>
            <!--</div> -->
        </div>
                :
</body>
</html>
```

本章で新規タスク入力行を追加したことにより、タスク一覧は必ず表示されます。よって、この制御は不要です。反対にあると、新規追加したToDoにタスクを入力できなくなります。

またToDoを新規追加した後、(ToDo一覧へ戻るのではなく)入力画面を再表示するようにします(①)。こうすれば、続けて新規追加したToDoのタスクを入力できます。

【リスト17-13】

com.example.todolist.controller.TodoListController#createTodo()

```
    // ToDo追加処理
    @PostMapping("/todo/create/do")
    public String createTodo(@ModelAttribute @Validated TodoData todoData,
                                        BindingResult result,
                                        Model model,
                                        RedirectAttributes redirectAttributes,
                                        Locale locale) {
        // エラーチェック
        boolean isValid = todoService.isValid(todoData, result, true, locale);
        if (!result.hasErrors() && isValid) {
            :
            // 追加完了メッセージをセットしてリダイレクト
            String msg = messageSource.getMessage("msg.i.todo_created", null,
locale);
            redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
            return "redirect:/todo/" + todo.getId();    // ①変更

        } else {
            :
        }
    }
```

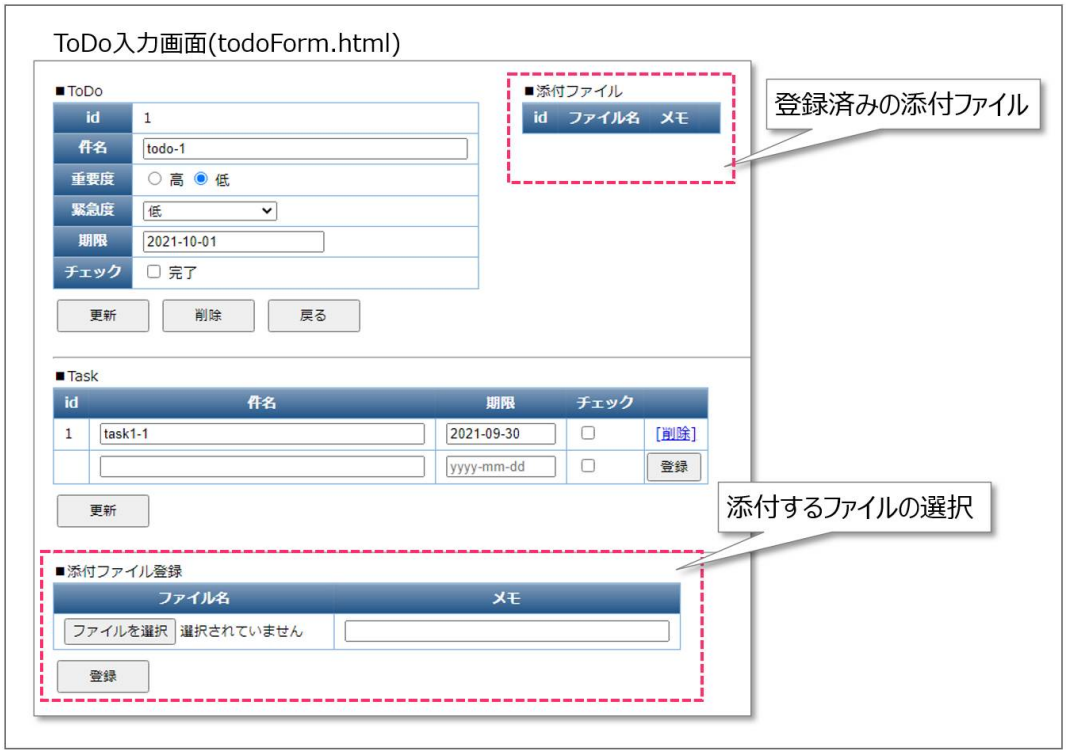
以上Todoとタスクを例に関連エンティティの扱い方について説明してきました。この1:nの関連は実際の業務システムでもよく使われていますので、自分で図を書きながら理解を深めていってください。

18. ファイルのアップロード/ダウンロード

ここまでのToDoアプリは文字(テキスト)情報だけでしたが、ファイルも扱えるようにすると便利かもしれません。例えば「新機能Xのリリース」というToDoには、仕様書や日程表(WBS)などのファイルも関連するでしょう。そういったものをToDoに添付する、というアイデアです。この添付のように、ファイルをサーバーへ送信することを「**アップロード**」と言います。反対の取り出し操作は「**ダウンロード**」です。本章ではこれらの機能をToDoアプリへ追加します。

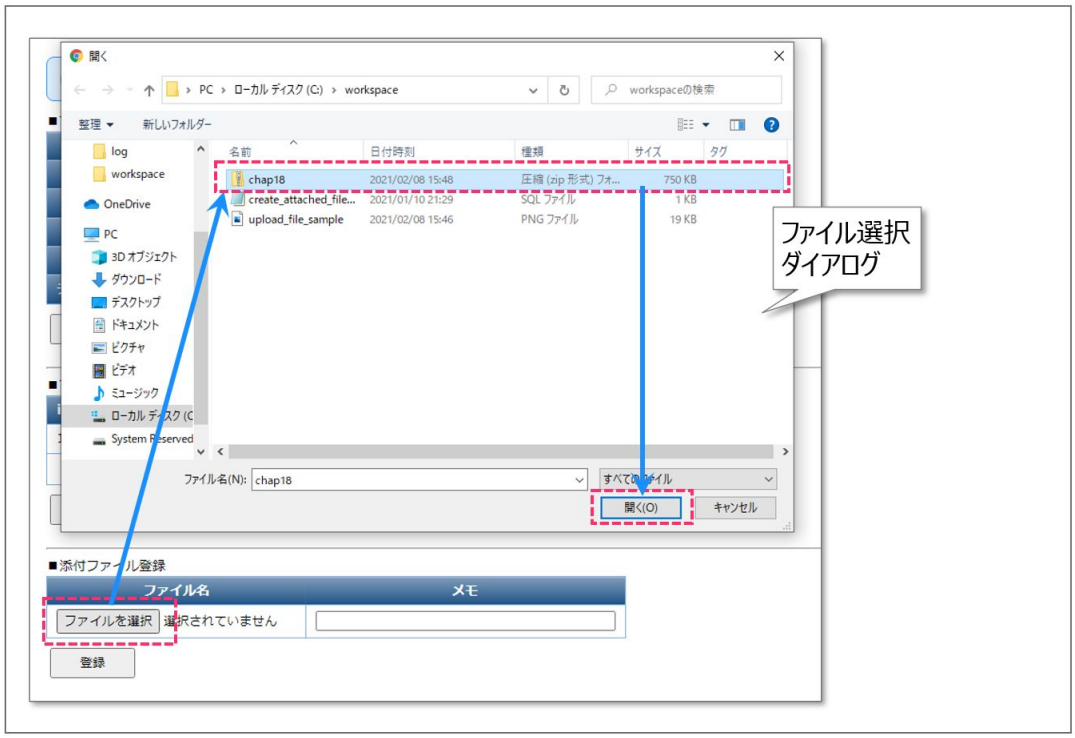
ToDoと添付ファイルの関係は1:nとします(1つのToDoに0個以上の添付ファイルが存在する)。ただしアノテーション(@ManyToOne,@OneToMany)を使わず、プログラムコード(ロジック)で関連を表します。

アップロード機能/ダウンロード機能は、ToDo入力画面(todoForm.html)に設けます。上述したようにToDoと添付ファイルは1:nのため、ここで操作するのが自然だと思います。下図のようにアップロードは画面下部で行い、画面右上に添付されたファイルの一覧を表示します。



【図18-1】添付ファイルに関連する項目

アップロードの操作は、まず[ファイルを選択]ボタンをクリックします。するとダイアログが表示されるので、ファイルを選択し[開く(O)]ボタンをクリックします。

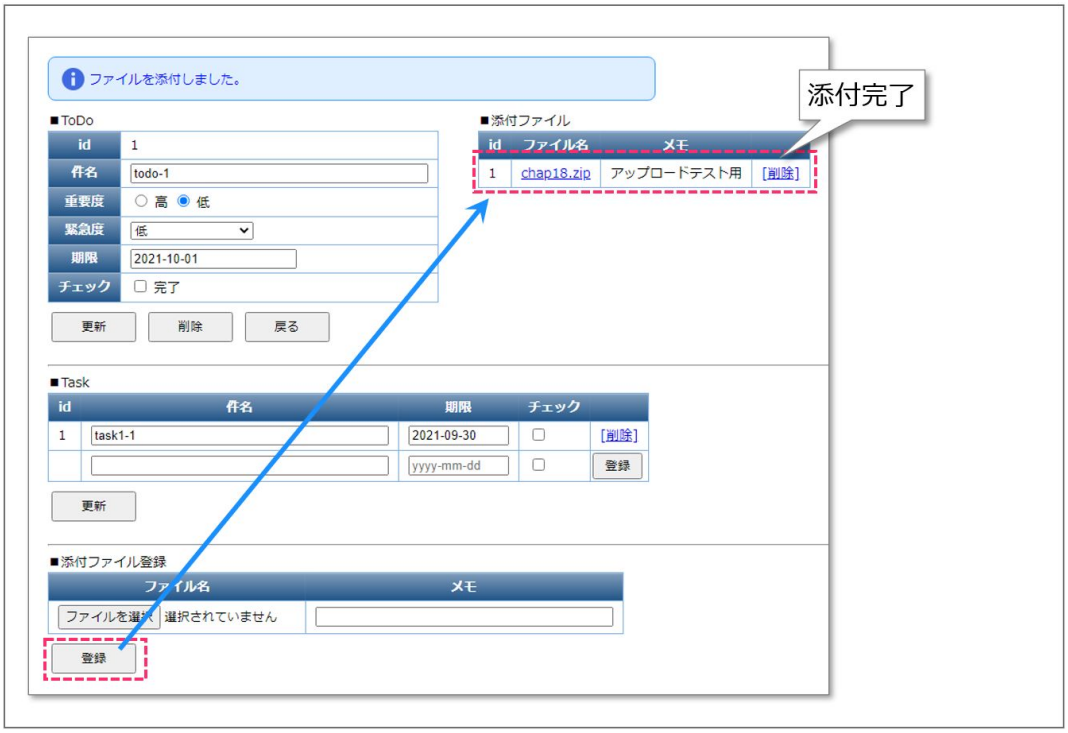


【図18-2】添付するファイルの選択

この段階では、まだアップロードしていません。選択したファイル名がブラウザに表示されるだけです。

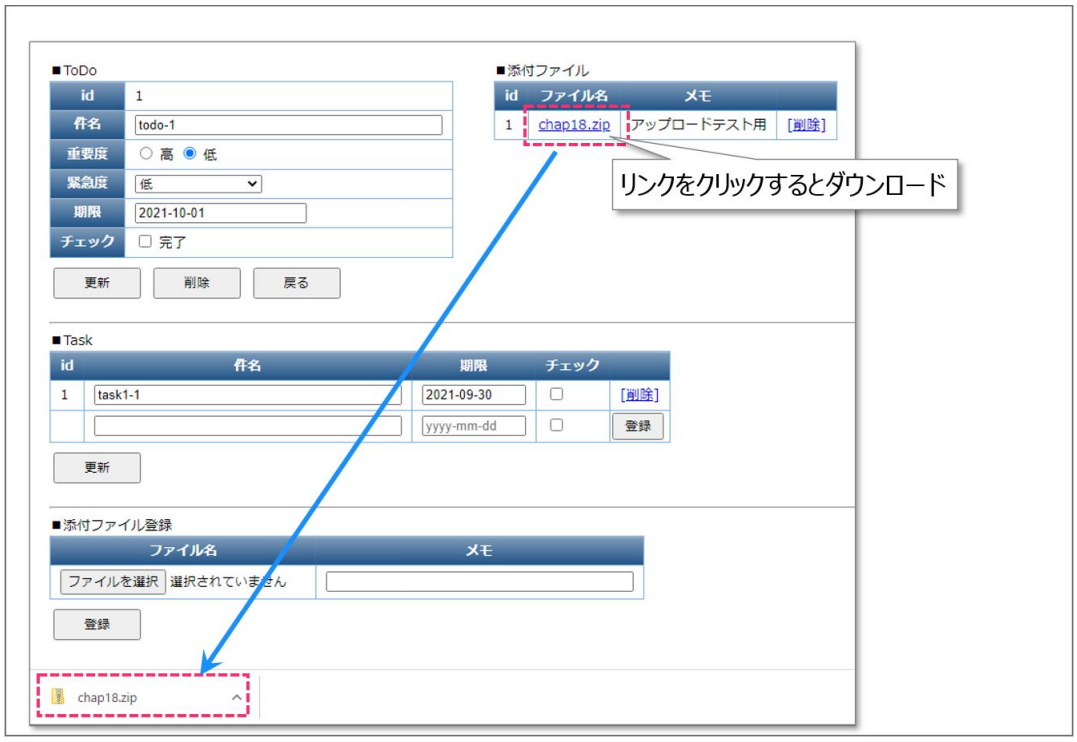
【図18-3】添付するファイルを選択した状態

次に[登録]ボタンをクリックすると、選択したファイルがサーバーへ送られます。サーバーはそれを格納し、入力画面を再表示します。このとき、添付ファイル一覧も更新されます。



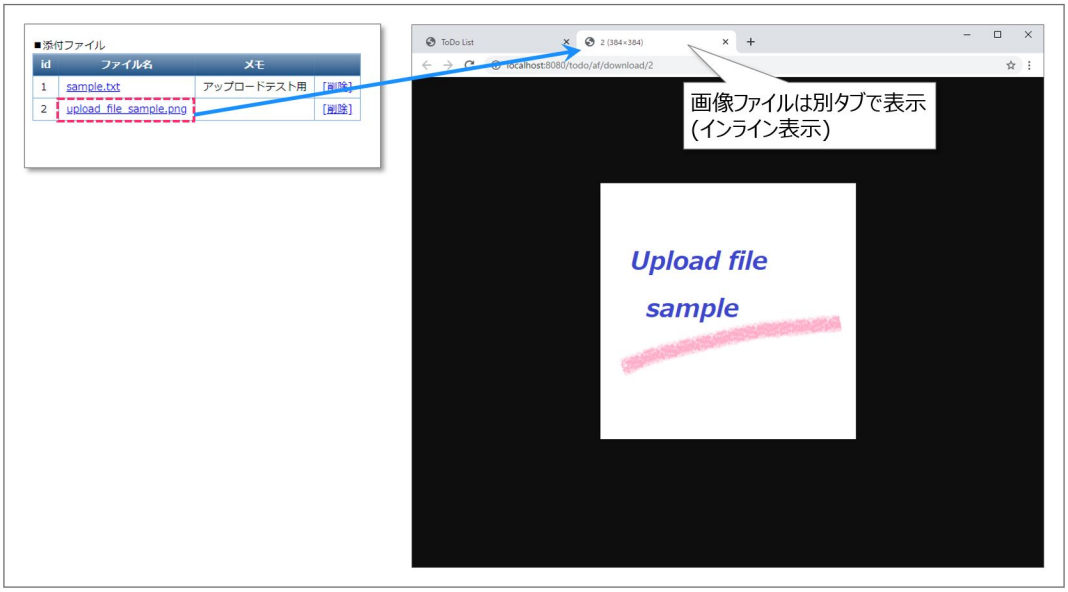
【図18-4】ファイルをアップロード(添付)

この一覧のファイル名(リンク)をクリックすると、そのファイルがサーバーからブラウザへダウンロードされます。



【図18-5】ファイルのダウンロード

ただしToDoアプリでは、PNGなどの画像ファイルは、ローカルに保存せず別タブに表示させます。

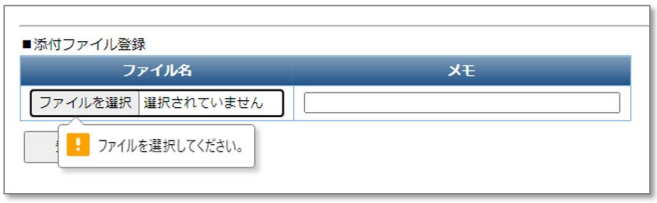


【図18-6】画像ファイルのインライン表示

また削除リンクをクリックすると、そのファイルが削除されます。

【図18-7】添付ファイルの削除

なおファイルを選択せず[登録]ボタンをクリックすると、エラーメッセージが表示されます。このチェックはブラウザ標準の機能を利用して行います。



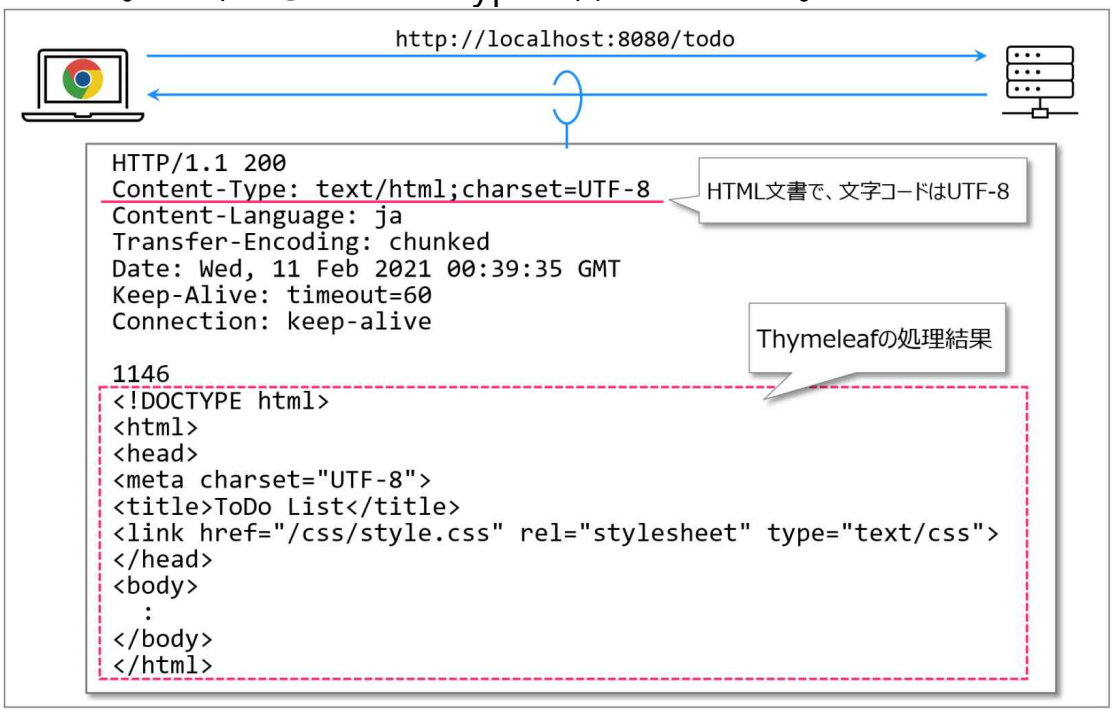
【図18-8】ファイルを選択しなかった場合

18.1 アップロードの仕組み

本節ではファイルをアップロードするのに必要な知識(Content-Type,multipart/form-dataなど)を説明した後、実際にToDoアプリへコードを追加していきます。

18.1.1 Content-Type

１つ目のポイントは「**Content-Type**」です。これはインターネットの世界で「**データの種類**」を表すために使われている符号のようなものです。例えばブラウザからhttp://localhost:8080/todoをアクセスすると、ToDoアプリは以下のようなデータを返します。この中にもContent-Typeが含まれています。



【図18-9】レスポンスに含まれるContent-Typeの例

２行目にあるContent-Typeは「このデータはhtml形式のテキストで、文字コードはUTF-8です」ということをブラウザに知らせるものです。ブラウザはこれを元にデータを表示(レンダリング)します。通常Content-Typeの設定は、ブラウザ～サーバー間で自動的に行われますが、ファイルのアップロード/ダウンロードでは、プログラマーが適切に行う必要があります。

上記Content-Typeの「text/html」の部分は、「**MIME(マイム)Type**」と呼ばれています。これがデータ種別を表しており、下記サイトを見ると膨大な数あることがわかります。

■Media Types

https://www.iana.org/assignments/media-types/media-types.xhtml#application

このうち本書に関連するのは下表のMIME Typeです。

【表18-1】MIME Type例

MIME Type	データ種別
text/plain	(プレーン)テキスト
text/html	HTML文書
text/css	CSS
image/gif	GIF画像
image/jpeg	JPEG画像
image/png	PNG画像
application/pdf	PDFドキュメント
multipart/form-data	複合データ型

アップロードと関係が深いのは、最後の「**multipart/form-data**」です。これは「複数個のデータ(multipart)から構成されるデータ」という意味です。次節で詳しく見ていきます。

18.1.2 multipart/form-data

【図18-1】などで示した入力画面の添付ファイル選択部分は、次のようになっています。

【リスト18-1】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
  :
<body>
```

```
    <!-- メッセージエリア -->
    <div th:replace="fragments :: msg_area"></div>
    <!-- 入力フォーム -->
    <form th:action="@{/}" method="post" th:object="${todoData}">
    :
    </form>
    <!-- 添付ファイル登録エリア追加　ここから ↓ ↓ ↓ -->
    <!-- 更新の場合、添付ファイル登録エリアを表示する -->
    <div th:if="${session.mode == 'update'}">
      <hr style="margin-top: 1em;">
      <span th:text="#{text.regist_attachedfile}"></span>
      <!-- ① -->
      <form th:action="@{/todo/af/upload}"
enctype="multipart/form-data" method="post">
        <table>
          <!-- 見出し行 -->
          <tr>
            <!-- ファイル -->
            <th th:text="#{label.filename}"></th>
            <!-- メモ -->
            <th th:text="#{label.note}"></th>
          </tr>
          <!-- 入力行 -->
          <tr>
            <!-- ファイル -->
            <td>
              <input type="file" name="file_contents" required><!-
- ②③ -->
            </td>
```

```
                <!-- メモ -->
                <td>
                    <input type="text" name="note" size="40">
                </td>
            </tr>
        </table>
        <!-- 登録ボタン -->
        <button type="submit" th:text="#{button.add}"></button>
        <!-- 添付先Todoのid -->
        <input type="hidden" name="todo_id"
th:value="${todoData.id}">
    </form>
  </div>
  <!-- 添付ファイル登録エリア追加　ここまで ↑ ↑ ↑ -->
</body>
</html>
```

【リスト18-2】

src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
label.filename=ファイル名
label.note=メモ
text.attachedfiles=■添付ファイル
text.regist_attachedfile=■添付ファイル登録
```

※text.regist_attachedfile以外は、本章後半で使います。

【リスト18-3】

src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
label.filename=File name
label.note=Note
text.attachedfiles=¥u25A0Attached files
text.regist_attachedfile=¥u25A0Upload attached file
```

※¥u25A0は■のUnicode表記

新しく出てきた属性は以下の 3 種類です。

①enctype="multipart/form-data"

・このform要素へ入力されたデータをサーバーへ送信する形式を表す属性

・ファイルアップロードでは、前述したmultipart/form-dataを指定する。

(省略時はapplication/x-www-form-urlencodedとなり、「名前=値」の形式になる)

②type="file"

・ファイル選択ボタンとファイル名フィールドを作成する。

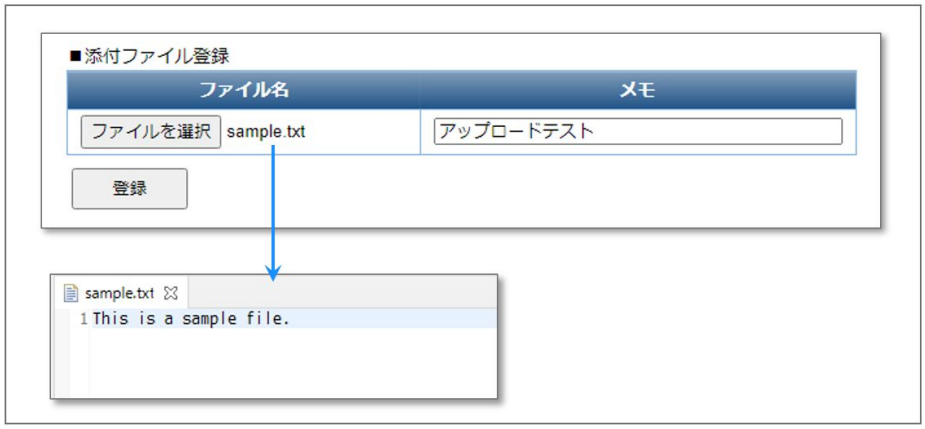
・①と組み合わせることで、選択したファイルをサーバーへ送信(アップロード)する。

③required

・入力必須フィールドであることを表す属性

・フォーム送信時、入力されていないと【図18-8】のようにエラーメッセージが表示される(送信は中止される)。

ここで下図のようにsample.txtを選択し、メモ欄に「アップロードテスト」と入力後、[登録]ボタンをクリックしたとします。



【図18-10】ファイル添付の操作例

このときサーバーには以下のようなデータが送信されます(一部省略しています)。

```
POST http://localhost:8080/todo/af/upload HTTP/1.1
Host: localhost:8080
 :
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryacEOktGeKBC0Gkep
 :
Cookie: JSESSIONID=...

------WebKitFormBoundaryacEOktGeKBC0Gkep
Content-Disposition: form-data; name="file_contents";filename="sample.txt"
Content-Type: text/plain

This is a sample file.
------WebKitFormBoundaryacEOktGeKBC0Gkep
Content-Disposition: form-data; name="note"

アップロードテスト用
------WebKitFormBoundaryacEOktGeKBC0Gkep
Content-Disposition: form-data; name="todo_id"

1
------WebKitFormBoundaryacEOktGeKBC0Gkep--
```

MIME Type
区切り文字列
file_contents欄
note欄
todo_id欄

【図18-11】[登録]ボタンクリックでサーバーへ送信されるデータ例

Content-TypeのMIME Typeは前述①のmultipart/form-dataです。続けて"boundary="という部分があります。boundaryは「境界」という意味で、データの境目に"----WebKitFormBoundaryacEOktGeKBC0Gkep"という文字列があることを表しています(この文字列は毎回変わります)。これで区切られた上図の青点線部分をサブパート(sub part)と言います。

各サブパートは、格納しているデータの情報を表す"Content-Disposition:～"という行で始まります。アップロードファイルの場合、以下のような形式になります(上図最初のサブパート)。

```
Content-Disposition: form-
data;name="file_contents";filename="sample.txt"
Content-Type: text/plain

This is a sample file.
```

- **form-data**
  - multipart/form-data中のContent-Dispositionは、常にform-dataから始まる。
- **name**
  - このサブパートに対応するHTMLのフィールド名
  - "file_contents"はファイル選択ボタンのname属性で指定したもの(【リスト18-1】)。
- **filename**
  - アップロードするファイル名(【図18-10】)
  - input要素がtype="file"の場合付加される。
- **Content-Type: text/plain**
  - データの種別(アップロードするファイル(sample.txt)から設定される)
    →PNGファイルの場合はimage/png、のように変わる。
- **This is a sample file.**
  - アップロードするファイルの内容(【図18-10】)
  - PNGファイルなどはバイナリ形式となる。

残りのサブパートはメモ欄とToDoのid(type="hidden"のため非表示)です。これもフィールドの内容がセットされています。このようにform要素でenctype="multipart/format-data"を指定すると、ファイルやフィール

ドなど異なる種類のデータをひとまとめにしてサーバーへ送ることができます。

18.1.3 サーバー側の処理

サーバーは前節のmultipart/form-data形式で送信されたデータから、ファイルやフィールドの値を取り出します。以下は【図18-10】の[登録]ボタンに対応するハンドラーメソッドuploadAttachedFile()です。これが【図18-11】のデータを受け取るわけですが、メソッドの引数定義は一見すると今までと同じです。これはmultipart/form-dataを特別扱いする必要がない、ということを表しています(uploadAttachedFile()の全体は【リスト18-8】で解説します)。

■com.example.todolist.controller.TodoListController#uploadAttachedFile()

```
    // 添付ファイルをアップロードする
    @PostMapping("/todo/af/upload")
    public String uploadAttachedFile(
                            @RequestParam("todo_id") int todoId,
                            @RequestParam("note") String note,
                            @RequestParam("file_contents") MultipartFile
fileContents,    // ①
                            RedirectAttributes redirectAttributes,
                            Locale locale) {
                        :
    }
```

ポイントは①の**MultipartFile**型の引数です。これで送信されたファイルの内容が、引数fileContentsへバインドされます。("file_contents"は

ファイル選択フィールドの名前(【リスト18-1】参照))。todo_idやnoteと同じようにファイルも@RequestParamで扱えるわけです。

> 以下のようにフォームクラスを定義し@ModelAttributeでバインドすることも可能です。こちらは同時にバリデーションも定義できます。
> ■フォームクラス定義例

```
package com.example.todolist.form;

import org.springframework.web.multipart.MultipartFile;
import lombok.Data;

@Data
public class UploadData {
	Integer todo_id;
	String note;
	MultipartFile file_contents;
}
```

> ■@ModelAttribute使用例

```
public String uploadAttachedFile(@ModelAttribute UploadData uploadData,
                                 RedirectAttributes redirectAttributes,
                                 Locale locale) {
```

取得したファイルは、一般的に以下のような方法で保存します。

(1)サーバーからアクセスできるフォルダに書き込む

(2)データベースのテーブルに格納する(BLOB型)

どちらも一長一短ありますが、ToDoアプリでは実行結果が確認しやすい(1)とします。ただし添付ファイルを管理するためのテーブルを1つ追加します。

作成するプロジェクトの仕様

プロジェクト名	Todolist11
作成ファイル	com.example.todolist.controller.DownloadController.java com.example.todolist.entity.AttachedFile.java com.example.todolist.form.AttachedFileData.java com.example.todolist.repository.AttachedFileRepository.java

```
Todolist11
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist11Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java(▲)
 │    ├ com.example.todolist.controller
 │    │    ├ DownloadController.java(★)
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
```

```
│   │   ├ AttachedFile.java(★)
│   │   ├ Task.java
│   │   └ Todo.java
│   ├ com.example.todolist.form
│   │   ├ AttachedFileData.java(★)
│   │   ├ TaskData.java
│   │   ├ TodoData.java(▲)
│   │   └ TodoQuery.java
│   ├ com.example.todolist.repository
│   │   ├ AttachedFileRepository.java(★)
│   │   ├ TaskRepository.java
│   │   └ TodoRepository.java
│   └ com.example.todolist.service
│       ├ DownloadService.java(★)
│       └ TodoService.java(▲)
└ src/main/resources
    ├ i18n
    │   ├ FixedDisplayStrings.properties
    │   ├ FixedDisplayStrings_en.properties(▲)
    │   ├ FixedDisplayStrings_ja.properties(▲)
    │   ├ OperationMessages_en.properties(▲)
    │   ├ OperationMessages_ja.properties(▲)
    │   ├ ValidationMessages_en.properties
    │   └ ValidationMessages_ja.properties
    :
    ├ templates
    │   ├ fragments.html
    │   ├ todoForm.html(▲)
    │   └ todoList.html
```

└application.properties(▲)

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

18.2 添付ファイルの管理

18.2.1 定義の追加

まずapplication.propertiesにファイルの格納場所や、ファイルの最大サイズなどを定義します。

【リスト18-4】src/main/resources/application.properties(追加分)

```
#Upload file save dir
attached.file.path=C:/temp/uploadFiles

#File upload limits
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=100MB
```

※attached.file.pathに警告マークが付きますが、そのままで大丈夫です。

- **attached.file.path**
  - アップロードしたファイルを格納するフォルダ名
    →この定義はToDoアプリ独自のもの。よってキー名称は、他と重複しないかぎり自由。
- **spring.servlet.multipart.max-file-size**
  - アップロードするファイルの最大サイズ(=1ファイル当たりの最大サイズ)
  - 省略時(デフォルト)1MB
- **spring.servlet.multipart.max-request-size**
  - multipart/format-data全体の最大サイズ(=全サブパートを合計した最大サイズ)
  - 省略時(デフォルト)10MB

次にアップロードされた添付ファイルを管理するattached_fileテーブルを追加します。

【リスト18-5】create_attached_file.sql

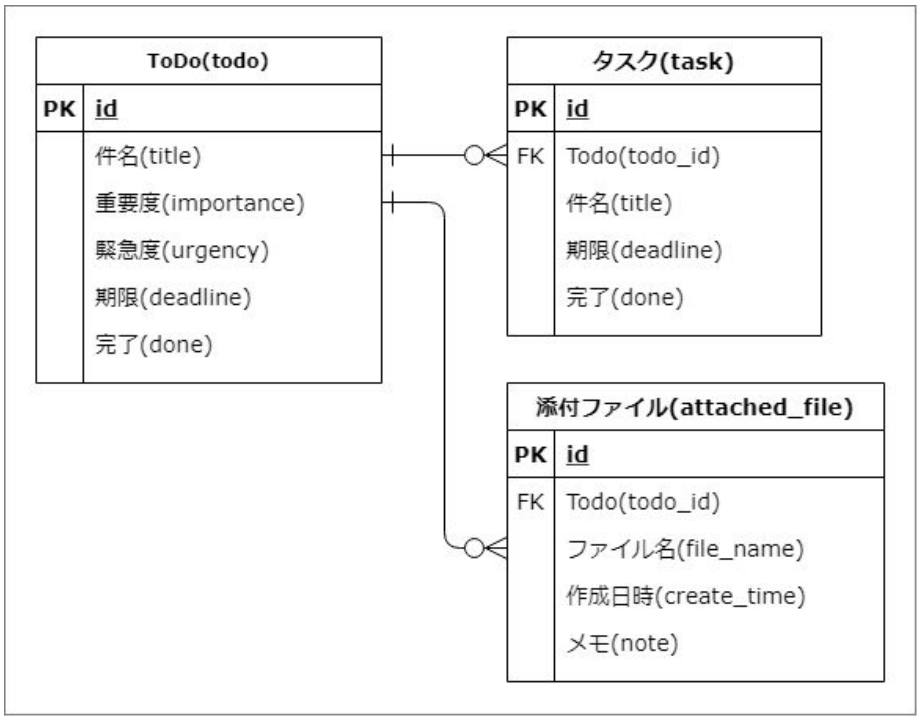
```
DROP TABLE attached_file;
CREATE TABLE attached_file(
        id                  SERIAL PRIMARY KEY,
        todo_id             INTEGER,
        file_name           TEXT,
        create_time         TEXT,
        note                TEXT
);
```

ポイントはtodo_id列です。ここに「この添付ファイルはどのToDoのものか？」を表すため、todo.idの値を格納します。

【表18-2】attached_fileテーブルの形式

列名	内容	データ型	制約	備考
id	1～	SERIAL	PRIMARY KEY	連番(自動採番)
todo_id	todo.id	INTEGER		添付先ToDoのid
file_name	添付ファイル名	TEXT		
create_time	作成日時	DATE		アップロード日時
note	メモ	TEXT		

ER図で表すと次のような関係になります。



【図18-12】本章テーブルのER図

以下が対応するエンティティクラスです。

【リスト18-6】com.example.todolist.entity.AttachedFile.java

```
package com.example.todolist.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "attached_file")
@Data
@NoArgsConstructor
public class AttachedFile {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Integer id;

	@Column(name = "todo_id")
	private Integer todoId;    // ①

	@Column(name = "file_name")
	private String fileName;
```

```
    @Column(name = "create_time")
    private String createTime;

    @Column(name = "note")
    private String note;

}
```

Taskエンティティでは、todo_id列に以下のようなプロパティを対応させていました。

```
    @ManyToOne
    @JoinColumn(name = "todo_id")
    private Todo todo;
```

しかしAttachedFileでは単なるInteger型(=todo.idの型)です(①)。これはアノテーションを使わず、プログラムコードでToDoとの関連付けを行う、という意味です。

リポジトリにはtodoIdをキーにして検索するメソッドを宣言します。
→todoIdは主キー(=@Idが付与されたプロパティ)でないため必要(前書「9.3 検索処理の定義・実行」参照)

【リスト18-7】
com.example.todolist.repository.AttachedFileRepository.java

```
package com.example.todolist.repository;

import java.util.List;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.AttachedFile;

@Repository
public interface AttachedFileRepository extends
JpaRepository<AttachedFile, Integer> {
	// todoIdをキーに検索する(idの昇順)
	List<AttachedFile> findByTodoIdOrderById(Integer todoId);
}
```

18.2.2 添付ファイルの格納処理

次に[登録]ボタンに対するハンドラーメソッドを追加します。

ここではアップロードされたファイル(①)が空(=0byte)でないことを確認(②)した上で、フォルダへ保存(③)します。保存処理は少々複雑なのでサービスクラスTodoServiceのsaveAttachedFile()で行います。

【リスト18-8】

com.example.todolist.controller.TodoListController#uploadAttachedFile()

```
        :
import org.springframework.web.multipart.MultipartFile;
        :
    // 添付ファイルをアップロードする
    @PostMapping("/todo/af/upload")
    public String uploadAttachedFile(
                                @RequestParam("todo_id") int todoId,
                                @RequestParam("note") String note,
                                @RequestParam("file_contents") MultipartFile
fileContents,    // ①
                                RedirectAttributes redirectAttributes,
                                Locale locale) {
            // ファイルが空？
            if (fileContents.isEmpty()) {    // ②
                // ファイルemptyのメッセージをセット
                String msg =
messageSource.getMessage("msg.w.attachedfile_empty", null,
locale);
```

```
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));

        } else {
            // ファイルを保存する
            todoService.saveAttachedFile(todoId, note, fileContents);
// ③
            // Upload完了メッセージをセット
            String msg =
messageSource.getMessage("msg.i.attachedfile_uploaded", null,
locale);
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("I", msg));
        }
        // 再表示
        return "redirect:/todo/" + todoId;
    }
```

【リスト18-9】
src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.attachedfile_uploaded=ファイルを添付しました。
msg.w.attachedfile_empty=指定されたファイルは空です。
```

【リスト18-10】
src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.attachedfile_uploaded=The file has been attached.
```

msg.w.attachedfile_empty=The specified file is empty.

TodoService#saveAttachedFile()は以下のようになっています。アップロードされたファイルをサーバー上に格納するとともに、その情報をattached_fileテーブルへ登録します。

【リスト18-11】
com.example.todolist.service.TodoService#saveAttachedFile()

```
package com.example.todolist.service;
   :
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.multipart.MultipartFile;
import com.example.todolist.entity.AttachedFile;
import com.example.todolist.repository.AttachedFileRepository;
import lombok.RequiredArgsConstructor;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
   :
@Service
@RequiredArgsConstructor // ④
public class TodoService {
      private final MessageSource messageSource;
      private final AttachedFileRepository attachedFileRepository;
// ①

      @Value("${attached.file.path}")    // ②
```

```
    private String ATTACHED_FILE_PATH;   // ③
    :
    public void saveAttachedFile(int todoId, String note,
MultipartFile fileContents) {
        // アップロード元ファイル名
        String fileName = fileContents.getOriginalFilename();   //
⑤

        // 格納フォルダの存在チェック
        File uploadDir = new File(ATTACHED_FILE_PATH);
        if (!uploadDir.exists()) {
            // ないのでディレクトリ作成
            uploadDir.mkdirs();
        }

        // 添付ファイルの格納(upload)時刻を取得
        SimpleDateFormat sdf = new
SimpleDateFormat("yyyyMMddHHmmssSSS");
        String createTime = sdf.format(new Date());

        // テーブルへ格納するインスタンス作成
        AttachedFile af = new AttachedFile();
        af.setTodoId(todoId);
        af.setFileName(fileName);
        af.setCreateTime(createTime);
        af.setNote(note);

        // アップロードファイルの内容を取得
        byte[] contents;
```

```
            try (BufferedOutputStream bos
                    = new BufferedOutputStream(
                            new FileOutputStream(

Utils.makeAttahcedFilePath(ATTACHED_FILE_PATH, af)
            ))) {
                // アップロードファイルを書き込む
                contents = fileContents.getBytes();    // ⑥
                bos.write(contents);

                // テーブルへ記録
                attachedFileRepository.saveAndFlush(af);    // ⑦

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

saveAttachedFile()にはattached_fileテーブルを操作するのでリポジトリ(【リスト18-7】)が必要です(①)。そしてapplication.properties(【リスト18-4】)に定義したアップロードファイルの格納先フォルダ名を**@Valueアノテーション**でATTACHED_FILE_PATHへ設定します(②③)。このように${}内にキー名称を記述すると、application.propertiesの対応する値が次の行の変数にセットされます。

```
@RequiredArgsConstructor // ④
public class TodoService {
    private final MessageSource messageSource;
```

```
    private final AttachedFileRepository attachedFileRepository;
// ①

    @Value("${attaced.file.path}")    // ②
    private String ATTACHED_FILE_PATH;    // ③ ←
"C:/temp/uploadFiles"が設定される
```

なおTodoServiceに付与するアノテーションは@AllArgsConstructorから@RequiredArgsConstructorへ変更します(④)。そうしないと以下のようなエラーメッセージが表示され、アプリが起動できません。

```
***************************
APPLICATION FAILED TO START
***************************

Description:
Parameter 2 of constructor in com.example.todolist.service.TodoService
required a bean of type 'java.lang.String' that could not be found.

Action:
Consider defining a bean of type 'java.lang.String' in your configuration.
```

これは@AllArgsConstructorが全フィールドを対象するコンストラクタを生成するため、追加したATTACHED_FILE_PATHにもインジェクションしようとするからです(String型にはインジェクションできない)。
@RequiredArgsConstructorならfinalのフィールドを対象とするため、このエラーを回避できます。

そして必要な情報をMultipartFile型の引数fileContentsから取得していきます。まずアップロードされたファイルの名前(【図18-10】の場合sample.txt)をgetOriginalFilename()で取得します(⑤)。

```
    // アップロード元ファイル名
    String fileName = fileContents.getOriginalFilename();    // ⑤
```

ファイルの内容はgetBytes()で取得し、BufferedOutputStreamで出力します。(⑥)

```
    // アップロードファイルを書き込む
    contents = fileContents.getBytes();    // ⑥
    bos.write(contents);
```

格納先ファイル名はcom.example.todolist.common.Utilsに追加した以下のメソッドで作成します。

【リスト18-12】
com.example.todolist.common.Utils#makeAttahcedFilePath()

```
        :
import com.example.todolist.entity.AttachedFile;
        :
    // 添付ファイルの格納ファイル名を作成する
    public static String makeAttahcedFilePath(String path,
AttachedFile af) {
            return path + "/" + af.getCreateTime() + "_" +
af.getFileName();
    }
```

このメソッドは、例えば

ATTACHED_FILE_PATH=C:/temp/uploadFiles(application.properties のattaced.file.path)
現在時刻=2020/2/11 16:48:16.456
アップロードファイル名=sample.txt

なら、以下をファイル名として返します。

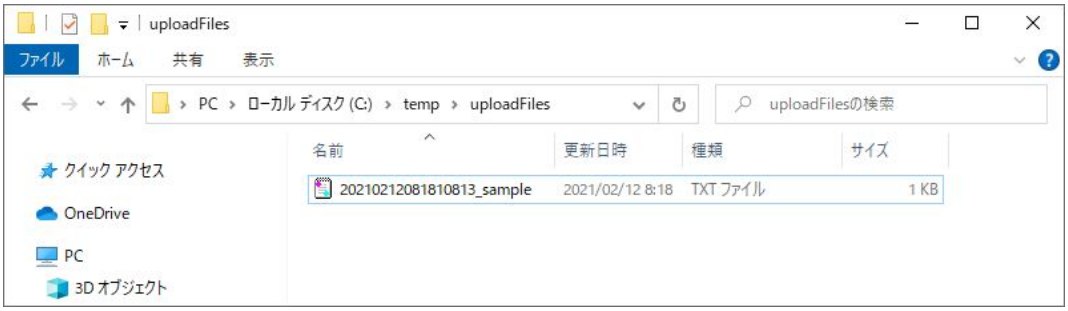
C:/temp/uploadFiles/20200201116481 6456_sample.txt

格納ファイル名に現在時刻を含めているのは、同名のファイルがアップロードされても別ファイルとして扱いたいためです。sample.txtのままでは、他の人が別なsample.txtをアップロードしたら、そのファイルで上書きされてしまいます。そこで現在時刻を付加してファイル名が同じにならないようにします。

最後に格納したファイルの情報をattached_fileテーブルへ追加します(⑦)。

```
// テーブルへ記録
attachedFileRepository.saveAndFlush(af);    // ⑦
```

ここまででファイルをアップロードできるようになります。



またattached_fileテーブルには、この情報が記録されます。

```
tododb=> SELECT * FROM attached_file;
 id | todo_id | file_name  |    create_time    |         note
----+---------+------------+-------------------+----------------------
  1 |       1 | sample.txt | 20210212081810813 | アップロードテスト用
(1 行)


tododb=>
```

> ToDoアプリではストリーム/ライターを使ったファイル入出力を行っていますが、
> 特に解説はしません。必要な方は、他のJava専門書などを参照してください。

18.3 ダウンロードの仕組み

次は添付ファイルを入力画面へ一覧表示し、ダウンロードできるようにします。

18.3.1 添付ファイルの表示

前述のハンドラーメソッドuploadAttachedFile(【リスト18-8】)は、最後に入力画面へリダイレクトしています。

```
    // 再表示
    return "redirect:/todo/" + todoId;
```

この結果実行されるtodoById()には、添付ファイル情報取得処理を追加します(と言っても実質的には下記①の1行だけです)

【リスト18-13】
com.example.todolist.controller.TodoListController#todoById()

```
package com.example.todolist.controller;
:
import java.util.List;
import com.example.todolist.entity.AttachedFile;
import com.example.todolist.repository.AttachedFileRepository;
:
public class TodoListController {
    :
    private final AttachedFileRepository attachedFileRepository; // ②
    :
```

```
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int
id, ModelAndView mv) {
        mv.setViewName("todoForm");
        // ToDo取得(Todo + Task)
        Todo todo = todoRepository.findById(id).get(); // ★
        // ①添付ファイル情報取得
        List<AttachedFile> attachedFiles =
attachedFileRepository.findByTodoIdOrderById(id);
        // 表示用データ作成
        mv.addObject("todoData", new TodoData(todo,
attachedFiles)); // ③
        session.setAttribute("mode", "update");
        return mv;
    }
    :
}
```

Taskはアノテーションで Todoと関連付けているので、Todoと一緒に取得できます(★)。しかしAttachedFile(=attached_fileテーブル)は、リポジトリ(【リスト18-7】)で宣言したメソッドを使い検索する必要があります。(②)。この引数はファイルを添付したTodoのidです。

そしてTodoと添付ファイルの情報からTodoDataオブジェクトを作成して、ToDo入力画面に渡します(③)。これは「15.1 フォームクラスの追加」で説明したように、ToDo入力画面のデータはTodoDataオブジェクトで表しているためです。添付ファイルを追加するなら、対応するプロパティが必要です。

そこでまず添付ファイル 1 件分のデータを表すフォームクラスAttachedFileDataを作成します。プロパティは画面表示で使うものです。このうちopenInNewTabは、添付ファイルを別タブに表示するか(=true)、ローカルに保存するか(=false)のフラグとします。

【リスト18-14】

com.example.todolist.form.AttachedFileData.java

```
package com.example.todolist.form;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class AttachedFileData {
	//　id
	private Integer id;
	//　アップロードしたときのファイル名
	private String fileName;
	//　メモ
	private String note;
	//　新しいタブで開くかどうか
	private boolean openInNewTab;
}
```

添付ファイルはタスクと同じようにToDoと1:nの関係なのでTodoDataにList<AttachedFileData>型のプロパティを追加し(①)、設定するデータはList<AttachedFile>型オブジェクトとしてコンストラクタで受け取ります(②)。

【リスト18-15】
com.example.todolist.form.TodoData#TodoData()

```
package com.example.todolist.form;
    :
import com.example.todolist.entity.AttachedFile;

public class TodoData {
    :
    // 添付ファイルのList
    private List<AttachedFileData> attachedFileList;   // ①
    :
    // TodoとAttachedFileから入力画面へ渡すTodoDataを作成する。
    public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
// ②
        // Todo部分
        :
        // 登録済Task
        :
        // 追加用Task
        :
        // ③添付ファイル
        attachedFileList = new ArrayList<>();
        String fileName;
        String fext;
        String contentType;
        boolean isOpenNewWindow;
        for (AttachedFile af : attachedFiles) {
            // ファイル名
            fileName = af.getFileName();
```

```
                // 拡張子
                fext = fileName.substring(fileName.lastIndexOf(".") +
1);
                // Content-Type
                contentType = Utils.ext2contentType(fext);    // ④
                // 別Windowで表示するか？
                isOpenNewWindow = contentType.equals("") ? false :
true;
                attachedFileList.add(
                        new AttachedFileData(af.getId(), fileName,
af.getNote(), isOpenNewWindow));
            }
        }
}
```

そしてattachedFilesの要素からAttachedFileDataオブジェクトを作成し、表示用のattachedFileListプロパティへ追加します(③)。このうち④のUtils.ext2contentType()は、ファイルの拡張子から対応するContent-Type(MIME Type)を求めるものです。

【リスト18-16】
com.example.todolist.common.Utils#ext2contentType()

```
    public static String ext2contentType(String ext) {
        String contentType;
        if (ext == null) {
            return "";
        }

        switch (ext.toLowerCase()) {
```

```
                // GIF
                case "gif":
                        contentType = "image/gif";
                        break;
                // JPEG
                case "jpg":
                case "jpeg":
                        contentType = "image/jpeg";
                        break;
                // PNG
                case "png":
                        contentType = "image/png";
                        break;
                // PDF
                case "pdf":
                        contentType = "application/pdf";
                        break;
                // 上記以外
                default:
                        contentType = "";
                        break;
        }
        return contentType;
    }
```

このメソッドは引数で渡された拡張子が"gif","jpg","jpeg","png","pdf"のいずれかなら、対応するContent-Type(【表18-1】)、それ以外は""を返すので、この結果からisOpenNewWindowを設定します。

```
    // Content-Type
```

```
    contentType = Utils.ext2contentType(fext);    // ④
    // 別Windowで表示するか？
    isOpenNewWindow = contentType.equals("") ? false : true;
```

入力画面ではこれらの内容からファイル一覧を作成します。

【リスト18-17】src/main/resources/templates/todoForm.html(一部抜粋)

```
<!DOCTYPE html>
:
<body>
    <!-- メッセージエリア -->
    <div th:replace="fragments :: msg_area"></div>
    <!-- 入力フォーム -->
    <form th:action="@{/}" method="post" th:object="${todoData}">
        <div style="display: flex"><!-- ① -->
            <div><!-- ② -->
                ■ToDo
                <!-- ToDO入力エリア -->
                <table>
                :
                </table>
            </div><!-- ② -->
            <!-- ③更新の場合、添付ファイル一覧を表示する ここから ↓ ↓ ↓ -
->
            <div th:if="${session.mode == 'update'}">
                <div style="margin-left: 3em;">
                    <span th:text="#{text.attachedfiles}"></span>
                    <table>
```

```
                <tr>
                    <th>id</th>
                    <th th:text="#{label.filename}"></th>
                    <th th:text="#{label.note}"></th>
                    <th></th>
                </tr>
                <tr th:each="af:*{attachedFileList}"><!-- ④ -->
                    <!-- id -->
                    <td th:text="${af.id}"></td>
                    <!-- ファイル名 -->
                    <!-- 別タブで表示するファイル -->
                    <td th:if="${af.openInNewTab}">
                        <!-- ⑤ -->
                        <a th:href="@{/todo/af/download/__${af.id}__}"
th:text="${af.fileName}"
                            target="_blank" rel="noopener noreferrer">
</a>
                    </td>
                    <!-- downloadするファイル -->
                    <td th:unless="${af.openInNewTab}">
                        <!-- ⑤ -->
                        <a th:href="@{/todo/af/download/__${af.id}__}"
th:text="${af.fileName}"></a>
                    </td>
                    <!-- メモ　 -->
                    <td th:text="${af.note}"></td>
                    <!-- 削除リンク　 -->
                    <td>
                        <!-- ⑥ -->
```

```
                        <a
th:href="@{/todo/af/delete(af_id=${af.id},todo_id=${todoData.id})}"
                            th:text="#{link.delete}"></a>
                    </td>
                </tr>
            </table>
        </div>
    </div><!-- ③更新の場合、添付ファイル一覧を表示する ここまで
↑ ↑ ↑ -->
    </div><!-- ① -->
    <!-- 更新の場合、Task一覧を表示する -->
    <div th:if="${session.mode == 'update'}">
        :
    </div>
    :
</body>
</html>
```

添付ファイル一覧はToDo入力欄と横並びにしたいので①でフレックスボックス("display: flex")を指定し、その中に②ToDo入力欄、③添付ファイル一覧を配置します。図で表すと以下のようになります。

【図18-13】フレックスボックスによる配置

一覧作成方法はToDo/タスクと同じです。th:eachでtodoDataのattachedFileListの要素数分trを作成します(④)。この中でid、添付ファイルのダウンロードリンク(⑤)、メモ、削除リンク(⑥)をtd要素で作成します。

⑤のリンクは【リスト18-16】ext2contentType()で求めたopenInNewTabの値によって、ファイルを別タブに表示するか、ローカルに保存するかを決めます。前者の場合、セキュリティ上の脆弱性(リスク)に対処するためrel="noopener noreferrer"も指定します(別タブから元の入力画面へのアクセスを防ぐ)。

```
    <!-- ファイル名 -->
    <!-- 別タブで表示するファイル -->
    <td th:if="${af.openInNewTab}">
      <!-- ⑤ -->
      <a th:href="@{/todo/af/download/__${af.id}__}"
 th:text="${af.fileName}"
        target="_blank" rel="noopener noreferrer"></a>
```

```
    </td>
    <!-- downloadするファイル -->
    <td th:unless="${af.openInNewTab}">
      <!-- ⑤ -->
      <a th:href="@{/todo/af/download/__${af.id}__}"
th:text="${af.fileName}"></a>
    </td>
```

18.3.2 コントローラーの追加

ここまで作成してきたコントローラーTodolistControllerは、Thymeleafが処理したテンプレート(HTML)をブラウザへ返すことを表す@Controllerを付与しています(前書「4.1 ThymeleafでHello, world!」参照)。しかしファイルをダウンロードしたときは、画面を出力しません(データに何の変更も加えていないため)。これは@Controllerが意図する動作とは異なります。

そこでファイルダウンロード用に新たなコントローラーDownloadControllerを追加します。こちらはコントローラーの処理結果をそのままブラウザへ送信するので**@RestController**を付与します(前書「3.1 Hello, world!」参照)。

【リスト18-18】
com.example.todolist.controller.DownloadController.java

```
package com.example.todolist.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import com.example.todolist.service.DownloadService;
import jakarta.servlet.http.HttpServletResponse;
```

```
import lombok.RequiredArgsConstructor;

@RestController
@RequiredArgsConstructor
public class DownloadController {
	private final DownloadService downloadService;

	// 添付ファイルのダウンロード処理
	@GetMapping("/todo/af/download/{afId}")
	public void downloadAttachedFile(@PathVariable(name =
"afId") int afId,
                                                HttpServletResponse
response) { // ①

		downloadService.downloadAttachedFile(afId, response); //
②
	}
}
```

ハンドラーメソッドdownloadAttachedFile()は、入力画面の添付ファイル一覧のリンクに対応しています。URLパス変数になっているのは添付ファイルのidです。

```
<a th:href="@{/todo/af/download/__${af.id}__}" ...></a>
```

ポイントは**HttpServletResponse**型の引数です(①)。これはブラウザへのレスポンスを管理するもので、ここに設定・出力した内容が処理結果となります。ダウンロードの場合は、この引数に対してファイルの中身を出

力するわけです。ただしこれも処理が少々複雑なので、サービスクラスにメソッドを作成します(②)。

18.3.3 ダウンロード処理

ファイルのダウンロード処理は大きく4ステップに分けることができます。いずれも前述のHttpServletResponseオブジェクトに対する処理となります。

①Content-Typeを設定する。
②Content-Dispositionを設定する。
③ファイルのサイズを設定する。
④ファイルの内容を出力する。

①Content-Typeを設定する

すでに説明したように、Content-Typeはデータ種別を表します。アップロードではブラウザが設定してくれましたが、ダウンロードのときはプログラムでセットします。そこで再びext2contentType(【リスト18-16】)を使います。

> Content-Type(MIMEタイプ)は次のメソッドでも取得できます。
> ・java.nio.file.Files#probeContentType(ファイル名から求める)
> ・java.net.URLConnection#guessContentTypeFromName(ファイルの中身から求める)

②Content-Dispositionを設定する

アップロードにも出てきましたが、ダウンロードでは以下のようにファイルをブラウザ上に表示するか、ローカルに保存するかを指定するのに使います。

Content-Disposition: inline

ファイルをブラウザ上に表示する。

Content-Disposition: attachment; filename="filename.exe"

filenameで指定した名前でファイルをローカルに保存する。

→Chromeのデフォルト動作ではダウンロードフォルダへ保存される。

③ファイルのサイズを設定する

Chromeでは省略可能ですが、設定するとダウンロードの状況(何％ダウンロードしたか？)が表示されます。

④ファイルの内容を出力する

HttpServletResponseから出力ストリームを取り出し、ファイルを書き込むとその内容がブラウザへ送信されます。これを受信したブラウザは②Content-Dispositionの設定内容から、表示するか、ローカル保存するか判断します。

以上をコード化したのがdownloadAttachedFile()です。

【リスト18-19】
com.example.todolist.service.DownloadService#downloadAttachedFile()

```
package com.example.todolist.service;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.URLEncoder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import com.example.todolist.common.Utils;
import com.example.todolist.entity.AttachedFile;
import com.example.todolist.repository.AttachedFileRepository;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class DownloadService {
    private final AttachedFileRepository attachedFileRepository;

    @Value("${attached.file.path}")
    private String ATTACHED_FILE_PATH;
```

```
        // 添付ファイルdownload処理
        public void downloadAttachedFile(int afId,
HttpServletResponse response) {
            // 添付ファイルの情報を取得する
            AttachedFile af =
attachedFileRepository.findById(afId).get();    // ※1

            // 拡張子からContent-Typeタイプを求める
            String fileName = af.getFileName();
            String fext = fileName.substring(fileName.lastIndexOf(".") +
1); // 拡張子(. の後ろ)
            String contentType = Utils.ext2contentType(fext);    // ※2

            // ダウンロードするファイル
            String downLoadFilePath =
Utils.makeAttahcedFilePath(ATTACHED_FILE_PATH, af);
            File downloadFile = new File(downLoadFilePath);    // ※3

            // ダウンロードファイル送信
            BufferedInputStream bis = null;
            OutputStream out = null;
            try {
                if (contentType.equals("")) {
                    // バイナリで送信
                    response.setContentType("application/force-
download");    // ①-1
                    // ローカル保存
                    response
```

```
                        .setHeader("Content-Disposition",
"attachment; filename=¥""
                                +
URLEncoder.encode(af.getFileName(), "UTF-8") + "¥""); // ②-1
                } else {
                    // 拡張子に対応するContent-Type
                    response.setContentType(contentType);    // ①-2
                    // 別タブ表示
                    response.setHeader("Content-Disposition",
"inline"); // ②-2
                }
                // ファイルのサイズ(byte)

response.setContentLengthLong(downloadFile.length());    // ③

                // ファイルの内容をブラウザへ出力
                bis = new BufferedInputStream(new
FileInputStream(downLoadFilePath));
                out = response.getOutputStream();
                byte[] buff = new byte[8 * 1024];
                int nRead = 0;
                while ((nRead = bis.read(buff)) != -1) {
                    out.write(buff, 0, nRead);    // ④
                }
                out.flush();

                // ファイルclose
                bis.close();
                out.close();
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

このメソッドでは上記①～④を実行するために、いくつかの準備処理をしています。

まずダウンロードする添付ファイルのidをキーにattached_fileテーブルを検索します(※１)。

```
    // 添付ファイルの情報を取得する
    AttachedFile af = attachedFileRepository.findById(afId).get();    // ※1
```

ファイルの拡張子からext2contentType()で対応するContent-Typeを求めます(※２)。

```
    String contentType = Utils.ext2contentType(fext);    // ※2
```

ダウンロードファイルを表すFileオブジェクトを作成します(※３)。ここから後でファイルサイズを取得します。

```
    String downLoadFilePath =
Utils.makeAttahcedFilePath(ATTACHED_FILE_PATH, af);
    File downloadFile = new File(downLoadFilePath);    // ※3
```

これで必要な情報が揃ったのでダウンロード処理を開始します。

まず※2でContent-Typeが得られた場合、それをセットし(①-1)、ファイルをブラウザへ表示するようにします(②-1)。

```
    // 拡張子に対応するContent-Type
    response.setContentType(contentType);    // ①-1
    // インライン表示
    response.setHeader("Content-Disposition", "inline");    // ②-1
```

setContentType()は、Content-Typeを設定するメソッドです。Content-DispositionはsetHeader()でレスポンスのヘッダー(Header)という部分に格納します(setContentType()のような専用メソッドはありません)。

一方、Content-Typeが得られなければ、ローカル保存するようにします。

```
    // バイナリで送信
    response.setContentType("application/octet-stream");    // ①-2
    // ファイル名
    response
            .setHeader("Content-Disposition",
                    "attachment; filename=¥""
                            + URLEncoder.encode(af.getFileName(), "UTF-8") +
"¥"");    // ②-2
```

①-2の"application/octet-stream"は、バイナリデータであることを表します。Chromeはこのタイプのデータを受信すると、それをローカルに保存します。

②-2のダウンロードファイル名はURLEncoder#encode()により、URLエンコードしています。これはファイル名に日本語(いわゆる全角文字)が含まれていたときに文字化けするのを防ぐためです。

ファイルサイズはsetContentLengthLong()で設定します(③)。

```
// ファイルのサイズ(byte)
response.setContentLengthLong(downloadFile.length());    // ③
```

そしてファイルの内容をgetOutputStream()で取得した出力ストリームに書き出します(④)。これでファイルがブラウザへ送信されます。

```
// ファイルの内容をブラウザへ出力
bis = new BufferedInputStream(new
FileInputStream(downLoadFilePath));
out = response.getOutputStream();
byte[] buff = new byte[8 * 1024];
int nRead = 0;
while ((nRead = bis.read(buff)) != -1) {
        out.write(buff, 0, nRead);    // ④
}
out.flush();
```

18.4 添付ファイルの削除

ファイルを添付できるようにしたら、それを削除する機能も必要でしょう。削除処理は2種類あります。

(1)画面の削除リンクをクリックした場合 → そのファイルのみ削除する。

(2)ToDoを削除した場合 → ToDoに関連する全ファイルを削除する。

削除対象は添付ファイルそのものと、その情報を格納したattached_fileテーブルのレコードです。

18.4.1 個別に削除する場合

入力画面の添付ファイル削除リンクは以下のようになっています(【リスト18-17】)。

```
    <!-- 削除リンク    -->
    <td>
      <a
th:href="@{/todo/af/delete(af_id=${af.id},todo_id=${todoData.id})}"
        th:text="#{link.delete}"></a>
    </td>
```

もしid=3のTodoが、id=5の添付ファイルを持っているとしたら、このリンクは以下のようになります。

```
<a
th:href="@{/todo/af/delete(af_id=${af.id},todo_id=${todoData.id})}"
...></a>
 ↓
<a th:href="/todo/af/delete?af_id=5&todo_id=3" ...></a>
```

以下はこのリンクに対応するハンドラーメソッドです。

【リスト18-20】

com.example.todolist.controller.TodoListController#deleteAttachedFile()

```
        // 添付ファイルを削除する
        @GetMapping("/todo/af/delete")
        public String deleteAttachedFile(@RequestParam(name =
"af_id") int afId,
                                         @RequestParam(name
= "todo_id") int todoId,
                                         RedirectAttributes
redirectAttributes,
                                         Locale locale) {
            // 添付ファイルを削除
            todoService.deleteAttachedFile(afId); // ①
            // attached_fileテーブルから削除
            attachedFileRepository.deleteById(afId); // ②

            // 削除完了メッセージをセットしてリダイレクト
            String msg =
messageSource.getMessage("msg.i.attachedfile_deleted", null,
locale);
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("I", msg));
            return "redirect:/todo/" + todoId;
        }
```

【リスト18-21】
src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.attachedfile_deleted=添付ファイルを削除しました。
```

【リスト18-22】
src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.attachedfile_deleted=The file has been deleted.
```

ここで添付ファイルを削除後(①)、attached_fileテーブルからもレコードを削除します(②)。実際のファイル削除処理はサービスクラスのメソッドで行います。

【リスト18-23】
com.example.todolist.service.TodoService#deleteAttachedFile()

```
    // 添付ファイル削除処理(AttachedFile.idで削除)
    public void deleteAttachedFile(int afId) {
        AttachedFile af = attachedFileRepository.findById(afId).get();
        File file = new
File(Utils.makeAttahcedFilePath(ATTACHED_FILE_PATH, af));
        file.delete();
    }
```

18.4.2 ToDoを削除した場合

処理方法は基本的に前節と同じですが、対象となる削除ファイルが複数件(0個以上)です。この処理をToDo削除に追加します。

【リスト18-24】
com.example.todolist.controller.TodoListController#deleteTodo()

```
        @PostMapping("/todo/delete")
        // @Transactional
        public String deleteTodo(@ModelAttribute TodoData todoData,
Model model,
                                 RedirectAttributes
redirectAttributes, Locale locale) {
            Integer todoId = todoData.getId();

            // 添付ファイルを削除
            todoService.deleteAttachedFiles(todoId); // ①

            // attached_fileテーブルから削除
            List<AttachedFile> attachedFiles
                =
attachedFileRepository.findByTodoIdOrderById(todoId);
            attachedFileRepository.deleteAllInBatch(attachedFiles); //
②

            // todoを削除
            todoRepository.deleteById(todoId);
            :
```

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

①の添付ファイル削除処理もサービスクラスで行います(こちらはdeleteAttachedFilesと複数形にしている)。削除するToDoのidをキーにTodoを検索し、該当する添付ファイル情報を取得します。そして拡張for文でそれらファイルをすべて削除します。

【リスト18-25】
com.example.todolist.service.TodoService#deleteAttachedFiles()

```
    // 添付ファイル削除処理(Todo.idで削除)
    public void deleteAttachedFiles(Integer todoId) {
        File file;
        List<AttachedFile> attachedFiles
            =
attachedFileRepository.findByTodoIdOrderById(todoId);
        for (AttachedFile af : attachedFiles) {
            file = new
File(Utils.makeAttahcedFilePath(ATTACHED_FILE_PATH, af));
            file.delete();
        }
    }
```

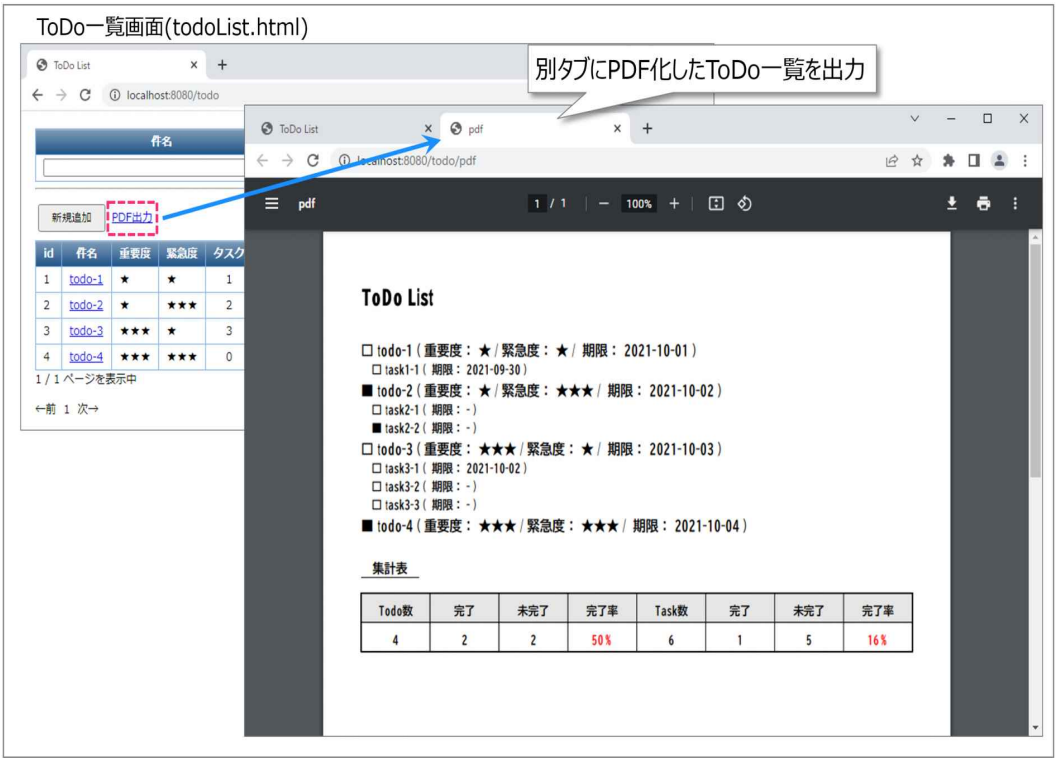
attached_fileのレコードは**deleteAllInBatch()**で一括削除します(②)。これも前書【表6-5】に含まれているリポジトリから自動生成されるメソッドです。引数にエンティティのリストを渡すと、それらをすべて削除します。

```
    // attached_fileテーブルから削除
    List<AttachedFile> attachedFiles
        = attachedFileRepository.findByTodoIdOrderById(todoId);
    attachedFileRepository.deleteAllInBatch(attachedFiles); // ②
```

19. PDF出力

実務ではシステムに登録されているデータを抽出して出力する機能を頻繁に作成します。代表的なものを3つあげるとすればCSV出力、PDF出力、Excel出力になるでしょう。このうちCSVは単なるテキストファイルなので、前章の応用で対処できます。しかしPDF/Excelは専用のファイルフォーマットを持っているため、PDF作成ライブラリ/Excel作成ライブラリを利用するのが一般的です。このときSpring Bootが提供する**ビュー(View)機能**を使うと、データ抽出とファイル作成をうまく分離できます。この方法を使い本章ではPDFファイル、次章ではExcelファイルを作成します。

以下は本章で実現するPDF出力の操作イメージです。ToDo一覧画面(todoList.html)のリンク[PDF出力]をクリックすると、全ToDo/タスクをPDFとして出力します。ブラウザではこれを別タブに表示します。



【図19-1】ToDoのPDF出力

> ChromeにはPDFファイル表示機能が内蔵されているので、追加ソフトのインストールは不要です。右上の[↓]ボタンでこの内容をローカルに保存できますし、プリンタマークで印刷もできま

> す。

19.1 PDF作成の仕組み

本節ではPDF出力用のビュー**AbstractPdfView**とPDFフォーマットで出力するために使用するライブラリ**OpenPDF**について説明します。

19.1.1 AbstractPdfView

ここまでToDoアプリは(前章のファイルダウンロードを除けば)処理結果をHTMLとしてブラウザに表示させてきました。この処理結果を「**View(ビュー)**」と言い、Spring BootではPDFやExcelもビューとして扱えます。

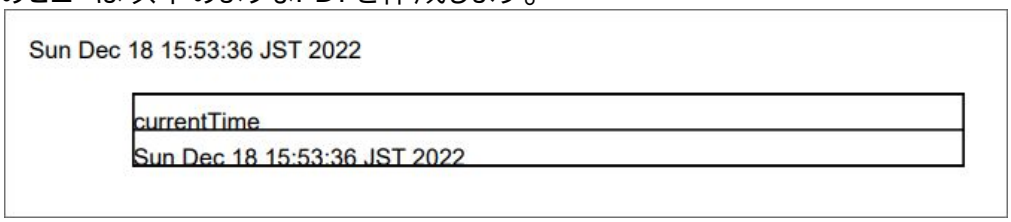
本章のようにPDFをビューとする場合には、抽象クラス**AbstractPdfView**を使います。このクラスには後述するように抽象メソッド**buildPdfDocument()**があり、これを実装したサブクラスを作成します。以下はその簡単な例です。呼び出し元からは"currentTime"という名前で、現在日時を表すjava.util.Date型のオブジェクトが渡されるとします。

```
package com.example.todolist.view;

import java.util.Map;
import org.springframework.web.servlet.view.document.AbstractPdfView;
import com.lowagie.text.Document;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class SamplePdf extends AbstractPdfView {
    @Override
    protected void buildPdfDocument(Map<String, Object> model,    // ①
                                    Document doc, // ④
                                    PdfWriter writer,
                                    HttpServletRequest request,
```

```
                                                HttpServletResponse response)
            throws Exception {
        // テキスト
        String currentTime =
((java.util.Date)model.get("currentTime")).toString();    // ①
        doc.add(new Paragraph(currentTime)); // ②④

        // 表
        Table table = new Table(1);
        table.addCell("currentTime"); // ③
        table.addCell(currentTime);
        doc.add(table); // ④
    }
}
```

※このSamplePdfはPDF生成プログラムの雰囲気を理解してもらうために用意したものです。
この段階では次節でインストールするOpenPDFが無いため文法エラーになります。
実際に試す場合は、プロジェクトにOpenPDFをインストールしてから行ってください。

このビューは以下のようなPDFを作成します。

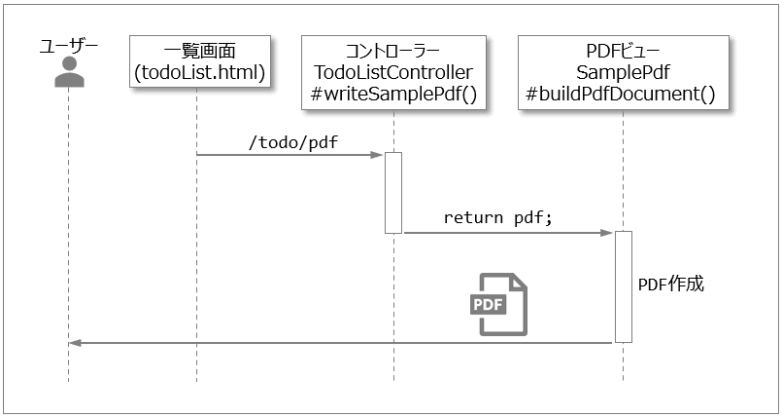


結果とプログラムを見比べると、大まかな構造がつかめると思います。
①データは引数のMap<String, Object>型オブジェクトから取得する。
②テキストはParagraphというクラスでラップしている。
③表はTableオブジェクトにaddCell()して作る。
④引数のDocument型オブジェクトにadd()すると、それがPDFになる。
大雑把ですが本章のPDFはこういった感じで作成していきます。

そしてこのビュー(SamplePdf)を呼び出すのはハンドラーメソッドです。

```
// PDF生成処理
@GetMapping("/todo/pdf")
public SamplePdf writeSamplePdf(SamplePdf pdf) {
        pdf.addStaticAttribute("currentTime", new java.util.Date());
        return pdf;
}
```

ここまでハンドラーメソッドの戻り値はModelAndViewやStringでしたが、これはSamplePdfになっています。さらに引数もSamplePdf型です。処理内容はPDFに表示するデータをaddStaticAttribute()で引数にセットし、それをreturnするだけです。これで自動的に上述のSamplePdf#buildPdfDocument()に制御が渡り、PDFを作成してブラウザへ返します。シーケンス図で表すと以下のようなイメージになります。



【図19-2】PDF出力のシーケンス

> 上図には記載していませんが、実際にはbuildPdfDocument()前後でSpring Bootが色々な処理をしています。興味がある方は調べてみてください。

一覧画面へ以下のリンク(①)を追加し、これをクリックすると、上記のPDFが別タブへ表示されます。

【リスト19-1】src/main/resources/templates/todoList.html(追加箇所)

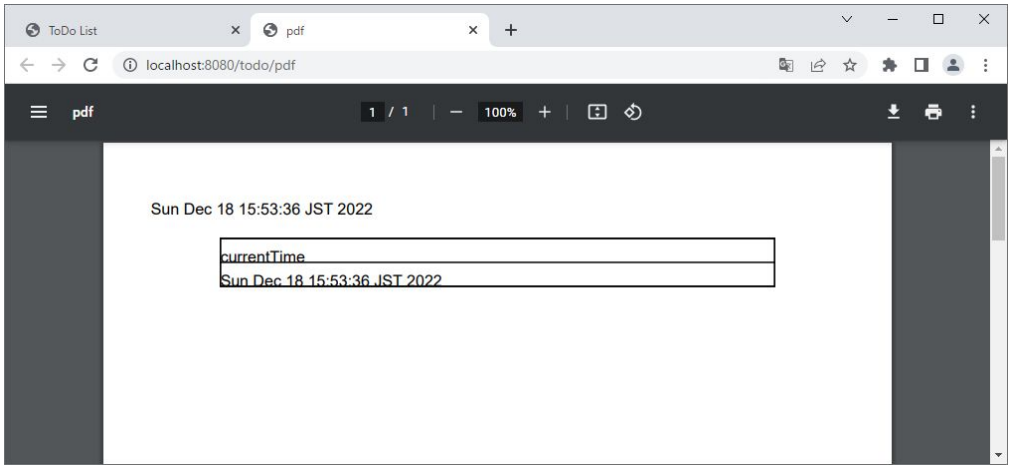
```
<!DOCTYPE html>
:
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 検索条件入力エリア -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
    :
    <!-- 新規追加ボタン -->
    <button type="submit" th:formaction="@{/todo/create/form}"
      th:text="#{button.new}"></button>
    <!-- ①PDF出力リンク を追加する-->
    <a th:href="@{/todo/pdf}" th:text="#{link.PDF}" target="_blank"
      rel="noopener noreferrer"></a>
  </form>
    :
</body>
</html>
```

【リスト19-2】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
link.PDF=PDF出力
```

【リスト19-3】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
link.PDF=Write PDF
```



> 前章ではダウンロードする際Content-Typeを設定しましたがPDFビューでは不要です。
> Spring Bootが自動的に"Content-Type:application/pdf"をセットしてくれます。

19.1.2 OpenPDF

前述のSamplePdfはDocument, Paragraph, PdfWriterといったクラスを使っていますが、これらのパッケージ名はcom.lowagie.textで始まっています。これはPDF作成ライブラリ「**OpenPDF**」のことです。AbstractPdfViewはPDFビューの枠組み(【図19-2】)だけを提供し、実際のPDF作成はこのライブラリ経由で行います。

> Java用のPDF作成ライブラリは何種類か存在しており、オープンソースではiTextが著名です。しかしSpringの公式リファレンスには、以下のような記述があります。
>
> Spring Web MVC > 1.11. View Technologies > 1.11.8　PDF and Excel
>
> https://docs.spring.io/spring-framework/docs/6.0.0/reference/html/web.html#mvc-view-document
>
> **Note**
>
> You should use the latest versions of the underlying document-generation libraries, if possible. In particular, we strongly recommend OpenPDF (for example, OpenPDF 1.2.12) instead of the outdated original iText 2.1.7, since OpenPDF is actively maintained and fixes an important vulnerability for untrusted PDF content.
>
> DeepLによる翻訳
>
> 「可能であれば、基礎となる文書生成ライブラリの最新バージョンを使用する必要があります。特に、OpenPDFは活発にメンテナンスされており、信頼できないPDFコンテンツに対する重要な脆弱性が修正されているので、古くなったオリジナルのiText 2.1.7ではなく、OpenPDF（例えば、OpenPDF 1.2.12）を強くお勧めします。」
>
> そこで本書もOpenPDFを使用しています(これもオープンソースです)。OpenPDFはコーディングに利用できる情報源が少々不足している感もありますが、幸いOpenPDFはiTextからフォーク(分岐)したものなのでiTextの情報が参考になります。OpenPDFで行き詰まったら、iTextではどうしているか調べてみると良いでしょう。

作成するプロジェクトの仕様

プロジェクト名	Todolist12
作成ファイル	com.example.todolist.view.TodoPdf.java

```
Todolist12
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist12Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    ├ DownloadController.java
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ AttachedFile.java
 │    │    ├ Task.java
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ AttachedFileData.java
 │    │    ├ TaskData.java
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    ├ AttachedFileRepository.java
 │    │    ├ TaskRepository.java
 │    │    └ TodoRepository.java(▲)
 │    ├ com.example.todolist.service
```

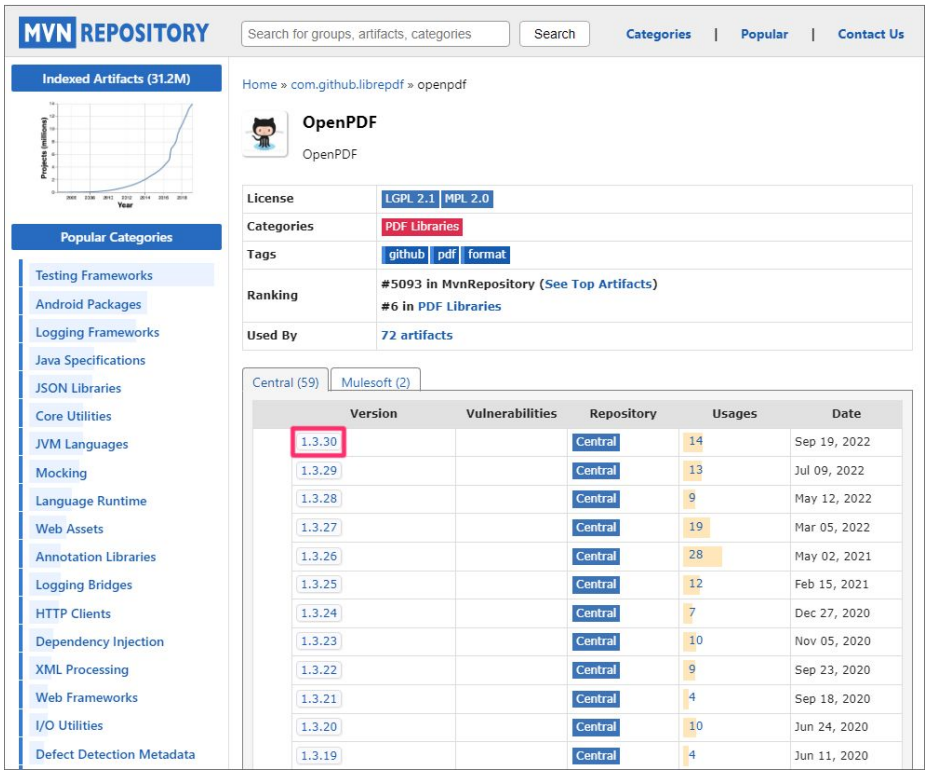
```
│   │   ├ DownloadService.java
│   │   └ TodoService.java
│   └ com.example.todolist.view(★)
│       └ TodoPdf.java(★)
└ src/main/resources
    ├ i18n
    │   ├ FixedDisplayStrings.properties
    │   ├ FixedDisplayStrings_en.properties(▲)
    │   ├ FixedDisplayStrings_ja.properties(▲)
    │   ├ OperationMessages_en.properties
    │   ├ OperationMessages_ja.properties
    │   ├ ValidationMessages_en.properties
    │   └ ValidationMessages_ja.properties
    :
    ├ templates
    │   ├ fragments.html
    │   ├ todoForm.html
    │   └ todoList.html(▲)
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

プロジェクトを作成したらOpenPDFを使えるようにします。現在の最新版は以下のページで確認できます。

https://mvnrepository.com/artifact/com.github.librepdf/openpdf



→最新版は1.3.30

本書では執筆時点の最新版1.3.30を使います。手順は以下の通りです。
(1)パッケージ・エクスプローラーでプロジェクト直下のpom.xmlをダブルクリックして開く。
(2)</dependencies>の直前に以下を追加する。

```
<!-- https://mvnrepository.com/artifact/com.github.librepdf/openpdf -->
<dependency>
    <groupId>com.github.librepdf</groupId>
    <artifactId>openpdf</artifactId>
    <version>1.3.30</version>
</dependency>
```

→ <!--　～ -->はコメントなので入力しなくて可

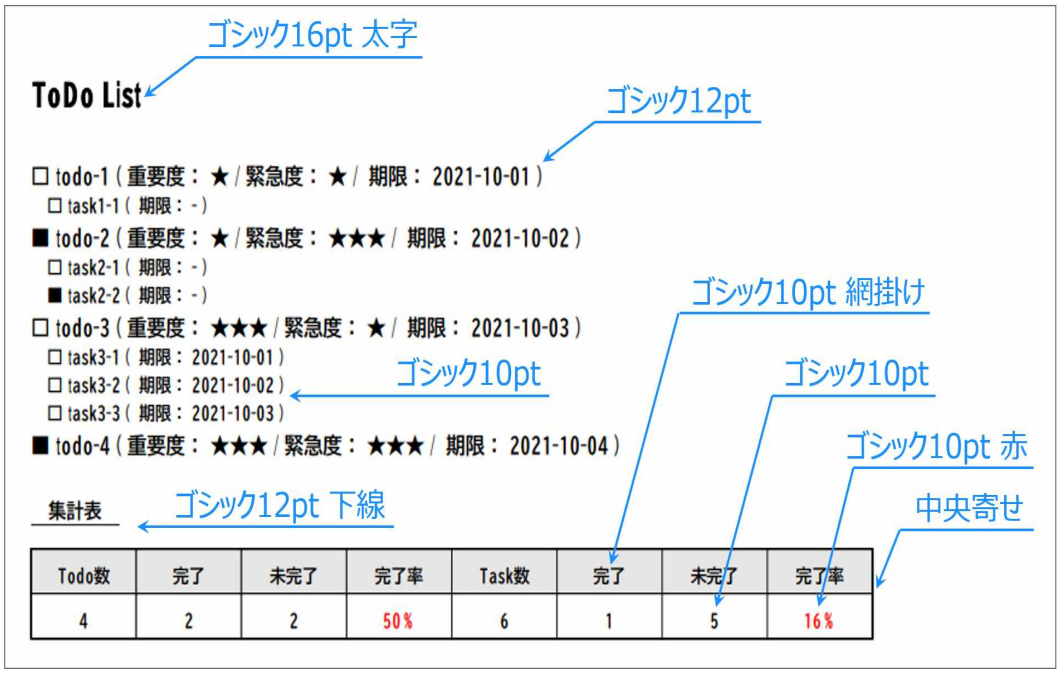
(3)[CTRL] + S を押下する　→　mavenがOpenPDFをダウンロードしてプロジェクトへ組み込む。

(4)pom.xmlを閉じる。

19.2 PDF文書の作成

19.2.1 作成仕様

本章では下図のようなPDFを出力できるようにします。



【図19-3】PDF出力仕様

ToDo List

・全てのToDoをid順(昇順)に出力する。

・関連するタスクはToDoに続けて出力する。

・完了しているToDo、タスクの行頭には■を出力する。未完なら□とする。

集計表

・ToDo、タスクそれぞれについて件数、完了数、未完了数、完了率を出力する。

その他

フォントなどの設定は上図のようにする。

19.2.2 出力データの取得

まず「全ToDoのid順(昇順)」に対応するメソッドfindAllByOrderById()をTodoのリポジトリに宣言します。

【リスト19-4】com.example.todolist.repository.TodoRepository.java(追加個所)

```
package com.example.todolist.repository;
import java.util.List;
   :
@Repository
public interface TodoRepository extends JpaRepository<Todo, Integer> {
      List<Todo> findAllByOrderById();    // ←追加
}
```

このように条件で絞り込まない場合、OrderByの前にもByが必要です。findAllOrderById()ではエラーになるので注意してください。

出力データはハンドラーメソッドで取得し、PDFビューへ渡します。TodoPdfは次節で定義するPDFビューです。これが【図19-3】で示した様式にデータを編集します。

【リスト19-5】
com.example.todolist.controller.TodoListController#writeTodoPdf()

```
      :
import com.example.todolist.view.TodoPdf;
      :
   // PDF生成処理
   @GetMapping("/todo/pdf")
   public TodoPdf writeTodoPdf(TodoPdf pdf) {
         List<Todo> todoList = todoRepository.findAllByOrderById();
         pdf.addStaticAttribute("todoList", todoList);
         return pdf;
   }
```

なお一覧画面には、このハンドラーメソッドを呼び出すリンクを追加してください(【リスト19-1】①)。

19.2.3 PDFビューの作成

ここで定義するPDFビューcom.example.todolist.view.TodoPdf.javaは200行以上あるため掲載しません。サポートサイトからダウンロードしたソースプログラムを参照してください。

【表19-1】TodoPdfで使用している主なOpenPDFのクラス(パッケージcom.lowagie.text)

クラス名	説明
Document	出力するPDF文書を表すクラス
Font	フォントを指定するためのクラス(色、太字、下線の指定も可能)
Phrase	文字列を設定するためのクラス
Paragraph	文字列を設定するためのクラス(インデントや表示位置の設定が可能)
Table	表を表すクラス
Cell	表の各セルを表すクラス

フォントの定義

TodoPdfでは、最初にFont(フォント)オブジェクトを生成しています。後述するようにフォントは文字列をセットするとき使いますが、その都度生成するよりパターン別に用意しておいた方が効率的です。

Fontクラスのコンストラクタは複数ありますが、ここで使っているのは第1引数にフォントの種類、第2引数にフォントの大きさ、第3引数にフォントのスタイルを渡すものです。

```
    // ゴシック16pt_太字
    Font font_g16_bold = new Font(
        BaseFont.createFont("HeiseiKakuGo-W5", "UniJIS-UCS2-H",
BaseFont.NOT_EMBEDDED),
        16,
        Font.BOLD);
```

第1引数：フォントの種類

BaseFont#createFont()で作成します。

第1引数：フォント名

日本語を出力する場合は、以下のどちらかを指定します。

・"HeiseiMin-W3"：明朝体

・"HeiseiKakuGo-W5"：ゴシック体

※著者の環境ではChrome/Edge(ともにVersion 108)の場合、どちらを指定してもゴシック体になります。

Firefox(Version 108(64ビット))では明朝体/ゴシック体を切り替えられました。

第2引数：エンコーディング名

日本語で横書きの場合、以下を指定します。

・"UniJIS-UCS2-H"：横書き

第3引数：PDFにフォントを埋め込むかどうか

第2引数：フォントの大きさ

第3引数：フォントのスタイル

以下のようなものがあります。

・Font.BOLD：太文字

・Font.BOLDITALIC：太文字＋斜体

・Font.ITALI：斜体

・Font.NORMAL：標準

・Font.UNDERLINE：下線

```
// ゴシック10pt_赤
Font font_g10_red = new Font(
    BaseFont.createFont("HeiseiKakuGo-W5", "UniJIS-UCS2-H",
BaseFont.NOT_EMBEDDED),
    10,
    Font.NORMAL
    new Color(255, 0, 0));
```

フォント色を指定する場合は、上記のように４番目の引数としてjava.awt.Colorを渡すコンストラクタを使います。Colorの引数はR, G, Bです。

文字列の出力

OpenPDFには文字列を表すクラスとしてChunk(かたまり), Phrases(語句/文節), Paragraph(段落)の３つがあります。TodoPdfではToDo一覧部分にParagraph、集計表内はPhrasesを使います。

Paragraphを使った見出し出力は以下のようになります。

```
// 見出し
doc.add(new Paragraph("ToDo List    ", font_g16_bold));
```

文字列とフォントからParagraphを生成し、それをビューの引数doc(Documentオブジェクト)へ追加します。

ToDo、タスク出力も同じです。それぞれヘルパーメソッドtodo2String()/task2String()で編集した結果をdocへ追加します。

```
// ToDoを編集して追加
 doc.add(new Paragraph(todo2String(todo), font_g12));

// タスクを編集して追加
doc.add(new Paragraph("¥t" + task2String(task), font_g10));
```

表の作成

表はTableクラスで作成します。

```
// 表の定義(8列)
Table summaryTable = new Table(8);
// セル幅
summaryTable.setWidth(100);
// 罫線との余白
summaryTable.setPadding(3);
```

コンストラクタの引数は列数を表しています(行数も指定可能)。

setWidth()は、用紙サイズ(デフォルトA4)を基準とした表の幅を設定します。100なら横幅一杯になります。

setPadding()は、罫線と文字列の間隔です。単位はポイント(1ポイント=25.4mm/72dpi=約0.35mm)です。

表の各セルはCellオブジェクトをaddCell()で追加して作成します。以下は表の見出し行部分です。

```
    // 見出し行
    // Todo数, 完了, 未完了, 完了率, Task数, 完了, 未完了, 完了率
    HorizontalAlignment hcenter = HorizontalAlignment.CENTER;
    String headers[]
            = {"Todo数", "完了", "未完了", "完了率", "Task数", "完了", "未完了", "完了率"};
    for (String header : headers) {
            summaryTable.addCell(makeCell(header, font_g10, 0.9f, hcenter));
    }
```

この中のmakeCell()はヘルパーメソッドで、引数を基にCellオブジェクトを生成して返します。

```
    // Cell作成ヘルパーメソッド
    private Cell makeCell(String content,    // 出力する文字列
                          Font font,    // フォント
                          float grayFill,    // 背景色の濃さ
                          HorizontalAlignment alignment) {    // 文字の水平方向の配置
        // Cellオブジェクト生成
        Cell cell = new Cell(new Phrase(content, font));
        // 網掛け
        cell.setGrayFill(grayFill);
        // 中央寄せ
        cell.setHorizontalAlignment(alignment);

        return cell;
    }
```

最初に文字列とフォントからPhraseオブジェクトを生成し、これを基にCellオブジェクトを作成します。
setGrayFill()は、背景色の濃さを設定します(0.0で黒、1.0で白になる)。
setHorizontalAlignment()は、文字の水平方向の配置を決めます。HorizontalAlignment.CENTERなら「中央寄せ」となります。

上述の拡張for文はaddCell()を8回実行します。これはテーブルの列数(=8)と同じであり、1 行分のセルをセットしたことになります。この状態でさらにaddCell()を呼び出すと、そのセルは自動的に次の行へ配置されます。

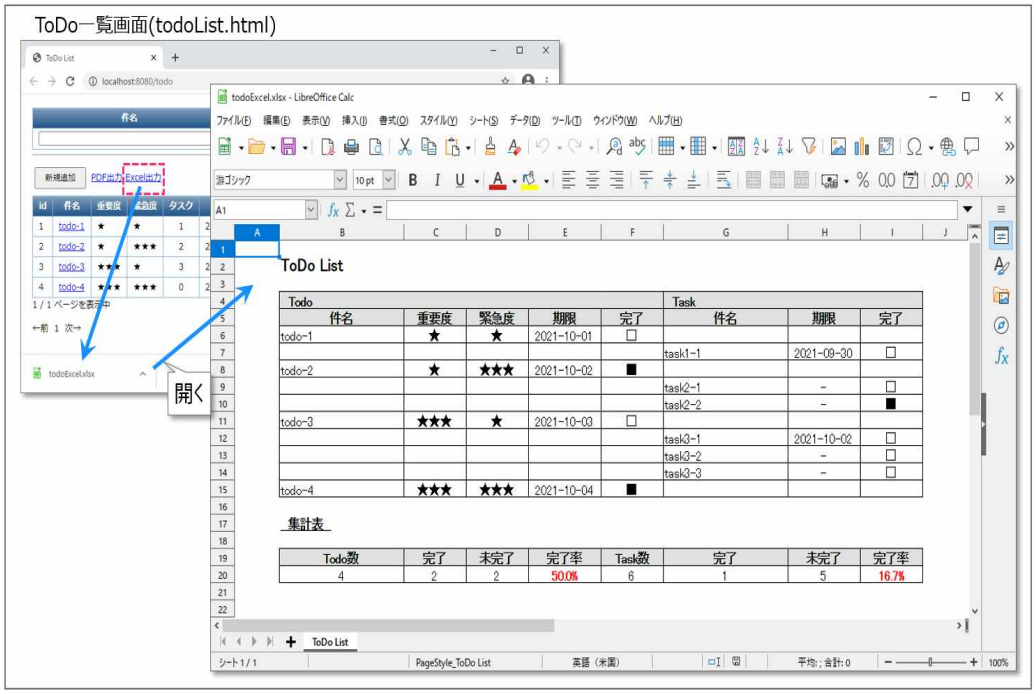
2 行目の集計数値も同じように作成し、最後にdocへ追加します。

```
    doc.add(summaryTable);
```

20. Excel出力

本章ではToDoをExcelファイルとしてダウンロードできるようにします。

以下は操作イメージです。ToDo一覧画面(todoList.html)のリンク[Excel出力]をクリックすると、全ToDo/タスクをExcelファイルとして自動的にダウンロードします。このファイルはフリーかつオープンソースの表計算ソフト**LibreOffice Calc**でも開くことができます。



【図20-1】ToDoのExcel出力(LibreOffice Calcで開いた状態)

> LibreOfficeのインストール方法を章末に載せてあります。興味がある方は参考にしてください。

20.1 Excel作成の仕組み

本節ではExcel出力用のビュー**AbstractXlsxView**とExcelフォーマットで出力するために使用するライブラリ**Apache POI**について説明します。

20.1.1 AbstractXlsxView

Excelファイルの作成方法は前章のPDFと同じ流れです。抽象クラス**AbstractXlsxView**のサブクラスを作成し、抽象メソッド**buildExcelDocument()**を実装します。このオブジェクトをハンドラーメソッドからreturnすればExcelファイルがブラウザへダウンロードされます。

以下は簡単なサンプルクラスです。呼び出し元からは"currentTime"という名前で現在日時を表すjava.util.Date型のオブジェクトが渡されるとします。

```
package com.example.todolist.view;

import java.net.URLEncoder;
import java.util.Map;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.apache.poi.ss.usermodel.BorderStyle;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.CellStyle;
import org.apache.poi.ss.usermodel.FillPatternType;
import org.apache.poi.ss.usermodel.Font;
import org.apache.poi.ss.usermodel.HorizontalAlignment;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.springframework.web.servlet.view.document.AbstractXlsxView;

public class SampleExcel extends AbstractXlsxView {
    @Override
```

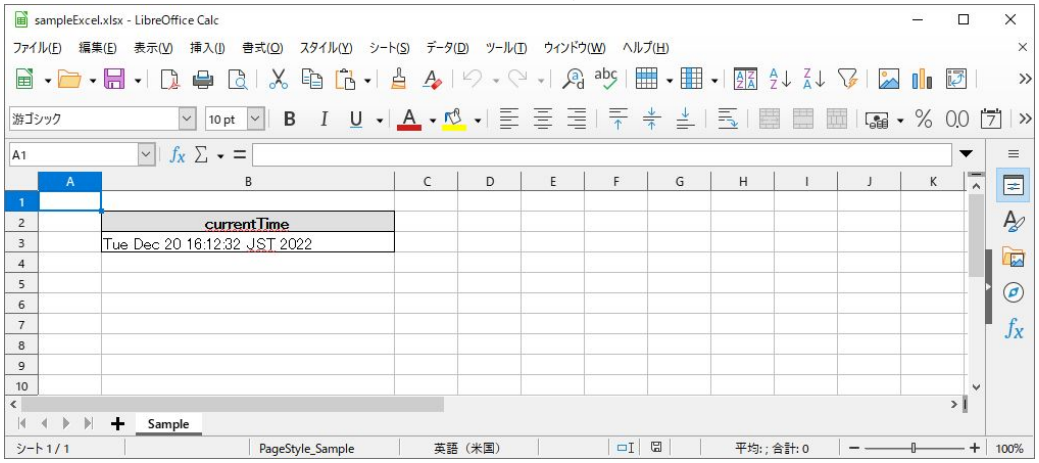
```
    protected void buildExcelDocument(Map<String, Object> model,
                                      Workbook workbook,
                                      HttpServletRequest request,
                                      HttpServletResponse response)
            throws Exception {

        // ①データ取得
        String currentTime = ((java.util.Date)model.get("currentTime")).toString();

        // ②ダウンロードファイル名をセット
        String fileName = (String)model.get("fileName");
        response
            .setHeader("Content-Disposition",
                "attachment; filename=¥"" + URLEncoder.encode(fileName, "UTF-8")
+ "¥"");

        // ③網掛け + 罫線 上右下左 + フォント12 + 文字中央寄せ + 太字
        CellStyle pattern_borderAll_fontSize12_center_bold = workbook.createCellStyle();

pattern_borderAll_fontSize12_center_bold.setFillPattern(FillPatternType.LESS_DOTS
);
        pattern_borderAll_fontSize12_center_bold.setBorderTop(BorderStyle.HAIR);
        pattern_borderAll_fontSize12_center_bold.setBorderRight(BorderStyle.HAIR);

pattern_borderAll_fontSize12_center_bold.setBorderBottom(BorderStyle.HAIR);
        pattern_borderAll_fontSize12_center_bold.setBorderLeft(BorderStyle.HAIR);

pattern_borderAll_fontSize12_center_bold.setAlignment(HorizontalAlignment.CEN
TER);

        Font font12_bold = workbook.createFont();
        font12_bold.setFontName("MS Pゴシック");
        font12_bold.setFontHeightInPoints((short)12);
        font12_bold.setBold(true);
```

```
pattern_borderAll_fontSize12_center_bold.setFont(font12_bold);

// ③罫線 上右下左 + フォント12
CellStyle borderAll_fontSize12 = workbook.createCellStyle();
borderAll_fontSize12.setBorderTop(BorderStyle.HAIR);
borderAll_fontSize12.setBorderRight(BorderStyle.HAIR);
borderAll_fontSize12.setBorderBottom(BorderStyle.HAIR);
borderAll_fontSize12.setBorderLeft(BorderStyle.HAIR);

Font font12 = workbook.createFont();
font12.setFontName("MS Pゴシック");
font12.setFontHeightInPoints((short)12);
borderAll_fontSize12.setFont(font12);

// ④シート作成
Sheet sheet = workbook.createSheet("Sample");
// 列幅設定
sheet.setColumnWidth(1, 256 * 40);

// ⑤データセット
// 2行目を作成(行番号は0から始まる)
Row row = sheet.createRow(1);
// 2行目にB列を作成(列番号は0→A列、1→B列、...)
Cell cell = row.createCell(1);
// B2に文字列と書式をセット
cell.setCellValue("currentTime");
cell.setCellStyle(pattern_borderAll_fontSize12_center_bold);

// C2に時刻と書式をセット
row = sheet.createRow(2);
cell = row.createCell(1);
cell.setCellValue(currentTime);
cell.setCellStyle(borderAll_fontSize12);
```

```
    }
}
```

※このSampleExcelはExcel生成プログラムの雰囲気を理解してもらうために用意したものです。この段階では次節でインストールするApache POIが無いため文法エラーになります。実際に試す場合は、プロジェクトにApache POIをインストールしてから行ってください。

このビューは以下のようなExcelファイルを作成します。



結果とプログラムを見比べると、大まかな構造がつかめると思います。
①引数のMap<String, Object>型オブジェクトからデータを取得する。
②ダウンロードファイル名を設定する。
③セルのスタイル(装飾)を定義する。
④引数のWorkbook型オブジェクトからシート(Sheet)を作成する。
⑤作成したシートに行作成→セル作成し、テキストとスタイルをセットする。
これがApache POIでExcelファイルを作成する基本的な流れと思って良いでしょう。
あとは必要に応じて計算式などを設定します。

このビューを呼び出すハンドラーメソッドは以下のようになっています。

```
    // Excel生成処理
    @GetMapping("/todo/excel")
    public SampleExcel writeTodoExcel(SampleExcel excel) {
        excel.addStaticAttribute("currentTime", new java.util.Date());
        excel.addStaticAttribute("fileName", "sampleExcel.xlsx");
```

```
        return excel;
    }
```

考え方は前章のPDFと同じです。ハンドラーメソッドの戻り値、および引数がSampleExcel型(ビュー)です。処理はExcelファイルに出力したいデータをaddStaticAttribute()でセットし、それをreturnするだけです。これで自動的に上述したSampleExcel#buildExcelDocument()に制御が渡り、Excelファイルを作成して、ブラウザへ返します。

一覧画面へ以下のリンク(①)を追加し、これをクリックすると、上記のようなExcelファイルがダウンロードされます。

【リスト20-1】src/main/resources/templates/todoList.html(追加箇所)

```
<!DOCTYPE html>
:
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- 検索条件入力エリア -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
  :
    <!-- 新規追加ボタン -->
    <button type="submit" th:formaction="@{/todo/create/form}"
      th:text="#{button.new}"></button>
    <!-- PDF出力リンク-->
    <a th:href="@{/todo/pdf}" th:text="#{link.PDF}" target="_blank"
      rel="noopener noreferrer"></a>
    <!-- ①Excel出力リンク を追加する -->
    <a th:href="@{/todo/excel}" th:text="#{link.Excel}"></a>
  </form>
    :
</body>
</html>
```

【リスト20-2】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
link.Excel=Excel出力
```

【リスト20-3】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
link.Excel=Write Excel
```

> SampleExcelもContent-Typeを設定していませんが、自動的に以下のMIME Typeがセットされています。
> application/vnd.openxmlformats-officedocument.spreadsheetml.sheet

20.1.2 Apache POI

前述のSampleExcelではorg.apache.poiで始まるパッケージをインポートしています。これはApacheソフトウェア財団が開発している、**Apache POI**(ポイ/ピーオーアイ)というExcelやWordといったMicrosoft Office形式のファイルを操作するためのJava用ライブラリです。

■POI API Documentation
https://poi.apache.org/apidocs/5.0/index.html

上記サイトを見ると数多くのパッケージがありますが、このうちExcel操作の基本となるのは以下の３つです。

・org.apache.poi.hssf：.xls形式のファイルを扱う
・org.apache.poi.xssf：.xlsx形式のファイルを扱う
・org.apache.poi.ss：.xlsおよび.xlsx共通のスーパーインターフェース

「.xls形式ならhssf」「.xlsx形式ならxssf」という位置づけですが、ToDoアプリでは特に区別する必要が無いのでorg.apache.poi.ssを使います。

作成するプロジェクトの仕様

プロジェクト名	Todolist13

作成ファイル	com.example.todolist.view.TodoExcel.java

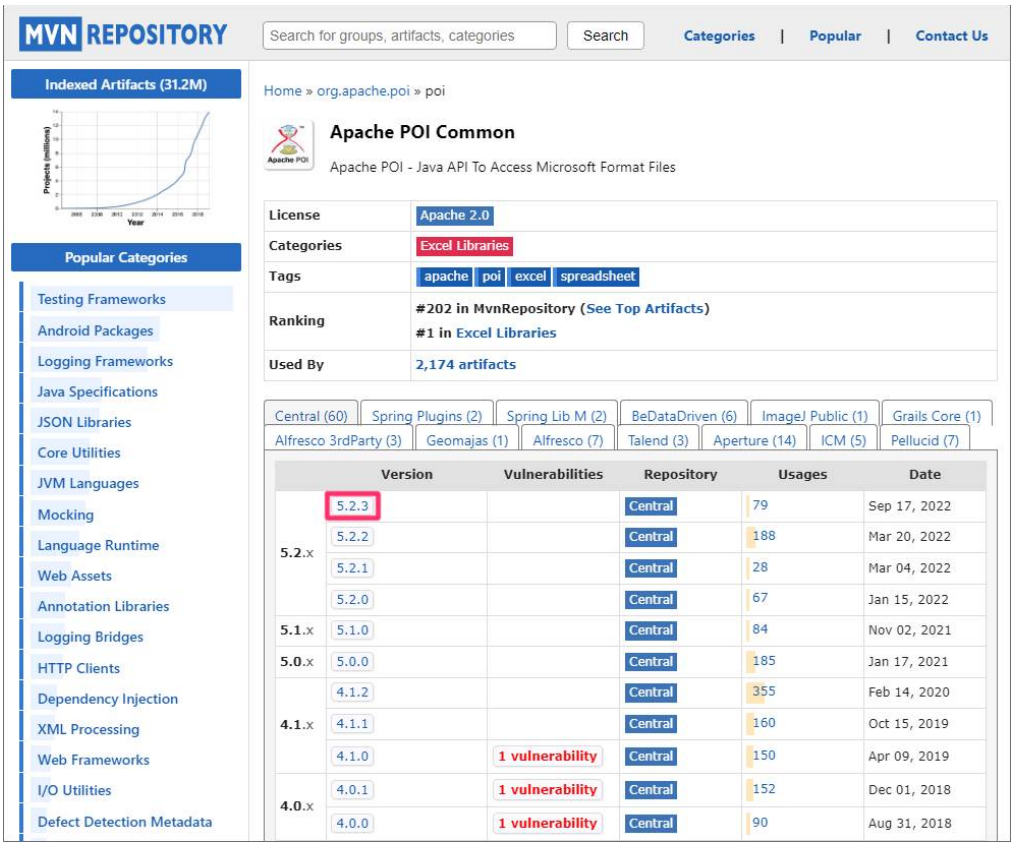
```
Todolist13
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist13Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.controller
 │    │    ├ DownloadController.java
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ AttachedFile.java
 │    │    ├ Task.java
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ AttachedFileData.java
 │    │    ├ TaskData.java
 │    │    ├ TodoData.java
 │    │    └ TodoQuery.java
 │    ├ com.example.todolist.repository
 │    │    ├ AttachedFileRepository.java
 │    │    ├ TaskRepository.java
 │    │    └ TodoRepository.java
 │    ├ com.example.todolist.service
 │    │    ├ DownloadService.java
 │    │    └ TodoService.java
 │    └ com.example.todolist.view
 │         ├ TodoExcel.java(★)
 │         └ TodoPdf.java
```

```
└ src/main/resources
    ├ i18n
    │    ├ FixedDisplayStrings.properties
    │    ├ FixedDisplayStrings_en.properties(▲)
    │    ├ FixedDisplayStrings_ja.properties(▲)
    │    ├ OperationMessages_en.properties
    │    ├ OperationMessages_ja.properties
    │    ├ ValidationMessages_en.properties
    │    └ ValidationMessages_ja.properties
    :
    ├ templates
    │    ├ fragments.html
    │    ├ todoForm.html
    │    └ todoList.html(▲)
    └application.properties
```

★：このプロジェクトで追加する
▲：前プロジェクトの内容を一部変更する

プロジェクトを作成したらApache POIを使えるようにします。使用可能なVersionは以下のページで確認できます。

https://mvnrepository.com/artifact/org.apache.poi/poi



→最新版は5.2.3

本書では執筆時点の最新版5.2.3を使います。手順は以下の通りです。
(1)パッケージ・エクスプローラーでプロジェクト直下のpom.xmlをダブルクリックして開く。
(2)</dependencies>の直前に以下を追加する。

```
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
		<dependency>
		<groupId>org.apache.poi</groupId>
		<artifactId>poi</artifactId>
		<version>5.2.3</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->
<dependency>
		<groupId>org.apache.poi</groupId>
		<artifactId>poi-ooxml</artifactId>
		<version>5.2.3</version>
</dependency>
```

```
            <version>1.3.30</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
        <dependency>
            <groupId>org.apache.poi</groupId>
            <artifactId>poi</artifactId>
            <version>5.2.3</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->
        <dependency>
            <groupId>org.apache.poi</groupId>
            <artifactId>poi-ooxml</artifactId>
            <version>5.2.3</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
```

→ <!--　～ -->はコメントなので入力しなくて可

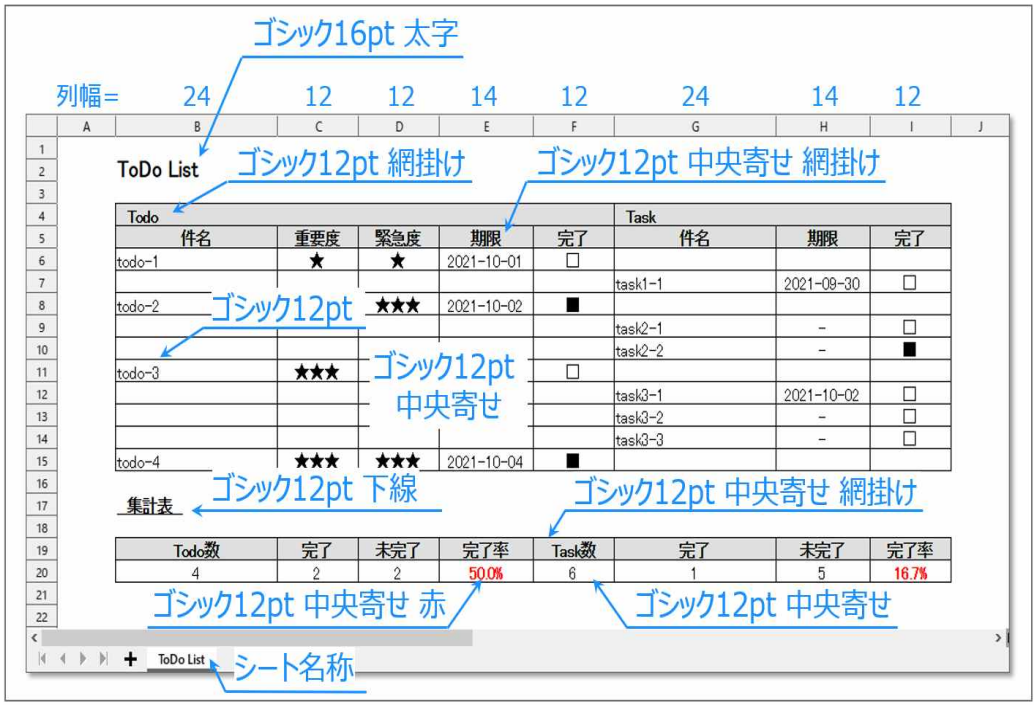
(3)[CTRL] + S を押下する　→　mavenがApache POIをダウンロードしてプロジェクトへ組み込む。

(4)pom.xmlを閉じる。

20.2 Excelファイルの作成

20.2.1 作成仕様

本章では下図のような一覧表と集計表からなるExcelファイルをダウンロードできるようにします。



【図20-2】Excel出力仕様

ToDo List

・全てのToDoをid順(昇順)に出力する。

・関連するタスクはToDoに続けて処理する。

・完了しているToDo、タスクの完了欄には■を出力する。未完なら□とする。

集計表

・ToDo/タスクそれぞれの件数、完了数、未完了数、完了率を出力する。

・各数値はセルに計算式を設定して求める。

その他

フォント、罫線の設定は上図のようにする。

20.2.2 出力データの取得

出力するデータは前章PDFと同じ「全ToDoのid順」です。メソッドは追加済みなので、それをハンドラーメソッドから呼び出し、結果をビューへセットします。またダウンロードする際のファイル名も渡します。

【リスト20-4】
com.example.todolist.controller.TodoListController#writeTodoPdf()

```
    // Excel生成処理
    @GetMapping("/todo/excel")
    public TodoExcel writeTodoExcel(TodoExcel excel) {
        List<Todo> todoList = todoRepository.findAllByOrderById();
        excel.addStaticAttribute("todoList", todoList);
        excel.addStaticAttribute("fileName", "todoExcel.xlsx");

        return excel;
    }
```

20.2.3 Excelビューの作成

ここで定義するExcelビューcom.example.todolist.view.TodoExcel.javaは400行近くあるため掲載しません。サポートサイトからダウンロードしたソースプログラムを参照してください。

【表20-1】TodoExcelで使用している主なApache POIのクラス(パッケージ org.apache.poi.ss.usermodel)

クラス名	説明
Workbook	出力するExcelファイルを表すクラス
Sheet	シートを表すクラス
Row	行を表すクラス
Cell	セルを表すクラス
CellStyle	セルの書式

FillPatternType	セルの塗りつぶし(網掛け)
BorderStyle	セルの罫線(枠線)
Font	セルのフォント
HorizontalAlignment	文字の水平方向位置
IndexedColors	セルの文字色

ダウンロードファイル名の設定

18章のダウンロードファイルと同じようにContent-Dispositionで設定します。HttpServletResponseオブジェクトはbuildExcelDocument()の引数となっているのでそれを使います。

```
//  ダウンロードファイル名をセット
   String fileName = (String)model.get("fileName");
   response
      .setHeader("Content-Disposition",
            "attachment; filename=¥"" + URLEncoder.encode(fileName, "UTF-8") +
"¥"");
```

CellStyleの定義(1)

このTodoExcelは、全体のおよそ1/3をセルの書式を表すCellStyleオブジェクトの生成に費やしています。

例えば以下のfontSize16_boldは見出し「ToDo List」用のものです。【図20-2】で示したように「ゴシックフォント/16ポイント/太字」としています。

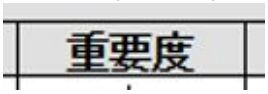
```
   // フォンサイズ 16 + 太文字
   CellStyle fontSize16_bold = workbook.createCellStyle(); // ①

   Font font16_bold = workbook.createFont(); // ②
   font16_bold.setFontName("MS Pゴシック"); // ③
   font16_bold.setFontHeightInPoints((short)16); // ④
   font16_bold.setBold(true); // ⑤

   fontSize16_bold.setFont(font16_bold); // ⑥
```

CellStyleオブジェクトはnew()ではなく、Workbook#createCellStyle()で作成します(①)。同様にFontオブジェクトもWorkbook#createFont()で作成します(②)。そして

Fontオブジェクトにフォント名(③)、サイズ(④)、太字かどうか(⑤)を設定し、最後にCellStyleオブジェクトにセットします(⑥)。

CellStyleの定義(2)

次のpattern_borderAll_fontSize12_center_boldは、セルの上下左右に罫線があり、塗りつぶし(網掛け)を施しています。また文字はセル内で中央寄せとしています。



CellStyleの作成(①)とFontの作成・設定(⑤⑥)は、前述のfontSize16_boldと同じ考え方です。

```
    // 網掛け + 罫線 上下左右 + フォント12 + 文字中央寄せ + 太字
    CellStyle pattern_borderAll_fontSize12_center_bold = workbook.createCellStyle();// ①

pattern_borderAll_fontSize12_center_bold.setFillPattern(FillPatternType.LESS_DOTS);//
②
    pattern_borderAll_fontSize12_center_bold.setBorderTop(BorderStyle.HAIR);// ③
    pattern_borderAll_fontSize12_center_bold.setBorderRight(BorderStyle.HAIR);
    pattern_borderAll_fontSize12_center_bold.setBorderBottom(BorderStyle.HAIR);
    pattern_borderAll_fontSize12_center_bold.setBorderLeft(BorderStyle.HAIR);

pattern_borderAll_fontSize12_center_bold.setAlignment(HorizontalAlignment.CENT
ER);// ④

    Font font12_bold = workbook.createFont();// ⑤
    :
    pattern_borderAll_fontSize12_center_bold.setFont(font12_bold);// ⑥
```

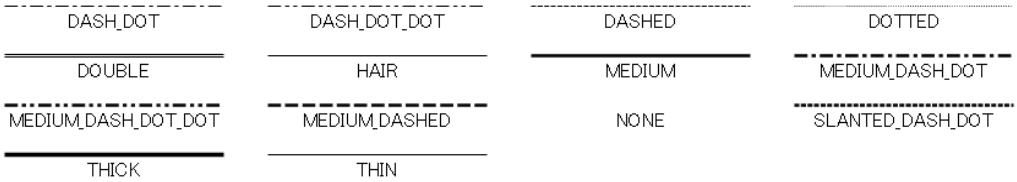
塗りつぶしはCellStyle#setFillPattern()で設定します(②)。引数のFillPatternType.LESS_DOTSは塗りつぶしパターンを表す定数です。指定可能なパターンを以下に示します。

ALT_BARS	BIG_SPOTS	BRICKS	DIAMONDS	FINE_DOTS
LEAST_DOTS	LESS_DOTS	NO_FILL	SOLID_FOREGROUND	SPARSE_DOTS
SQUARES	THICK_BACKWARD_DIAG	THICK_FORWARD_DIAG	THICK_HORZ_BANDS	THICK_VERT_BANDS
THIN_BACKWARD_DIAG	THIN_FORWARD_DIAG	THIN_HORZ_BANDS	THIN_VERT_BANDS	

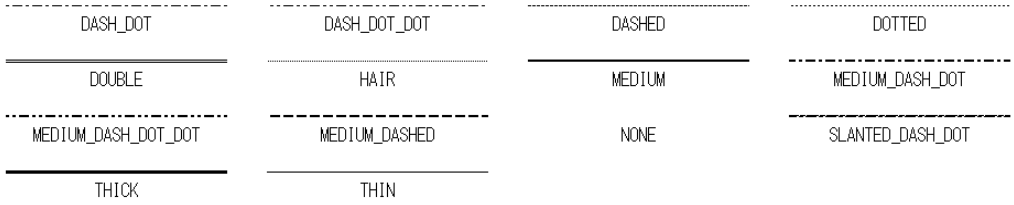
上記はLibreOffice Calcで表示した場合です。Excelでは以下のようになります。

このように見栄えが少々異なります(Excelの方が正確)。

次のsetBorderTop(), setBorderRight(), setBorderBottom(), setBorderLeft()はセルの上、右、下、左の罫線を設定します(③)。引数のBorderStyle.HAIRは罫線の種類です。指定可能な線種を以下に示します。



上記はLibreOffice Calcで表示した場合です。Excelでは以下のようになります。



これも若干見栄えが異なるようです。

setAlignment()は文字の配置を決めます(④)。指定可能な引数は以下の通りです。

・HorizontalAlignment.LEFT：左寄せ
・HorizontalAlignment.CENTER：中央寄せ
・HorizontalAlignment.RIGHT：右寄せ

CellStyleの定義(3)

文字の下線はFont#setUnderline()で設定します。

```
// フォント12 + 下線 + 太字
:
font12_underline_bold.setUnderline(Font.U_SINGLE);
```

Font.U_SINGLEで1本、Font.U_DOUBLEなら2本引かれます。

CellStyleの定義(4)

最後は文字色です。Font#setColor()でshort型引数を指定します。

```
// 罫線 上右下左 + 文字中央寄せ + フォント12 + 赤 + 太字
:
font12_red_bold.setColor(IndexedColors.RED.getIndex());
```

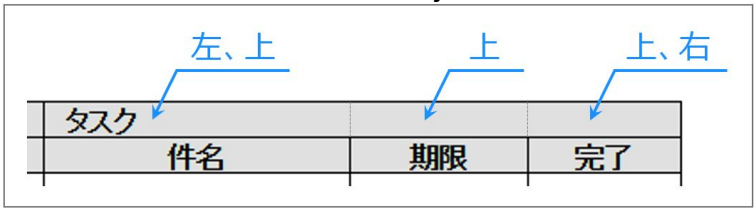
IndexedColorsは、使用可能な色を表す列挙型のクラスです。getIndex()で対応するshort値を取得し、それをsetColor()でCellStyleオブジェクトにセットします。以下は、IndexedColorsで定義されている色の一覧です。

AQUA	AUTOMATIC	BLACK	BLACK1	BLUE
BLUE_GREY	BLUE1	BRIGHT_GREEN	BRIGHT_GREEN1	BROWN
CORAL	CORNFLOWER_BLUE	DARK_BLUE	DARK_GREEN	DARK_RED
DARK_TEAL	DARK_YELLOW	GOLD	GREEN	GREY_25_PERCENT
GREY_40_PERCENT	GREY_50_PERCENT	GREY_80_PERCENT	INDIGO	LAVENDER
LEMON_CHIFFON	LIGHT_BLUE	LIGHT_CORNFLOWER_BLUE	LIGHT_GREEN	LIGHT_ORANGE
LIGHT_TURQUOISE	LIGHT_TURQUOISE1	LIGHT_YELLOW	LIME	MAROON
OLIVE_GREEN	ORANGE	ORCHID	PALE_BLUE	PINK
PINK1	PLUM	RED	RED1	ROSE
ROYAL_BLUE	SEA_GREEN	SKY_BLUE	TAN	TEAL
TURQUOISE	TURQUOISE1	VIOLET	WHITE	WHITE1
YELLOW	YELLOW1			

> 色の見栄えはLibreOffice CalcとExcelで同じになるようです。

CellStyleの組み合わせ

後はこれらを組み合わせてCellStyleオブジェクトを作成していきます。



例えば上記のタスクという部分は、罫線を左と上、上のみ、上と右に設定した3つのセルで構成しています。これに合わせてCellStyleオブジェクトも、以下の3パターン定義しています。

- pattern_borderLeftTop_fontSize12_bold
- pattern_borderTop_fontSize12_bold
- pattern_borderTopRight_fontSize12_bold

Sheetの設定

ここから実際にExcelファイルを作成していきます。まずWorkbook内にSheetオブジェクトを作ります。

```
        // private final int COL_A = 0;
        private final int COL_B = 1;
        private final int COL_C = 2;
        private final int COL_D = 3;
        private final int COL_E = 4;
        private final int COL_F = 5;
        private final int COL_G = 6;
        private final int COL_H = 7;
        private final int COL_I = 8;
                :
        private final int[] COLUMN_WIDTH = {
                0, 256 * 24, 256 * 12, 256 * 12, 256 * 14, 256 * 12, 256 * 24, 256 * 14, 256
 * 12 };
                :
                // シート作成
                Sheet sheet = workbook.createSheet("ToDo List");    // ①
                // 枠線を消去
                sheet.setDisplayGridlines(false);    // ②
                // 列幅設定
                for (int col = COL_B; col < COLUMN_WIDTH.length; col++) {
                        sheet.setColumnWidth(col, COLUMN_WIDTH[col]);    // ③
                }
```

①Workbook#createSheet()

・Sheetオブジェクトを作成する。引数はシート名。

②Sheet#setDisplayGridlines()

・シートに枠線を表示するかどうかの設定。デフォルトはtrue(表示)。

③Sheet#setColumnWidth()

・列の幅を設定する。

・1番目の引数 - 列指定

0始まりの数字で表す(A列は0, B列は1, ...)。

→TodoExcelではコードを読みやすくするため COL_B～COL_Iという定数を定義している。

・2番目の引数 - 列幅

単位は文字幅の1/256(1文字分=256, 2文字分=512,...)。

→【図20-2】で示した列幅を配列COLUMN_WIDTHに256 * 文字数 という形で定義している。

Cellの設定

TodoExcelではCellに値と書式を設定するsetCellValue()というヘルパーメソッドを作成しています。

```
// タイトル設定
setCellValue(sheet, 1, COL_B, "ToDo List", fontSize16_bold);
```

引数はシート、行番号(0～)、列番号(0～)、設定する文字列、セルの書式です。これを以下のように処理しています。

```
// 指定されたシート(行番号, 列番号）のセルに文字列、セルスタイルを設定する
    private void setCellValue(Sheet sheet, int rowIndex, int columnIndex, String text,
                                        CellStyle cellStyle) {
        Cell cell = setCellValue(sheet, rowIndex, columnIndex, text);    // ①
        cell.setCellStyle(cellStyle);    // ⑩
    }

// 指定されたシート(行番号, 列番号）のセルに文字列を設定する
    private Cell setCellValue(Sheet sheet, int rowIndex, int columnIndex, String text) {
        Cell cell = getCell(sheet, rowIndex, columnIndex);    // ②
        cell.setCellValue(text);    // ⑧
        return cell;    // ⑨
    }

// 指定されたシート(行番号, 列番号）のセルを返す
    private Cell getCell(Sheet sheet, int rowIndex, int columnIndex) {
```

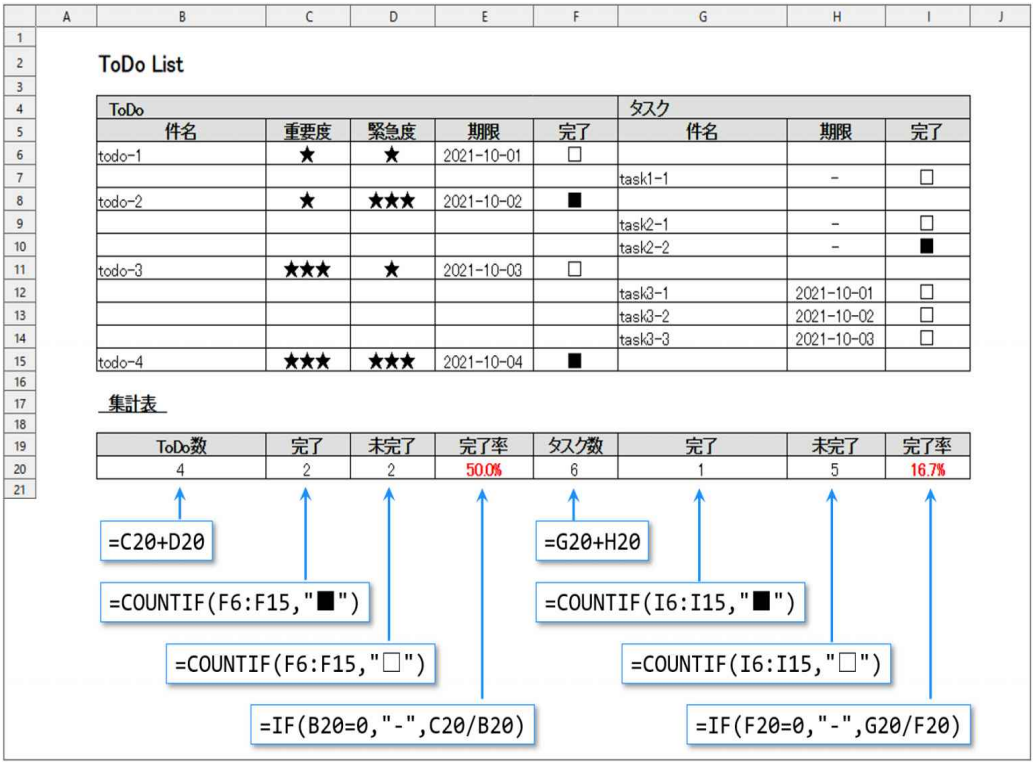
```
        // シートから行を取得（無ければ作成）
        Row row = sheet.getRow(rowIndex);   // ③
        if (row == null) {
            row = sheet.createRow(rowIndex);   // ④
        }
        // 行からセルを取得（無ければ作成)
        Cell cell = row.getCell(columnIndex);   // ⑤
        if (cell == null) {
            cell = row.createCell(columnIndex);   // ⑥
        }
        return cell;   // ⑦
    }
```

①setCellValue()を呼び出す。引数はシート、行番号、列番号、設定する文字列。
②getCell()を呼び出す。引数はシート、行番号、列番号。
③シートから指定された行(Rowオブジェクト)を取得する。
④行が無い場合(=まだその行が作成されていない)、シートにその行を作成する。
⑤行から指定された列のセル(Cellオブジェクト)取得する。
⑥セルが無い場合(=まだそのセルが作成されていない)、行にそのセルを作成する。
⑦取得(または作成)したCellオブジェクトを返す。
⑧②の結果(=⑦でreturnしたCell)に、文字列をセットする。
⑨Cellオブジェクトを返す。
⑩①の結果(=⑨でreturnしたCell)に、セル書式をセットする。

ポイントは③～⑥です。ここで指定された行→セルを取得し、無ければ作成します。前述のSampleExcelはセルB2,B3を1回だけ操作するので、直接createRow()/createCell()を呼び出しました。しかし一般的には、このように行/セルの有無をチェックしながら処理した方が良いでしょう。このgetCell()のようなメソッドがあれば、呼び出し側はそのセルが作成されているかどうか気にする必要が無くなります。

20.3 計算式の設定

集計表の各数値はセルの計算式で求めます。この計算式はToDo Listの件数によって変える必要があります。例えばToDoとタスクが合わせて10件あれば、B20～I20に以下のような計算式を設定します。



【図20-3】計算式の設定例

計算式はCell#setCellFormula()で設定します。上図のようにセルB20に計算式=C20+D20を設定するときは次のようにします。

```
getCell(sheet, 19, 1).setCellFormula("C20+D20");
```

※設定する計算式には"="をつけない
※設定先のセルを指定する行番号、列番号は0から始まることに注意

> 計算式はA1形式です。Apache POIはR1C1形式をサポートしていません。

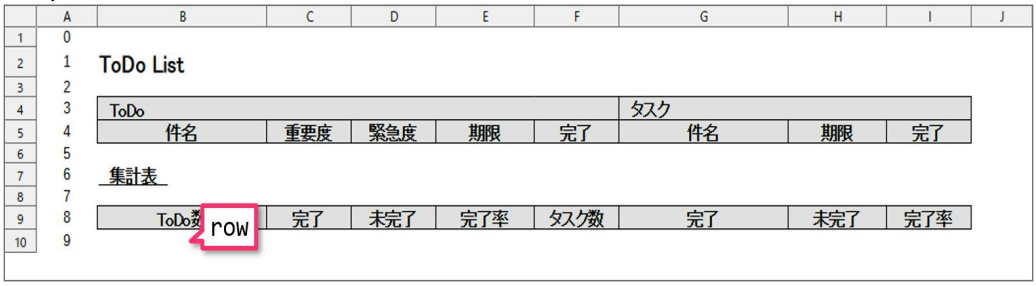
集計表の計算では、以下の点に留意する必要があります。

(1)ToDoデータが無かった場合
(2)ToDoデータの件数は変動する

(1)ToDoデータが無かった場合

計算式は不要とし、各セルに"-"をセットすることにします。

TodoExcelでは、データが無かった場合、次に値をセットする行番号を示す変数rowは9(=10行目)になっています。



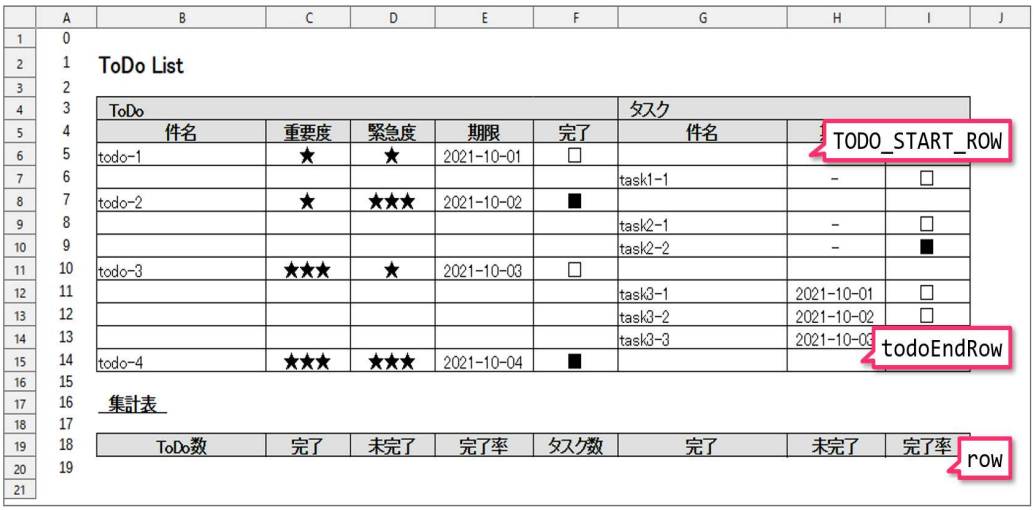
※A列の値は、0から始まる行番号を表しています。

そこでこのrowを使って、B10～I10に"-"をセットします。

```
if (todoList.size() == 0) {
        // Todoが無かった場合
        for (int col = COL_B; col <= COL_I; col++) {
                setCellValue(sheet, row, col, "-", borderAll_fontSize12_center);
        }
} else {
    :
}
```

(2)ToDoデータの件数は変動する

下図は集計表の見出し(9行目)作成が終わった状態です。計算式を設定するのはrow行目です。またToDoデータはTODO_START_ROW～todoEndRow行にあります。これを基にして計算式を作ります。



※A列の値は、0から始まる行番号を表しています。

B列：ToDo数

完了数 + 未完了数 なので上図の場合、"C20+D20"です。

変数rowで表すと "C" + (row+1) + "+D" + (row + 1) です。

これをそのまま設定しても良いのですが、String#format()を使い計算式を把握しやすくします。

```
getCell(sheet, row, COL_B).setCellFormula("C" + (row + 1) + " + D" + (row + 1));
 ↓
getCell(sheet, row, COL_B).setCellFormula(String.format("C%d + D%d", row + 1, row + 1));
 ↓
getCell(sheet, row, COL_B).setCellFormula(String.format("C%d + D%<d", row + 1));
```

C列：完了ToDo数

F列にある"■"の個数なので、COUNTIF(F6:F15,"■")で求められます。

TODO_START_ROW,todoEndRowを使って表すと以下のよう式になります。

```
String.format("COUNTIF(F%d:F%d,¥"■¥")", TODO_START_ROW + 1, todoEndRow + 1)
```

D列：未完了ToDo数

C列の"■"を"□"へ変更するだけです。

E列：ToDo完了率

C20/B20ですが、B20=0の場合に備えIFでチェックします。

```
String.format("IF(B%d=0,¥"-¥",C%<d/B%<d)", row + 1)
```

F～I列：

考え方はB～E列と同じです。

また完了率のセルには、DataFormatを使い「%形式で小数第一位まで表示」する書式を設定します。

```
setCellStyle(sheet, row, COL_E, borderAll_fontSize12_center_red_bold_percent);
setCellStyle(sheet, row, COL_I, borderAll_fontSize12_center_red_bold_percent);
```

CellStyleでは以下のように定義します。

```
// 罫線 上右下左 + 文字中央寄せ + フォント12 + 赤 + 太字 + %
CellStyle borderAll_fontSize12_center_red_bold_percent =
workbook.createCellStyle();
 :
DataFormat format = workbook.createDataFormat(); // ①
short percent = format.getFormat("0.0%"); // ②
borderAll_fontSize12_center_red_bold_percent.setDataFormat(percent); // ③
```

まずDataFormatオブジェクトをWorkbook#createDataFormat()で作成します(①)。これにgetFormat()で表示形式を与えると、対応するshort型の値が返されます(②)。CellStyle オブジェクトには、このshort値をsetDataFormat()でセットします。

最後に設定した計算式を実行します。

```
// サーバー側で再計算
workbook.getCreationHelper().createFormulaEvaluator().evaluateAll();
```

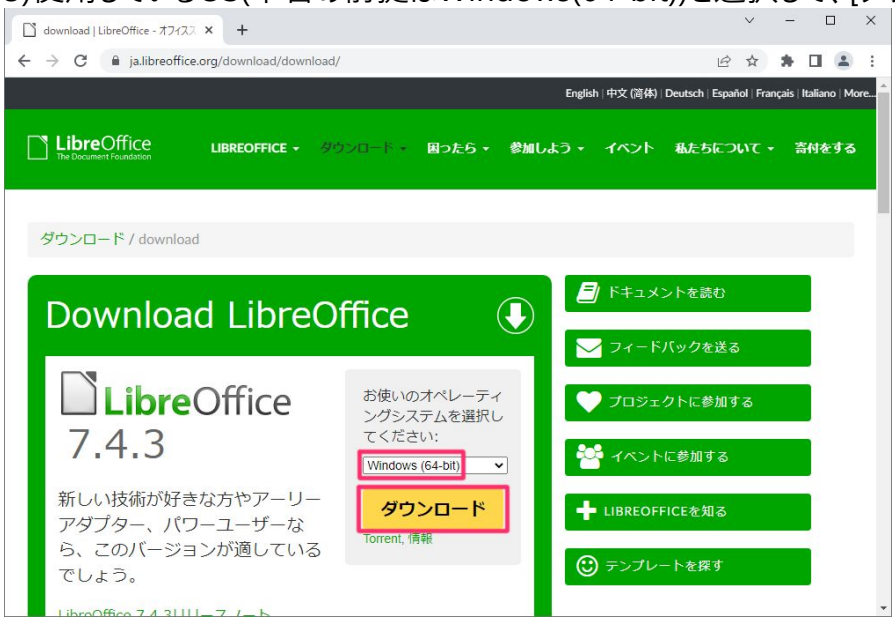
Excelは計算式を入力すると即計算しますが、Apache POIはそうなりません。そこで上記のコードを実行して、設定した計算式を再計算させます。

補足：LibreOfficeインストール方法

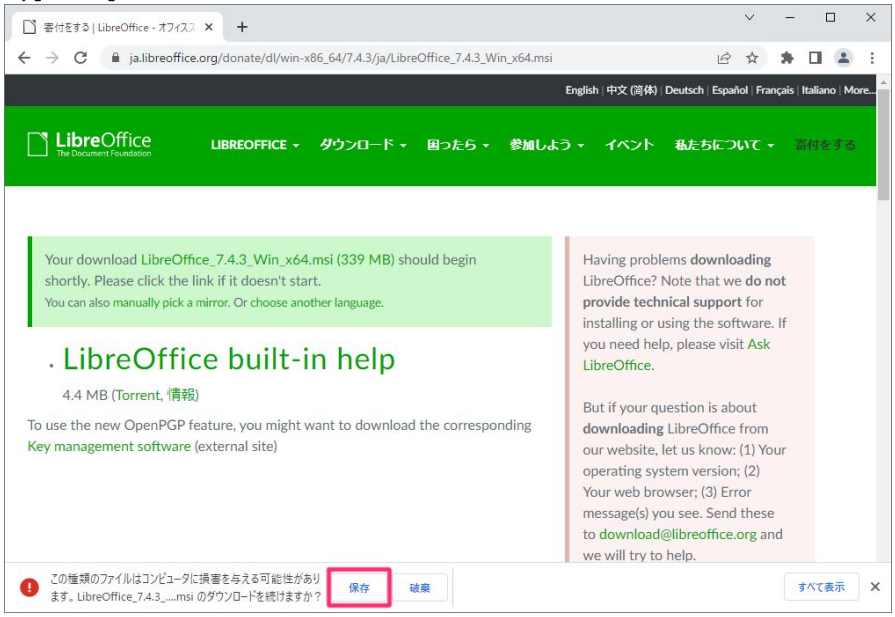
1)ブラウザでLibreOfficeのページを開く。
https://ja.libreoffice.org/

2)[ダウンロード]をクリックする。

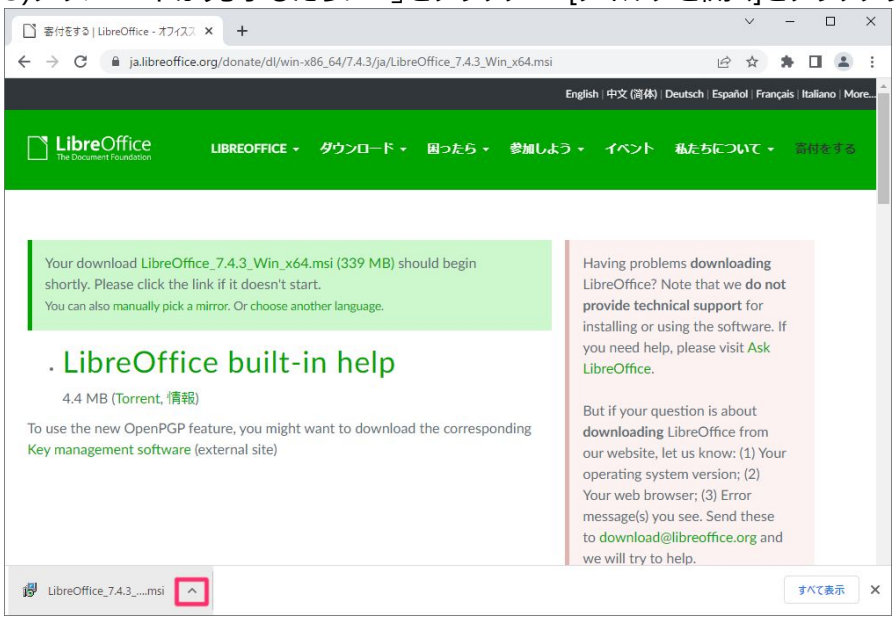
3)使用しているOS(本書の前提はWindows(64-bit))を選択して、[ダウンロード]をクリックする。



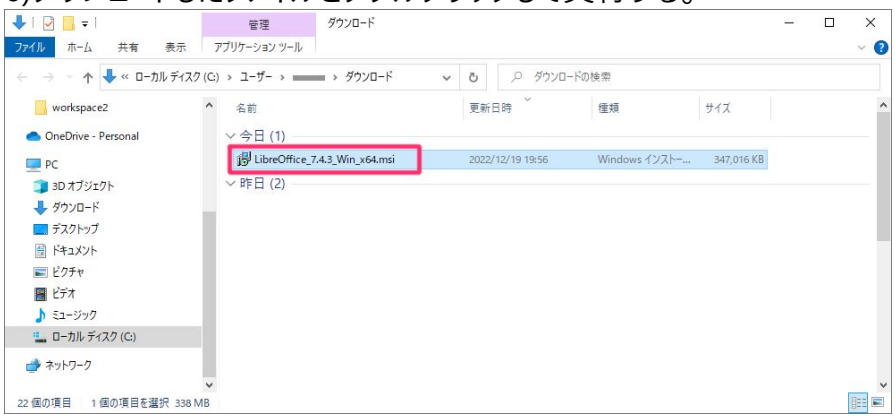
4)[保存]ボタンをクリックする。



5)ダウンロードが完了したら「 ^ 」をクリック > [フォルダを開く]をクリックする。

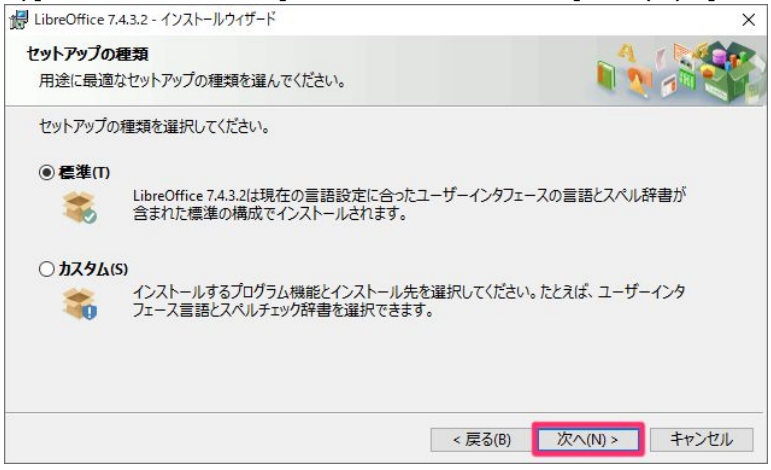


6)ダウンロードしたファイルをダブルクリックして実行する。

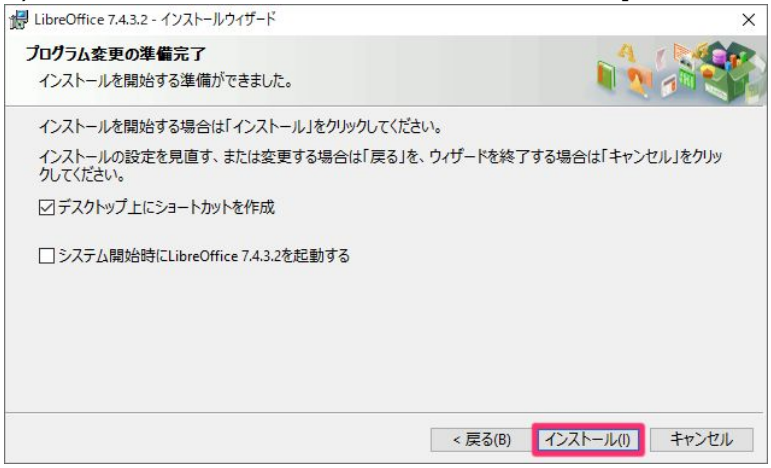


7)「インストールウィザード」で[次へ(N)>]ボタンをクリックする。

8)[セットアップの種類]は標準のままでOK > [次へ(N)>]ボタンをクリックする。

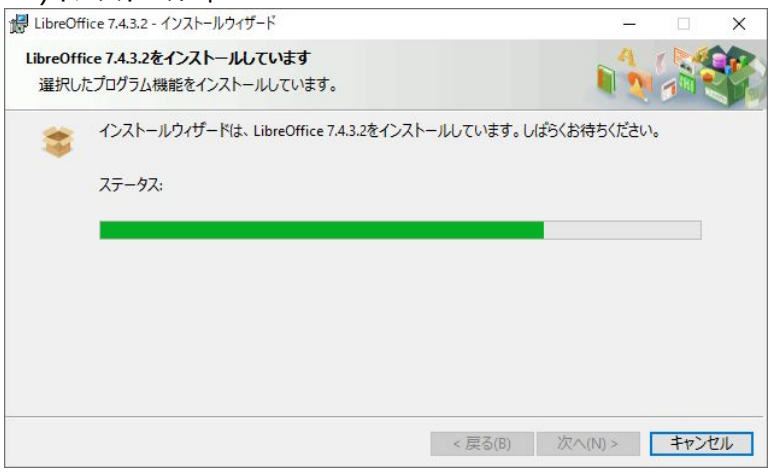


9)「プログラム変更の準備完了」もそのままでOK [インストール(I)>]ボタンをクリックする。



10)「このアプリがデバイスに変更を加えることを許可しますか？」のダイアログが表示された[はい]をクリックする。

11)インストール中

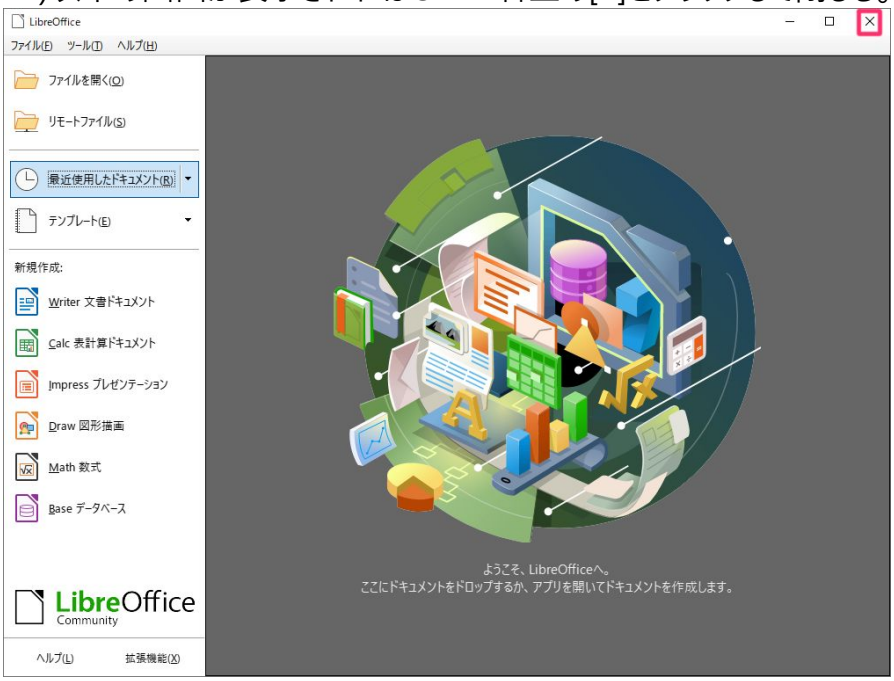


12)インストールが完了したら[完了(F)]ボタンをクリックする。


LibreOffice 7.4.3.2 - インストールウィザード
インストールウィザードが完了しました
LibreOffice 7.4.3.2が正常にインストールされました。「完了」をクリックしてウィザードを終了してください。
< 戻る(B)
完了(F)
キャンセル

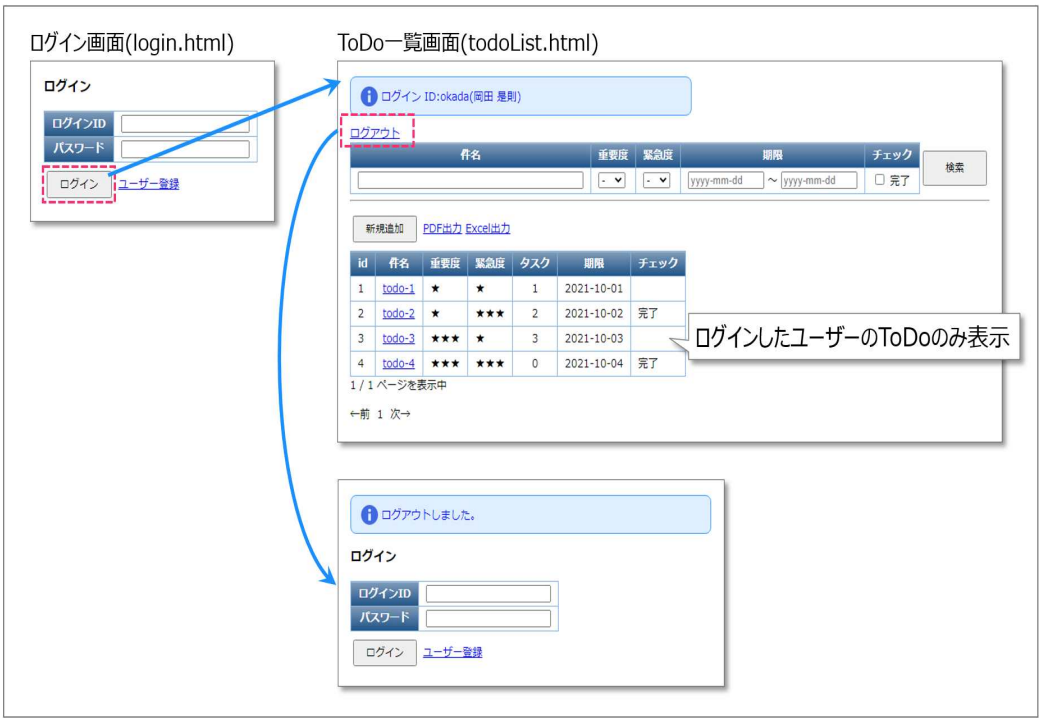
13)デスクトップ上に作成されたアイコンをクリックしてLibreOfficeを起動する。

14)以下の画面が表示されればOK > 右上の[X]をクリックして閉じる。



21. ログイン認証

本章ではToDoアプリにログイン画面を追加し、ログインに成功したら、その人のデータのみ表示します。このためToDoに「所有者(owner;オーナー)」という考え方を取り入れます。またToDoアプリの利用者を追加する画面も設けます。



【図21-1】ログイン/ログアウト操作

> 本章のタイトルは「ログイン**認証**」ですが、これとよく似た言葉に「**認可**」というものがあります。それぞれ以下のような意味を持っています。
>
> **・認証(Authentication)**
>
> 通信相手が誰か確認すること。
>
> 本章の場合、ログインに成功したら(=IDとパスワードを知っている)その人だと認める(特定する)、
>
> という意味で「ログイン認証」としています。
>
> **・認可(Authorization)**
>
> ある条件下でリソースへのアクセス権限を与えること。
>
> 本章の場合、「自分が所有するもの」という条件でToDoを操作できるようにする、ということです。

> 実システムではもっと複雑ですが、認証/認可という考えがある、ということは覚えておいた方が良いでしょう。

> Spring Bootには認証/認可を管理する「Spring Security」という機能があります。これを使えば、本章と次章の大半は設定だけで実現できます。しかしその一方、理解するのに骨が折れる機能でもあります。基礎知識が無いとなおさら大変です。このToDoアプリで認証/認可がどういうものか学んでからチャレンジすることをお勧めします。

作成するプロジェクトの仕様

プロジェクト名	Todolist14
作成ファイル	com.example.todolist.controller.LoginController.java com.example.todolist.entity.Account.java com.example.todolist.form.LoginData.java com.example.todolist.form.RegistData.java com.example.todolist.repository.AccountRepository.java com.example.todolist.service.LoginService.java src/main/resources/templates/loginForm.html src/main/resources/templates/registForm.html

```
Todolist14
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist14Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.config
 │    │    └ LoginCheckFilter.java
 │    ├ com.example.todolist.controller
 │    │    ├ DownloadController.java
 │    │    ├ LoginController.java(★)
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java(▲)
 │    │    └ TodoDaoImpl.java(▲)
 │    ├ com.example.todolist.entity
 │    │    ├ Account.java(★)
 │    │    ├ AttachedFile.java
 │    │    ├ Task.java
 │    │    └ Todo.java(▲)
```

```
│   ├ com.example.todolist.form
│   │   ├ AttachedFileData.java
│   │   ├ LoginData.java(★)
│   │   ├ RegistData.java(★)
│   │   ├ TaskData.java
│   │   ├ TodoData.java
│   │   └ TodoQuery.java
│   ├ com.example.todolist.repository
│   │   ├ AccountRepository.java(★)
│   │   ├ AttachedFileRepository.java
│   │   ├ TaskRepository.java
│   │   └ TodoRepository.java(▲)
│   ├ com.example.todolist.service
│   │   ├ DownloadService.java
│   │   ├ LoginService.java(★)
│   │   └ TodoService.java
│   └ com.example.todolist.view
│       ├ TodoExcel.java
│       └ TodoPdf.java
└ src/main/resources
    ├ i18n
    │   ├ FixedDisplayStrings.properties
    │   ├ FixedDisplayStrings_en.properties(▲)
    │   ├ FixedDisplayStrings_ja.properties(▲)
    │   ├ OperationMessages_en.properties(▲)
    │   ├ OperationMessages_ja.properties(▲)
    │   ├ ValidationMessages_en.properties(▲)
    │   └ ValidationMessages_ja.properties(▲)
    :
    ├ templates
    │   ├ fragments.html
    │   ├ loginForm.html(★)
    │   ├ registForm.html(★)
    │   ├ todoForm.html(▲)
```

```
    │    └ todoList.html(▲)
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

21.1 ユーザー管理テーブル

ログイン機能の前に、ユーザーを管理するaccountテーブルを追加します。

【リスト21-1】create_account.sql

```
DROP TABLE account;
CREATE TABLE account
(
	id		SERIAL PRIMARY KEY,
	login_id	TEXT UNIQUE,
	name		TEXT,
	password	TEXT
);

INSERT INTO account(login_id, name, password) VALUES('okada', '岡田 是則', 's6rizqfk');
INSERT INTO account(login_id, name, password) VALUES('inoue', '井上 俊憲', 'g73phw5n');
INSERT INTO account(login_id, name, password) VALUES('inagaki', '稲垣 絵美', 's59mrtw3');
```

【表21-1】accountテーブルの形式

列名	内容	データ型	制約	備考
id	1～	SERIAL	PRIMARY KEY	連番(自動採番)
login_id	ログインID	TEXT	UNIQUE	重複不可
name	氏名	TEXT		
password	パスワード	TEXT		

ユーザーがログイン画面で入力するIDはlogin_idに格納します。これはユーザー毎に異なるものなので、login_id列には、重複した値の格納を防ぐ**UNIQUE(ユニーク)制約**を設定します。こうすると【リスト21-1】実行後、同じ値を登録しようとすると以下のようにエラーとなります('okada'という値は、login_id列にあるため)。

> UNIQUE制約はPRIMARY KEY制約と似ていますが、以下の点が異なります。
> ・UNIQUE制約の列にはNULLを格納できる(←→PRIMARY KEY制約の列にはNULLを格納できない)
> ・UNIQUE制約はテーブルに複数設定できる(←→PRIMARY KEY制約はテーブルに1つだけ)

> ToDoアプリではパスワードをそのままaccount.passwordに格納します。しかし実際のシステムでは、万が一パスワードが漏洩しても解読できないようハッシュ化するなどの対策が必要です。興味がある方は調べてみてください。

対応するエンティティクラスは以下のようにします

【リスト21-2】com.example.todolist.entity.Account.java

```
package com.example.todolist.entity;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "account")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;

    @Column(name = "login_id")
    private String loginId;

    @Column(name = "name")
    private String name;

    @Column(name = "password")
    private String password;
```

```
}
```

またリポジトリにはloginIdをキーにして検索するfindByLoginId()を宣言します。これで入力されたログインIDがaccountテーブルに存在するかチェックします。

【リスト21-3】com.example.todolist.repository.AccountRepository.java

```
package com.example.todolist.repository;

import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.Account;

@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {
	Optional<Account> findByLoginId(String loginId);
}
```

loginIdはUNIQUE制約を設定したlogin_id列に対するものなので、findByLoginId()の結果は 1 件または 0 件(=存在しない)です。そこで戻り値の型をOptional<Account>にします。これで該当するloginIdが無ければnullが返されます。

21.2 ログイン画面

次にログイン画面です。パスワード欄のinput要素がtype="password"であることに留意してください。

【リスト21-4】src/main/resources/templates/loginForm.html

```
<!doctype html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <!-- メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <h3 th:text="#{text.login}"></h3>
  <form th:action="@{/login/do}" method="post" th:object="${loginData}">
    <table>
      <!-- ユーザーID -->
      <tr>
        <th th:text="#{label.loginId}"></th>
        <td>
          <input type="text" name="loginId">
          <div th:if="${#fields.hasErrors('loginId')}"
            th:errors="*{loginId}" th:errorclass="red"></div>
        </td>
      </tr>
      <!-- パスワード -->
      <tr>
        <th th:text="#{label.password}"></th>
        <td>
          <input type="password" name="password">
          <div th:if="${#fields.hasErrors('password')}"
```

```
                th:errors="*{password}" th:errorclass="red"></div>
            </td>
          </tr>
        </table>
        <!-- ログインボタン -->
        <button type="submit" th:text="#{button.login}"></button>
        <!-- 新規登録リンク -->
        <a th:href="@{/regist}" th:text="#{text.register}"></a>
    </form>
</body>
</html>
```

またToDo一覧画面(todoList.html)、Todo入力画面(todoForm.html)にログアウトのリンクを追加します。

【リスト21-5】src/main/resources/templates/todoList.html(追加箇所)

```
<!DOCTYPE html>
:
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- ログアウトリンク -->
  <a th:href="@{/logout}" th:text="#{text.logout}"></a><!-- 追加 -->
  <!-- 検索条件入力エリア -->
  <form th:action="@{/}" method="post" th:object="${todoQuery}">
    :
```

【リスト21-6】src/main/resources/templates/todoForm.html(追加箇所)

```
<!DOCTYPE html>
:
<body>
  <!-- 操作メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <!-- ログアウトリンク -->
  <a th:href="@{/logout}" th:text="#{text.logout}"></a><!-- 追加 -->
  <!-- 入力フォーム -->
  <form th:action="@{/}" method="post" th:object="${todoData}">
    :
```

【リスト21-7】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
label.loginId=ログインID
label.password=パスワード
button.login=ログイン
button.register=登録
text.login=ログイン
text.logout=ログアウト
```

```
text.register=ユーザー登録
```

※button.registerは本章後半の【リスト21-27】で使います。

【リスト21-8】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
label.loginId=Login ID
label.password=Password
button.login=Login
button.register=Register
text.login=Login
text.logout=Logout
text.register=User registration
```

このログイン画面に対応するフォームクラスを作成します。

【リスト21-9】com.example.todolist.form.LoginData.java

```
package com.example.todolist.form;

import jakarta.validation.constraints.NotBlank;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
public class LoginData {
	@NotBlank
	 private String loginId;

	@NotBlank
	private String password;
}
```

単純にユーザーID,パスワードを保持するだけです。@NotBlankにより未入力であればエラーとします。

【リスト21-10】src/main/resources/i18n/ValidationMessages_ja.properties(追加分)

```
NotBlank.loginData.loginId=入力されていません
NotBlank.loginData.password=入力されていません
NotFound.loginData.loginId=ユーザーIDまたはパスワードが一致しません
NotFound.loginData.password=ユーザーIDまたはパスワードが一致しません
```

※NotFound.loginData.loginId, NotFound.loginData.passwordはサービスクラスで使います。

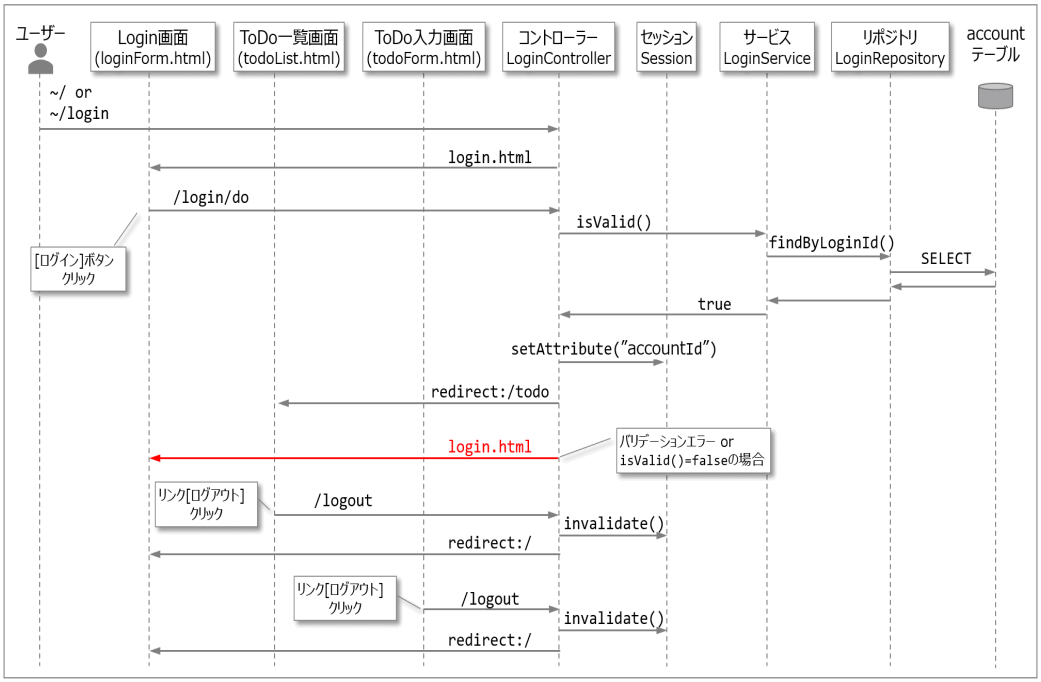
【リスト21-11】

src/main/resources/i18n/ValidationMessages_en.properties(追加分)

```
NotBlank.loginData.loginId=Not entered.
NotBlank.loginData.password=Not entered.
NotFound.loginData.loginId=User ID or password does not match.
NotFound.loginData.password=User ID or password does not match.
```

最後にログイン/ログアウトを制御するコントローラーLoginControllerを新規作成します。TodolistControllerにハンドラーメソッドを追加する方法もありますが、ToDoデータの操作(検索、追加、更新、削除)とLogin操作(認証)は目的が異なるので新設します。

このコントローラーに必要な処理をシーケンス図で表すと次のようになります。



【図21-2】ログイン/ログアウトに関連するシーケンス

ログインに成功したら、セッションに"accountId"というデータを格納します。そしてログアウト時に削除します。これで"accountId"がセッションにあるかどうかで、ログインしているかどうかがわかります。

コントローラーは以下のようになります。

【リスト21-12】com.example.todolist.controller.LoginController.java

```
package com.example.todolist.controller;

import java.util.Locale;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
```

```java
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import com.example.todolist.common.OpMsg;
import com.example.todolist.entity.Account;
import com.example.todolist.form.LoginData;
import com.example.todolist.repository.AccountRepository;
import com.example.todolist.service.LoginService;
import jakarta.servlet.http.HttpSession;
import lombok.RequiredArgsConstructor;

@Controller
@RequiredArgsConstructor
public class LoginController {
	private final AccountRepository accountRepository;
	private final LoginService loginService;
	private final MessageSource messageSource;
	private final HttpSession session;

	// Login画面表示
	@GetMapping("/")
	public ModelAndView showLogin(ModelAndView mv) {
		mv.setViewName("loginForm");
		mv.addObject("loginData", new LoginData());
		return mv;
	}

	@GetMapping("/login")
	public ModelAndView login(ModelAndView mv) {
		mv.setViewName("loginForm");
		mv.addObject("loginData", new LoginData());
		return mv;
	}

	// ログインボタン押下
```

```
    @PostMapping("/login/do")
    public String login(@ModelAttribute @Validated LoginData loginData,
                        BindingResult result,
                        Model model,
                        RedirectAttributes redirectAttributes,
                        Locale locale) {
        // バリデーション
        if (result.hasErrors()) {
            // エラーあり -> エラーメッセージをセット
            String msg
                =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
            model.addAttribute("msg", new OpMsg("E", msg));
            return "loginForm";
        }

        // サービスでチェック
        if (!loginService.isValid(loginData, result, locale)) {
            // エラーあり -> エラーメッセージをセット
            String msg
                =
messageSource.getMessage("msg.e.input_something_wrong", null, locale);
            model.addAttribute("msg", new OpMsg("E", msg));
            return "loginForm";
        }

        // (念のため)セッション情報をクリアする
        session.invalidate();

        // LoginしたユーザーのaccountIdをセッションへ格納する
        Account account =
accountRepository.findByLoginId(loginData.getLoginId()).get();
        session.setAttribute("accountId", account.getId()); // ①
```

```
            // Login成功メッセージをセットしてリダイレクト
            String msg = messageSource.getMessage(
                    "msg.i.login_successful",
                    new Object[] { account.getLoginId(), account.getName() },
                    locale);
            redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
            return "redirect:/todo";
        }

        // Logout処理
        @GetMapping("/logout")
        public String logout(RedirectAttributes redirectAttributes, Locale locale) {
            // セッション情報をクリアする
            session.invalidate(); // ②

            // Logout完了メッセージをセットしてリダイレクト
            String msg = messageSource.getMessage("msg.i.logout_successful",
null, locale);
            redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
            return "redirect:/";
        }

}
```

【リスト21-13】src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.login_successful=ログイン ID:{0}({1})
msg.i.logout_successful=ログアウトしました。
```

【リスト21-14】src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.login_successful=Logged in ID:{0}({1})
msg.i.logout_successful=Logged out
```

ログイン成功時、そのユーザーのaccount.idを"accountId"という名前でセッションへ格納してから(①)、ToDo一覧画面を表示します。

```
    // Loginしたユーザーのaccount.idをセッションへ格納する
    Account account = accountRepository.findByLoginId(loginData.getLoginId()).get();
    session.setAttribute("accountId", account.getId()); // ①
    :
    return "redirect:/todo";
```

また[ログアウト]リンクに対応するlogout()では、セッションを破棄してクリアします(②)。

```
    // セッション情報をクリアする
    session.invalidate(); // ②
```

入力されたログインIDの存在チェック、パスワードチェックはサービスクラスで行います。

【リスト21-15】com.example.todolist.service.LoginService.java

```
package com.example.todolist.service;

import java.util.Locale;
import java.util.Optional;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Service;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import com.example.todolist.entity.Account;
import com.example.todolist.form.LoginData;
import com.example.todolist.repository.AccountRepository;
import lombok.AllArgsConstructor;

@Service
@AllArgsConstructor
public class LoginService {
    private final MessageSource messageSource;
    private final AccountRepository accountRepository;
```

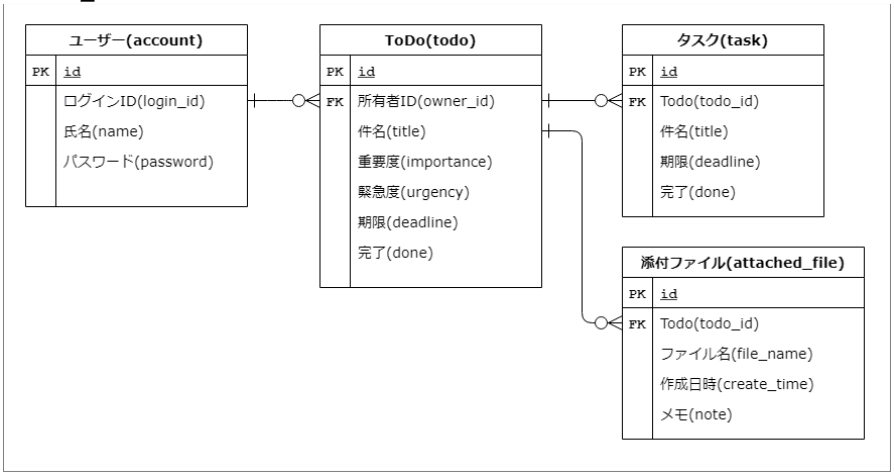
```
    // Loginチェック
    public boolean isValid(LoginData loginData, BindingResult result, Locale
locale) {
        // ログインIDが登録されているか？
        Optional<Account> account =
accountRepository.findByLoginId(loginData.getLoginId());
        if (account.isEmpty()) {
            // 登録されていない
            FieldError fieldError = new FieldError(
                result.getObjectName(),
                "password",
                messageSource.getMessage("NotFound.loginData.loginId",
null, locale));
            result.addError(fieldError);
            return false;
        }

        // パスワードチェック
        if(!account.get().getPassword().equals(loginData.getPassword())){
            // 登録されているものと違う
            FieldError fieldError = new FieldError(
                result.getObjectName(),
                "password",
                messageSource.getMessage("NotFound.loginData.password",
null, locale));
            result.addError(fieldError);
            return false;
        }

        return true;
    }
}
```

これでhttp://localhost:8080/またhttp://localhost:8080/loginへアクセスするとログイン画面が表示されます。また前述したように「セッションに"accountId"というデータがあれば、ログイン認証が済んでいる」ということがわかります。これは、次節で利用します。

21.3 表示データの限定

前節までででaccountテーブルに登録されたログインID/パスワードに基づいて認証できるようになりました。しかしログイン後表示されるToDo一覧には、他ユーザーのToDoも含まれています。これをログインした人のものだけにします。

最初にtodoテーブルへ「誰のToDoか?」を表すINTEGER型のowner_id(所有者ID)列を追加します。ここにはaccount.idの値を格納し、accountテーブルとtodoテーブルを1:nの関係にします。これもattached_fileと同様にアノテーションではなく、プログラムコードで表していきます。



【図21-3】本章テーブルのER図

owner_id列の追加は以下のSQLで行います。1を設定するので、既存のToDo/タスクは、すべて岡田さんのものになります(**【リスト21-1】**参照)。

【リスト21-16】alter_todo.sql

```
ALTER TABLE todo ADD COLUMN owner_id INTEGER;
UPDATE todo SET owner_id = 1;
```

todoテーブルにowner_id列を追加したので、エンティティクラスにも対応するプロパティを追加します(①)。

【リスト21-17】com.example.todolist.entity.Todo.java(追加部分)

```
package com.example.todolist.entity;
```

```
   :
public class Todo {
      :
      @Column(name = "owner_id") // ①
      private Integer ownerId; // ①

      @OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)
      @OrderBy("id asc")
      private List<Task> taskList = new ArrayList<>();
      :
}
```

そしてToDo登録時、このownerIdにログインしているユーザーのaccount.idをセットします。

【リスト21-18】
com.example.todolist.controller.TodoListController#createTodo(追加箇所)

```
      // ToDo追加処理
      @PostMapping("/todo/create/do")
      public String createTodo(@ModelAttribute @Validated TodoData todoData,
                               BindingResult result,
                               Model model,
                              RedirectAttributes redirectAttributes, Locale
locale) {
            // エラーチェック
            boolean isValid = todoService.isValid(todoData, result, true, locale);
            if (!result.hasErrors() && isValid) {
                  // エラーなし -> 追加
                  Todo todo = todoData.toTodoEntity();
                  todo.setOwnerId((Integer)session.getAttribute("accountId")); // ①
                  todo = todoRepository.saveAndFlush(todo);
                  :
            } else {
                  // エラーあり -> エラーメッセージをセット
                  :
```

```
        }
    }
```

TodoエンティティのownerIdにセッションから取得したaccountIdの値をセットします(①)。この値は**【リスト21-12】**①で設定していますがaccount.idです。これでこのToDoの所有者はログインユーザーになります。

このownerIdを使って表示データを制限する必要があるのは以下の5箇所です。
(1)ToDoの一覧表示処理
(2)ToDoの検索処理
(3)ページリンククリック時の処理
(4)[PDF出力]リンククリック時の処理
(5)[Excel出力]リンククリック時の処理

(1)ToDoの一覧表示処理

ここまで全ToDoを表示していましたが、これをownerIdで制限します。以下は変更後のハンドラーメソッドです。

【リスト21-19】
com.example.todolist.controller.TodoListController#showTodoList()

```
    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                                     @PageableDefault(page = 0, size = 5,
sort = "id")
                                     Pageable pageable) {
        :
        // Todo検索
        Integer accountId = (Integer)session.getAttribute("accountId"); // ①
        Page<Todo> todoPage
            = todoDaoImpl.findByCriteria(todoQuery, accountId, prevPageable); //
②
        mv.addObject("todoQuery", todoQuery); // 検索条件
        mv.addObject("todoPage", todoPage); // page情報
        mv.addObject("todoList", todoPage.getContent()); // 検索結果
```

```
        return mv;
    }
```

まずセッションからaccountIdを取得します(①)。これはログインしたユーザーのaccount.idの値です(【リスト21-12】①)。これをfindByCriteria()の引数に追加し、取得するToDoを限定します(②)。

そのためにfindByCriteria()の定義を変更します(※1)。

【リスト21-20】com.example.todolist.dao.TodoDao.java(変更箇所)

```
package com.example.todolist.dao;
  :
public interface TodoDao {
  // Criteria APIによる検索
  Page<Todo> findByCriteria(TodoQuery todoQuery, Integer accountId, Pageable pageable); // ※1
}
```

これに基づき実装クラスを変更します。

【リスト21-21】com.example.todolist.dao.TodoDao.TodoDaoImpl.java(変更箇所)

```
package com.example.todolist.dao;
  :
public class TodoDaoImpl implements TodoDao {
    private final EntityManager entityManager;

    // Criteria APIによる検索
    @Override
    public Page<Todo> findByCriteria(TodoQuery todoQuery,
                                     Integer accountId,    // ★1
                                     Pageable pageable) {
        // todoQueryから検索条件を設定
        :
        // 所有者
```

```
            predicates.add(
                    builder.and(builder.equal(root.get(Todo_.OWNER_ID), accountId)));
// ★2

            // SELECT作成
            Predicate[] predArray = new Predicate[predicates.size()];
            predicates.toArray(predArray);
            query =
query.select(root).where(predArray).orderBy(builder.asc(root.get(Todo_.id)));
        :
    }
}
```

findByCriteria()は、引数todoQueryの内容から検索条件を作成しpredicatesオブジェクトに設定しています(前書「10.4　Criteria APIによる動的クエリの実行」参照)。この最後に"and owner_id = accountId" に相当する条件を追加します(★1,2)。これでSELECTした結果はログインユーザーのToDoだけになります。

(2)ToDoの検索処理

[検索]ボタンクリック時の処理です。これもセッションからaccountIdを取得し(①)、findByCriteria()で絞り込みます(②)。

【リスト21-22】
com.example.todolist.controller.TodoListController#queryTodo()

```
    // ToDo検索処理
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
                                  BindingResult result,
                                  @PageableDefault(page = 0, size = 5, sort = "id")
                                  Pageable pageable,
                                  ModelAndView mv, Locale locale) {
        mv.setViewName("todoList");

        Page<Todo> todoPage = null;
        if (todoService.isValid(todoQuery, result, locale)) {
            // エラーが無ければ検索
            Integer accountId = (Integer)session.getAttribute("accountId"); // ①
            todoPage = todoDaoImpl.findByCriteria(todoQuery, accountId , pageable); // ②
            :
        } else {
            // 検索条件エラーあり -> エラーメッセージをセット
            :
        }

        return mv;
    }
```

(3)ページリンククリック時の処理

これも(1)(2)と同じ考え方です。

【リスト21-23】
om.example.todolist.controller.TodoListController#queryTodo()

```
    // ページリンク押下時
    @GetMapping("/todo/query")
    public ModelAndView queryTodo(@PageableDefault(page = 0, size = 5, sort = "id")
                                  Pageable pageable,
                                  ModelAndView mv) {
        :
          // sessionに保存されている条件で検索
          TodoQuery todoQuery = (TodoQuery)session.getAttribute("todoQuery");
          Integer accountId = (Integer)session.getAttribute("accountId");// 追加
          Page<Todo> todoPage = todoDaoImpl.findByCriteria(todoQuery, accountId , pageable);// 変更
        :
    }
```

(4)[PDF出力]リンククリック時の処理

19章でPDF/Excel出力データを検索するためTodoリポジトリにfindAllByOrderById(全ToDoのid順(昇順))を宣言しました。これを「owner_idをキーにして検索した結果のid順(昇順)」へ変更します。

【リスト21-24】com.example.todolist.repository.TodoRepository.java

```
@Repository
public interface TodoRepository extends JpaRepository<Todo, Integer> {
//    List<Todo> findAllByOrderById();
      List<Todo> findByOwnerIdOrderById(Integer accountId);
}
```

【リスト21-25】
com.example.todolist.controller.TodoListController#writeTodoPdf()

```
      // PDF生成処理
      @GetMapping("/todo/pdf")
      public TodoPdf writeTodoPdf(TodoPdf pdf) {
            Integer accountId = (Integer)session.getAttribute("accountId");    // 追加
            List<Todo> todoList =
todoRepository.findByOwnerIdOrderById(accountId);    // 変更
            pdf.addStaticAttribute("todoList", todoList);
            return pdf;
      }
```

(5)[Excel出力]リンククリック時の処理

こちらもfindByOwnerIdOrderById()を使うように変更します。

【リスト21-26】
com.example.todolist.controller.TodoListController#writeTodoExcel()

```
      // Excel生成処理
      @GetMapping("/todo/excel")
      public TodoExcel writeTodoExcel(TodoExcel excel) {
            Integer accountId = (Integer)session.getAttribute("accountId");    // 追加
            List<Todo> todoList =
todoRepository.findByOwnerIdOrderById(accountId);    // 変更
```

```
            excel.addStaticAttribute("todoList", todoList);
            excel.addStaticAttribute("fileName", "todoExcel.xlsx");

            return excel;
        }
```

これでログインしたユーザーのToDoだけを表示できるようになります。

補足

本節でtodoテーブルにowner_idを追加しましたが、この情報はid(todo.id)と同様に
・更新画面表示時todoForm.htmlへセットする
・[更新]ボタンクリック時、todoForm.htmlから取り込みtodoテーブルを更新する
という処理を行う必要があります。
そのため、以下のコードを追加してください。
→追加しないとToDoを更新した際にowner_idがnullになるため、それ以降表示されなくなってしまいます。

■com.example.todolist.form.TodoData.java(一部抜粋)

```
public class TodoData {
	private Integer id;
	private Integer ownerId;// !!!この行を追加!!!

	@NotBlank
	private String title;
	:
	public Todo toEntity() {
		Todo todo = new Todo();
		todo.setId(id);
		todo.setOwnerId(ownerId);// !!!この行を追加!!!
		todo.setTitle(title);
		:
		return todo;
	}
	public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
		this.id = todo.getId();
		this.ownerId = todo.getOwnerId();// !!!この行を追加!!!
		this.title = todo.getTitle();
		:
	}
	:
}
```

■src/main/resources/templates/todoForm.html(一部抜粋)

```
	:
<body>
	:
		■ToDo
		<!-- ToDO入力エリア -->
		<table>
		  <!-- id -->
```

```
            <tr>
              <th>id</th>
              <td>
                <span th:text="*{id}"></span>
                <!-- 更新 のために必要 -->
                <input type="hidden" th:field="*{id}">
                <input type="hidden" th:field="*{ownerId}"> <!-- !!!この行を追加!!! -->
              </td>
            </tr>
            <!-- 件名 -->
          :
```

■**src/main/resources/i18n/OperationMessages_en.properties(追加分)**

```
msg.e.operation_error=Not the operator's todo
```

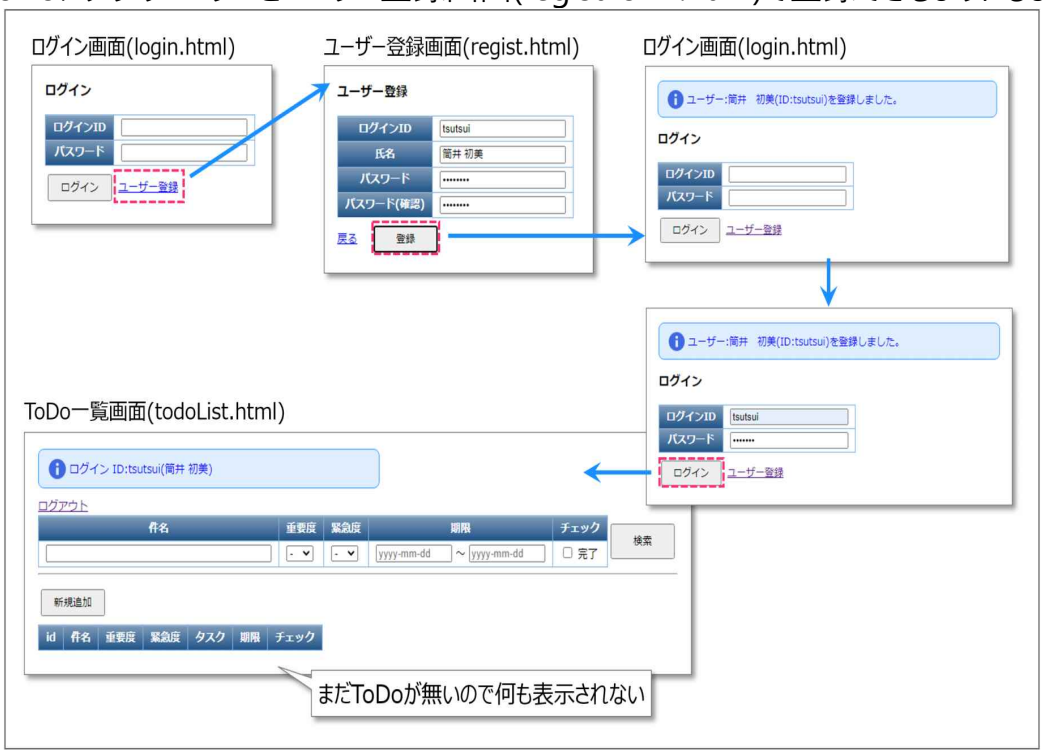
■**src/main/resources/i18n/OperationMessages_ja.properties(追加分)**

```
msg.e.operation_error=操作者のTodoではありません。
```

→この行はプログラムの異常終了を避けるために追加しています。実際にこれらのメッセージが表示されるわけではありません。

21.4 ユーザーの追加

次にToDoアプリのユーザーをユーザー登録画面(registForm.html)で登録できるようにします。



【図21-4】ユーザー登録操作

まず画面を作成します。

【リスト21-27】src/main/resources/templates/registForm.html

```
<!doctype html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <!-- メッセージエリア -->
  <div th:replace="fragments :: msg_area"></div>
  <form th:action="@{/regist/do}" method="post" th:object="${registData}">
```

```
<h3 th:text="#{text.register}"></h3>
<!-- ユーザーID -->
<table>
  <tr>
    <th th:text="#{label.loginId}"></th>
    <td>
      <input type="text" name="loginId" th:value="*{loginId}">
      <div th:if="${#fields.hasErrors('loginId')}"
        th:errors="*{loginId}" th:errorclass="red"></div>
    </td>
  <tr>
    <!-- 氏名 -->
  <tr>
    <th th:text="#{label.name}"></th>
    <td>
      <input type="text" name="name" th:value="*{name}">
      <div th:if="${#fields.hasErrors('name')}"
        th:errors="*{name}" th:errorclass="red"></div>
    </td>
  <tr>
    <!-- パスワード -->
  <tr>
    <th th:text="#{label.password}"></th>
    <td>
      <input type="password" name="password1">
      <div th:if="${#fields.hasErrors('password1')}" th:errors="*{password1}"
        th:errorclass="red"></div>
    </td>
  <tr>
    <!-- パスワード(確認) -->
  <tr>
    <th th:text="#{label.rePassword}"></th>
    <td>
      <input type="password" name="password2">
```

```
            <div th:if="${#fields.hasErrors('password2')}" th:errors="*{password2}"
              th:errorclass="red"></div>
          </td>
        <tr>
      </table>
      <!-- 登録ボタン -->
      <button type="submit" th:text="#{button.register}"></button>
      <!-- 戻るボタン -->
      <a th:href="@{/regist/cancel}" th:text="#{button.cancel}"></a>
    </form>
</body>
</html>
```

【リスト21-28】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
label.name=氏名
label.rePassword=パスワード(確認)
```

【リスト21-29】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
label.name=Name
label.rePassword=Password(re-enter)
```

対応するフォームクラスも作成します。

【リスト21-30】com.example.todolist.form.RegistData.java

```
package com.example.todolist.form;

import org.hibernate.validator.constraints.Length;
import com.example.todolist.entity.Account;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Pattern;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
```

```
@NoArgsConstructor
public class RegistData {
	//　入力必須
	@NotBlank
	private String name;

	//　入力必須、長さ=8～16文字、半角英数字
	@NotBlank
	@Length(min=8, max=16) // ①
	@Pattern(regexp = "^[a-zA-Z0-9]+$") // ②
	private String loginId;

	@NotBlank
	@Length(min=8, max=16) // ①
	@Pattern(regexp = "^[a-zA-Z0-9]+$") // ②
	private String password1;

	@NotBlank
	@Length(min=8, max=16) // ①
	@Pattern(regexp = "^[a-zA-Z0-9]+$") // ②
	private String password2;

	public Account toEntity() {
		return new Account(null, this.loginId, this.name, this.password1);
	}
}
```

loginId, password1, password2には@Length, @Patternバリデーションも付与しています。

①@Length

・入力値(文字列)の文字数がmin～maxの範囲内になければバリデーションエラー

・ここでは8～16文字以外ならエラーになる。

② @Pattern

・入力値が引数regexpで指定した**正規表現(regular expression)**と一致しない場合、バリデーションエラー
・"^[a-zA-Z0-9]+$"は「半角英数字だけで構成されている」の意味
　→空白や@などの特殊記号、全角文字などを含む場合は一致しないためエラーとなる。

> 本書では正規表現について解説しませんが、上記のようなパターンを表すのに幅広く使われています。興味がある方は調べてみてください。

一方、入力されたpassword1とpassword2が同じか？、ログインIDが未使用のものかどうか？のチェックはサービスクラスで行います。

【リスト21-31】com.example.todolist.service.LoginService.java(追加分)

```
package com.example.todolist.service;
   :
import com.example.todolist.form.RegistData;
   :
public class LoginService {
   :
      // 登録画面用のチェック
      public boolean isValid(RegistData registData, BindingResult result, Locale
locale) {
            if (!registData.getPassword1().equals(registData.getPassword2())) {
                  // パスワード不一致
                  FieldError fieldError = new FieldError(
                        result.getObjectName(),
                        "password2",
                        messageSource.getMessage("Unmatch.registData.password",
null, locale));
                  result.addError(fieldError);
                  return false;
            }

            // ログインIDがすでに使われていないか？
          Optional<Account> account=
accountRepository.findByLoginId(registData.getLoginId());
          if (account.isPresent()) {
                  // 登録されている => 使われている
                  FieldError fieldError = new FieldError(
                        result.getObjectName(),
                        "loginId",
                        messageSource.getMessage("AlreadyUsed.registData.loginId",
null, locale));
                  result.addError(fieldError);
```

```
                return false;
            }

            return true;
        }
}
```

【リスト21-32】src/main/resources/i18n/ValidationMessages_ja.properties(追加分)

```
NotBlank.registData.loginId=入力されていません
Length.registData.loginId={2}～{1}文字としてください
Pattern.registData.loginId=半角英数字で入力してください
AlreadyUsed.registData.loginId=このログインIDはすでに使われています

NotBlank.registData.name=入力されていません

NotBlank.registData.password1=入力されていません
Length.registData.password1={2}～{1}文字としてください
Pattern.registData.password1=半角英数字で入力してください

NotBlank.registData.password2=入力されていません
Length.registData.password2={2}～{1}文字としてください
Pattern.registData.password2=半角英数字で入力してください
Unmatch.registData.password=パスワードが一致しません
```

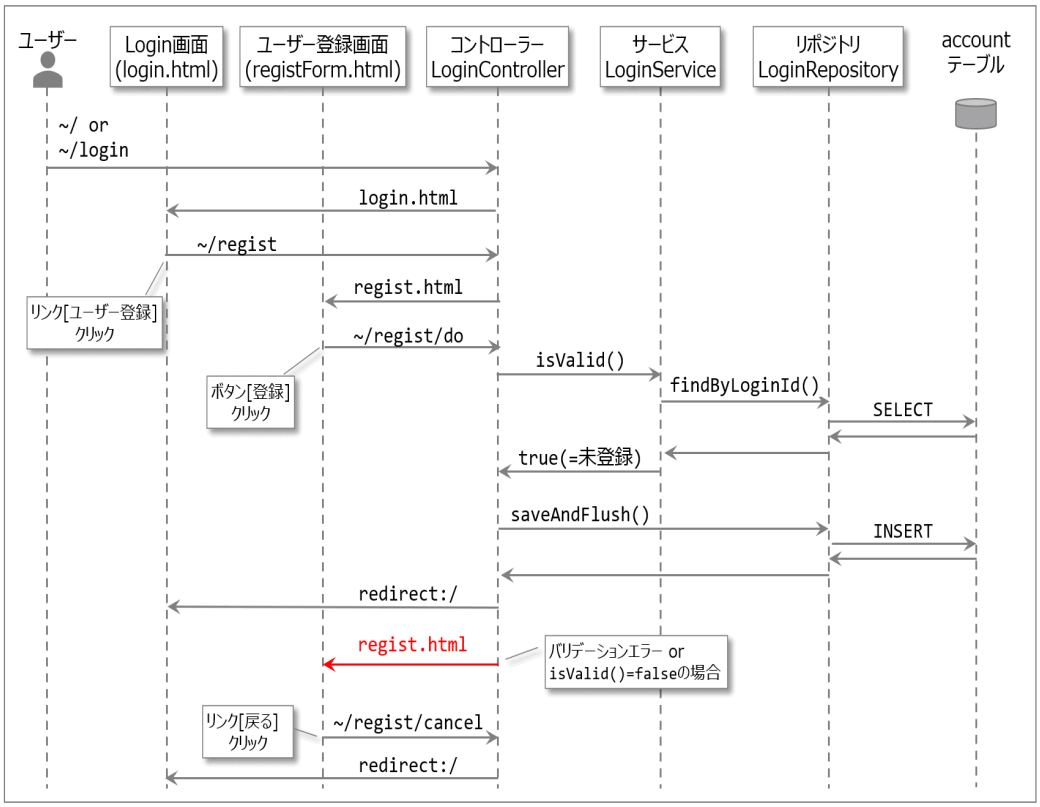
※Length.registData.loginIdなどの{2}、{1}は、表示時min, maxの値に置換されます。

【リスト21-33】src/main/resources/i18n/ValidationMessages_en.properties(追加分)

```
NotBlank.registData.loginId=Not entered.
Length.registData.loginId=Please enter {2} or more and {1} characters or less.
Pattern.registData.loginId=Please enter in half-width alphanumeric characters(A-Za-z0-9)
AlreadyUsed.registData.loginId=This login Id is already in use.

NotBlank.registData.name=Not entered.
```

```
NotBlank.registData.password1=Not entered.
Length.registData.password1=Please enter {2} or more and {1} characters or less.
Pattern.registData.password1=Please enter in half-width alphanumeric characters(A-Za-z0-9)

NotBlank.registData.password2=Not entered.
Length.registData.password2=Please enter {2} or more and {1} characters or less.
Pattern.registData.password2=Please enter in half-width alphanumeric characters(A-Za-z0-9)
Unmatch.registData.password=Passwords do not match
```

最後にハンドラーメソッドをコントローラーLoginControllerへ追加します。関連する処理をシーケンス図で表すと次のようになります。



【図21-5】ユーザー登録に関連するシーケンス

以上をコードにすると以下のようになります。

【リスト21-34】com.example.todolist.controller.LoginController.java(追加分)

```
package com.example.todolist.controller;
   :
import com.example.todolist.form.RegistData;
   :
public class LoginController {
   :
    // ユーザー新規登録 - 画面表示
    @GetMapping("/regist")
    public ModelAndView showRegist(ModelAndView mv) {
        mv.setViewName("registForm");
        mv.addObject("registData", new RegistData());
        return mv;
    }

    // ユーザー新規登録 - 登録
    @PostMapping("/regist/do")
    public String registNewUser(@ModelAttribute @Validated RegistData
registData,
                                BindingResult result,
                                Model model,
                                RedirectAttributes redirectAttributes,
                                Locale locale) {
        // エラーチェック
        boolean isValid = loginService.isValid(registData, result, locale);
        if (!result.hasErrors() && isValid) {
            // エラーなし -> 登録
            Account account = registData.toEntity();
            accountRepository.saveAndFlush(account);

            // 登録完了メッセージをセットしてリダイレクト
            String msg = messageSource.getMessage(
                "msg.i.regist_successful",
```

```
                    new Object[] { account.getName(), account.getLoginId()    },
                    locale);
            redirectAttributes.addFlashAttribute("msg", new OpMsg("I", msg));
            return "redirect:/";

        } else {
            // エラーあり -> エラーメッセージをセット
            String msg
              = messageSource.getMessage("msg.e.input_something_wrong",
null, locale);
            model.addAttribute("msg", new OpMsg("E", msg));
            return "registForm";
        }
    }

    // ユーザー新規登録 - 戻る
    @GetMapping("/regist/cancel")
    public String registCancel() {
        return "redirect:/";
    }
}
```

【リスト21-35】src/main/resources/i18n/OperationMessages_ja.properties(追加分)

```
msg.i.regist_successful=ユーザー:{0}(ID:{1})を登録しました。
```

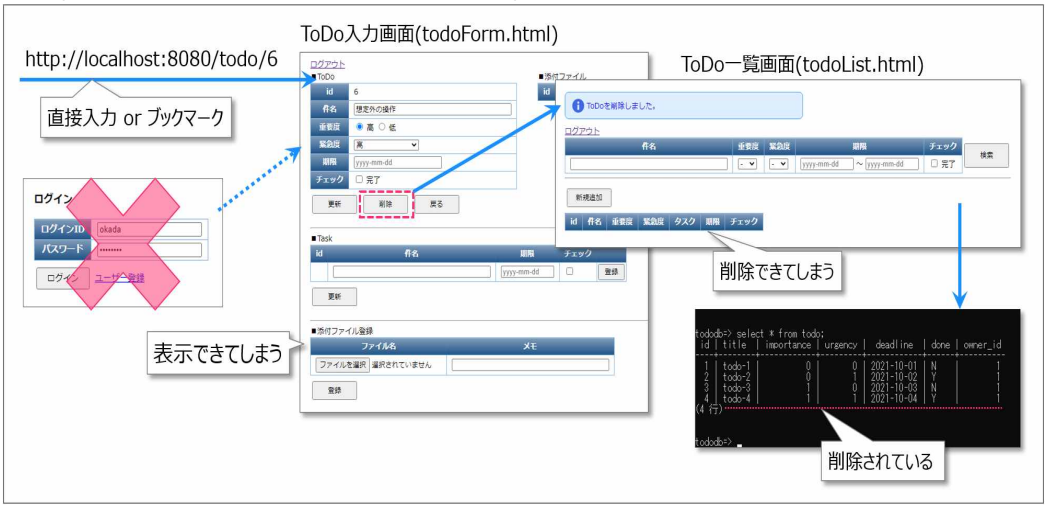
【リスト21-36】src/main/resources/i18n/OperationMessages_en.properties(追加分)

```
msg.i.regist_successful=Registered User:{0}(ID:{1})
```

これでユーザーを登録できるようになります。

22. アクセス制御とエラー処理

前章ではログイン認証を設け、自分のToDoのみ表示するようにしました。しかし、これだけでは不十分です。例えば以下の操作を見てください。



【図22-1】想定外の操作例

ブラウザからhttp://localhost:8080/todo/**6**をアクセスしたとき、もしtodo.id=6のToDoがあれば、入力画面に表示されます。ToDoアプリのidは推測が容易(1～の整数値)なため、総当たりでやれば**他の人のものも閲覧できる**可能性が高いでしょう。さらに[削除]ボタンをクリックすれば、このToDoを削除できてしまいます。

こういったことが起こる原因は2つあります。
(1)ログイン画面を経由しなくてもToDoアプリを操作できる。
(2)操作対象のデータがその人のものかどうかチェックしていない。
本章ではこの問題に対処していきます。

作成するプロジェクトの仕様

プロジェクト名	Todolist15
作成ファイル	com.example.todolist.config.LoginCheckFilter.java src/main/resources/templates/error/404.html src/main/resources/templates/error.html

```
Todolist15
 ├ src/main/java
 │    ├ com.example.todolist
 │    │    └ Todolist15Application.java
 │    ├ com.example.todolist.common
 │    │    ├ OpMsg.java
 │    │    └ Utils.java
 │    ├ com.example.todolist.config(★)
 │    │    └ LoginCheckFilter.java(★)
 │    ├ com.example.todolist.controller
 │    │    ├ DownloadController.java(▲)
 │    │    ├ LoginController.java
 │    │    └ TodoListController.java(▲)
 │    ├ com.example.todolist.dao
 │    │    ├ TodoDao.java
 │    │    └ TodoDaoImpl.java
 │    ├ com.example.todolist.entity
 │    │    ├ Account.java
 │    │    ├ AttachedFile.java
 │    │    ├ Task.java
 │    │    └ Todo.java
 │    ├ com.example.todolist.form
 │    │    ├ AttachedFileData.java
 │    │    ├ LoginData.java
 │    │    ├ RegistData.java
 │    │    ├ TaskData.java
 │    │    ├ TodoData.java
```

```
│   │   └ TodoQuery.java
│   ├ com.example.todolist.repository
│   │   ├ AccountRepository.java
│   │   ├ AttachedFileRepository.java
│   │   ├ TaskRepository.java
│   │   └ TodoRepository.java
│   ├ com.example.todolist.service
│   │   ├ DownloadService.java(▲)
│   │   ├ LoginService.java
│   │   └ TodoService.java
│   └ com.example.todolist.view
│       ├ TodoExcel.java
│       └ TodoPdf.java
└ src/main/resources
    ├ i18n
    │   ├ FixedDisplayStrings.properties
    │   ├ FixedDisplayStrings_en.properties(▲)
    │   ├ FixedDisplayStrings_ja.properties(▲)
    │   ├ OperationMessages_en.properties
    │   ├ OperationMessages_ja.properties
    │   ├ ValidationMessages_en.properties
    │   └ ValidationMessages_ja.properties
    :
    ├ templates
    │   ├ error(★)
    │   │   └ 404.html(★)
    │   ├ error.html(★)
    │   ├ fragments.html
    │   ├ loginForm.html
    │   ├ registForm.html
    │   ├ todoForm.html
    │   └ todoList.html
    └application.properties
```

★：このプロジェクトで追加する

▲：前プロジェクトの内容を一部変更する

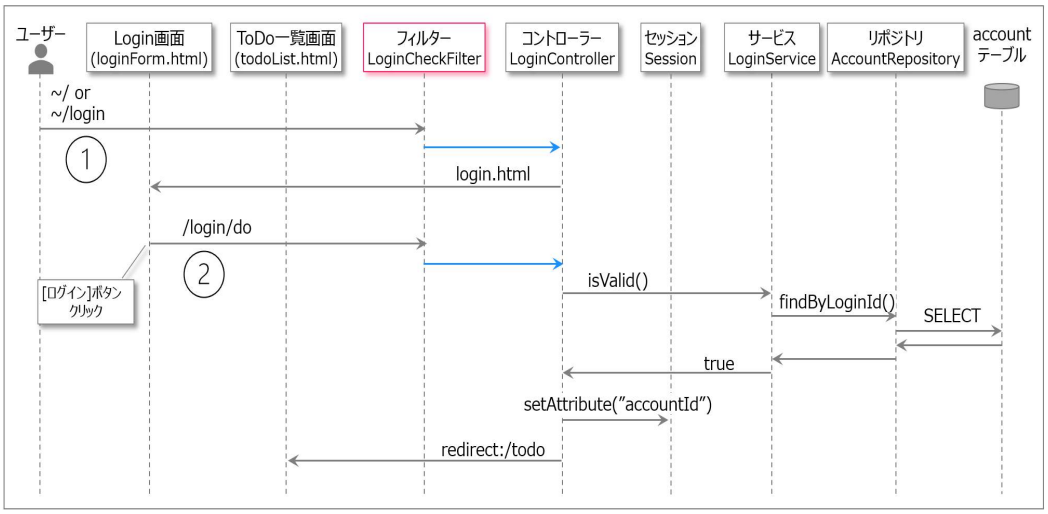
22.1 フィルターによるアクセス制御

まず「ログイン画面を経由しなくてもToDoアプリを操作できる」という問題から考えていきます。この単純な解決方法としては「ハンドラーメソッドの先頭でログインしているかどうかチェックする」というのがあります。以下はその例です。

```
    // ToDo一覧表示

    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                                     @PageableDefault(page = 0, size =
5, sort = "id")
                                     Pageable pageable) {
        // ログインしているかチェックする
        Integer _accountId= (Integer)session.getAttribute("accountId");
        if (_accountId== null) {
            // ログインしていないので、ログイン画面へ遷移する
            mv.setViewName("loginForm");
            mv.addObject("loginData", new LoginData());
            return mv;
        }

        // sessionから前回の検索条件を取得
        :
    }
```

わかりやすいのですが、ハンドラーメソッドが多いと大変です。同じ処理があちらこちらに点在するやり方は、スマートとは言えないでしょう。こういった場合「**フィルター**」という機能を使うと、シンプルにログインチェックを実現できます。以下本節では、ログインしているかどうかで処理を振り分けるLoginCheckFilterを追加します。

下図はLoginCheckFilterがどのような働きをするかシーケンス図で表したものです。ポイントはブラウザからのリクエストを、(コントローラーの前に)フィルターが受け取るところです。

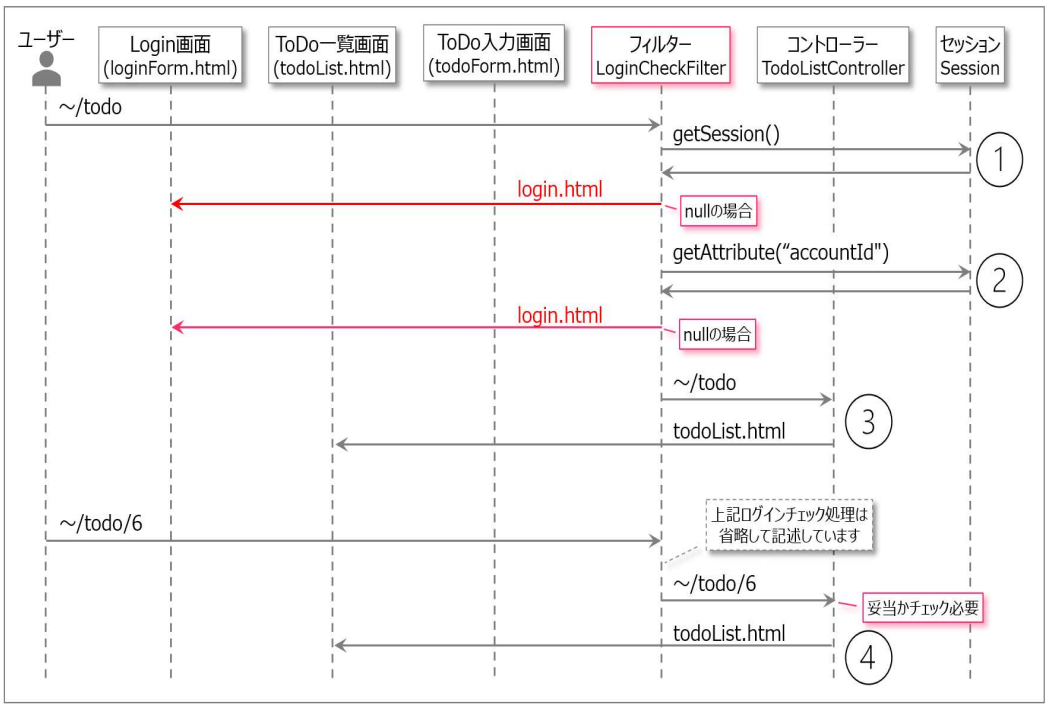


【図22-2】LoginCheckFilter(1)

上図はログイン画面を表示するため http://localhost:8080/ または http://localhost:8080/login をリクエストした場合の例です。上述したように、このリクエストはLoginCheckFilterに行きます(①)。ログイン画面はアクセス制限する必要がないので、LoginCheckFilterはリクエストをコントローラーLoginControllerへ渡します(/, /todoに対するハンドラーメソッドを定義しているため)。これでログイン画面が表示されます。

ログイン画面の[ログイン]ボタンをクリックしたときも同様です(②)。ログイン認証もアクセス制限不要なので、このリクエストもLoginControllerへ渡します。これで認証に成功したら、前章で説明したように、そのユーザーのaccountのidを"accountId"という名称でセッションへ書き込みます。

次はhttp://localhost:8080/**todo**を(ログインせず)直接アクセスした場合です。このアクセス先はToDo一覧画面なので、ユーザーが未ログインならLoginCheckFilterはこれを拒否しなければなりません。



【図22-3】LoginCheckFilter(2)

この場合LoginCheckFilterは、まずセッションを取得します(①)。無ければこのユーザーは未ログインと判断し、ログイン画面を表示します(フィルターがリダイレクトする)。セッションがあればaccountIdを取得します(②)。これも無ければ未ログインとしてログイン画面へリダイレクトします。

セッションにaccountIdが存在すれば、コントローラーTodoListControllerへリクエストを渡し(③)、あとの処理を任せます。ただし/todo/**6**のようにidが指定されている場合は、さらにそのidのデータがログイン者のものか？というチェックが必要です(④)。これについては次節で対処します。

上記の処理を行うLoginCheckFilterは以下のようになっています。

【リスト22-1】com.example.todolist.config.LoginCheckFilter.java

```
package com.example.todolist.config;

import java.io.IOException;
import org.springframework.stereotype.Component;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
```

```
import jakarta.servlet.ServletResponse;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;

@Component // ②
public class LoginCheckFilter implements Filter { // ①
    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {

        // HttpServletRequest/Response は ServletRequest/Response のサブクラス
        HttpServletRequest req = (HttpServletRequest)request;   // ④
        HttpServletResponse res = (HttpServletResponse)response;

        String uri = req.getRequestURI();
        if (uri.startsWith("/todo") || uri.startsWith("/task")) {   // ③
            // sessionが存在するか？
            HttpSession session = req.getSession(false);
            if (session == null) {
                // session無し -> Login画面へリダイレクト   // ⑤
                res.sendRedirect("/login");

            } else {
                // sessionにaccountIdが存在するか(=loginしたか?)
                Integer accountId = (Integer)session.getAttribute("accountId");
                if (accountId == null) {
                    // accountId無し -> Loginしていない -> Login画面へリダイレクト   // ⑥
                    res.sendRedirect("/login");

                } else {
```

```
                    //    Loginしている -> コントローラーへリクエストを渡す
                    chain.doFilter(request, response);// ⑦
                }
            }

        } else {
            // check対象外 -> コントローラーへリクエストを渡す
            chain.doFilter(request, response);// ⑧
        }
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {}

    @Override
    public void destroy() {}
}
```

フィルターはjakarta.servlet.Filterインターフェースを実装したクラスとします(①)。このインターフェースは3つの抽象メソッドを持っていますが、ここでは**doFilter()**のみ実装します。init(フィルタ初期化処理)、destroy(フィルタ終了処理)は使わないので、オーバーライドのみ(メソッド本体は無し)とします。

またフィルタークラスには**@Componentアノテーション**を付与します(②)。これによりToDoアプリ起動時、LoginCheckFilterが【図22-2】【図22-3】のように配置されます。

以下は実装するdoFilter()のメソッドシグネチャです。このメソッドで処理を振り分けます。

```
void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
```

request
クライアントからのリクエストを保持している → ユーザーがリクエストしたURLパスはここから取得する

response
クライアントへのレスポンスを保持している → 要求されたURLパスと異なる画面へリダイレクトするのに使用する

chain
次のフィルター、またはリソースを保持している → コントローラーへリクエストを渡すのに使用する

LoginCheckFilterではURLパスが"/todo","/task"で始まるものだけをチェックします(③)。URLパスはdoFilter()の引数request(ServletRequest型)の内部に保持していますが、これを取得するには一度サブクラスのHttpServletRequest型へダウンキャストとします(④)。

```
    // HttpServletRequest/Response は ServletRequest/Response のサブクラス
    HttpServletRequest req = (HttpServletRequest)request;    // ④
    HttpServletResponse res = (HttpServletResponse)response;

    String uri = req.getRequestURI();
    if (uri.startsWith("/todo") || uri.startsWith("/task")) {    // ③
```

```
    :
```

URLが合致したらgetSession(false)でセッションを取得します。null、すなわちセッションが存在しなければ"/login"へリダイレクトしてloginForm.htmlを表示します(⑤)。これは【図22-3】①部分の処理です。セッションにaccountIdが存在しない場合も"/login"へリダイレクトします(⑥)(【図22-3】②)。

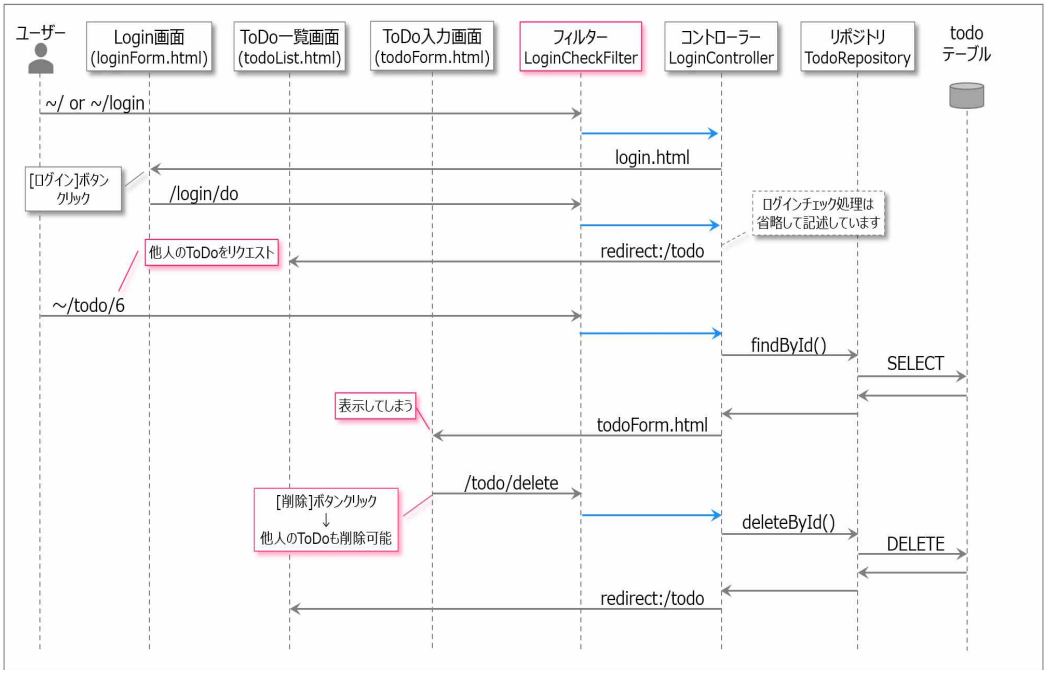
accountIdが取得できればログイン済みと判断し、コントローラーに制御を渡します(⑦)。これでURLパスに一致するメソッドハンドラーが呼び出されます(【図22-3】③)。

一方、"/todo", "/task"以外なら、そのままコントローラーに制御を渡します(⑧)。(【図22-2】①②)

これで未ログインユーザーが"/todo", "/task"で始まるURLパスへアクセスすると、ログイン画面が表示されるようになります。しかし【図22-3】④の"/todo/6"というURLパスは、ログインしていても、そのユーザーのものかチェックが必要です。つまり本人のものである場合のみ、表示する制御が必要です。これは次節で説明します。

22.2 プログラムによるアクセス制御

ここで何が問題になるか以下のシーケンス図で整理します。



【図22-4】他の人のToDoを削除できるパターン

ユーザーがログインに成功しToDo一覧画面へ遷移したとします。そしてhttp://localhost:8080/todo/6を入力したとします。このときtodo.id=6のデータが存在すれば、誰のものであっても入力画面に表示されてしまいます。ここで[削除]ボタンをクリックすれば、このToDoは削除されます。つまりログイン認証だけでは、他の人のToDoを操作できてしまう可能性があるわけです。

解決方法はいくつかありますが、ToDoアプリではハンドラーメソッドに、idが操作者のものかチェックする処理を追加することにします。対象となるハンドラーメソッドは以下の通りです。

・URLパスが"/todo", "/task"で始まる、かつ
・GETリクエスト(URLを直接入力した場合のため)、かつ
・URLパス変数、クエリー文字列にidを持つ

これに該当するのは次の処理です。

(1)ToDo表示処理：todoById()
(2)Task削除処理：deleteTask()

(3)添付ファイル削除処理：deleteAttachedFile()
(4)添付ファイルのダウンロード処理：downloadAttachedFile()

(1)ToDo表示処理：todoById()

以下はここまでのtodoById()です。URLパス変数で指定されたidのToDoを表示します。

```
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int id,
                                 ModelAndView mv) {
        mv.setViewName("todoForm");
        // ToDo取得
        Todo todo = todoRepository.findById(id).get();
        // 添付ファイル取得
        List<AttachedFile> attachedFiles =
attachedFileRepository.findByTodoIdOrderById(id);
        // 表示用データ作成
        mv.addObject("todoData", new TodoData(todo, attachedFiles));
        session.setAttribute("mode", "update");
        return mv;
    }
```

これに操作者のToDoかチェックする処理を加えると以下のようになります。

【リスト22-2】

com.example.todolist.controller.TodoListController#todoById()

```
    :
import java.util.Optional;
    :
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int id,
                                 ModelAndView mv,
                                 RedirectAttributes redirectAttributes, // ⑤
                                 Locale locale) { // ⑥
        // ToDo取得
        Optional<Todo> someTodo = todoRepository.findById(id); // ①
        someTodo
```

```
                .ifPresentOrElse(todo -> {
                    // todoは存在する
                    Integer accountId = (Integer)session.getAttribute("accountId");
// ②
                    // 操作者のToDoか?
                    if (todo.getOwnerId().equals(accountId)) {
                        mv.setViewName("todoForm");
                        // 添付ファイル取得
                        List<AttachedFile> attachedFiles
                            = attachedFileRepository.findByTodoIdOrderById(id);
                        // 表示用データ作成
                        mv.addObject("todoData", new TodoData(todo,
attachedFiles));
                        session.setAttribute("mode", "update");
                    } else {
                        // 操作者のものでない
                        operationError(mv, redirectAttributes, locale); // ③
                    }
                }, () -> {
                    // todoが存在しない
                    operationError(mv, redirectAttributes, locale); // ④
                });

        return mv;
    }
    :
    // errorへリダイレクトする準備
    private void operationError(ModelAndView mv, RedirectAttributes
redirectAttributes,
                                Locale locale) {
        session.invalidate();
        String msg = messageSource.getMessage("msg.e.operation_error", null,
locale);
        redirectAttributes.addFlashAttribute("msg", new OpMsg("E", msg));
```

```
            mv.setViewName("redirect:/error");
        }
```

まずURLパス変数で指定されたidでTodoを検索します(①)。このとき変更前とは違いget()を実行していません。よってOptional<Todo>型の結果が返され、someTodoは以下どちらかの状態となります。

・someTodo.isPresent() == true　該当するTodoを取得できた
・someTodo.isPresent() == false　idに該当するTodoは存在しない

そこで次のようなコードにしがちですが、これではOptionalのメリットが活かせません。

```
    Optional<Todo> someTodo = todoRepository.findById(id);
    if (someTodo.isPresent()){
        Todo todo = someTodo .get();
        // todoの所有者をチェック
            :
        // todoの表示処理
            :
    } else {
        //   todoが存在しないのでエラー処理
            :
    }
```

ここはラムダ式を使って以下のようにした方が良いでしょう。

```
    Optional<Todo> someTodo = todoRepository.findById(id);
    someTodo.ifPresentOrElse(todo -> {
        // todoの所有者をチェック
            :
        // todoの表示処理
            :
    }, () -> {
        //   todoが存在しないのでエラー処理
            :
    });
```

コメントを省くと以下のような形になっています。

```
someTodo.ifPresentOrElse(todo -> {}, ()->{})
```

ifPresentOrElse()はsomeTodoがnullでなければ最初の{}内、nullなら2つ目の{}内を実行します。todoにはsomeTodo.get()の結果がセットされており、最初の{}内で使えます。こうすればnullかどうかで処理を分岐できます。

最初の{}内ではセッションからacountIdを取得し、todo.ownerIdと比較します(②)。一致すれば操作者のToDoなので表示し、それ以外はエラー画面へリダイレクトします(③)。該当するTodoが無い場合も、エラー画面へリダイレクトします(④)。

なおハンドラーメソッドの引数にはリダイレクト先へメッセージを渡すためのRedirectAttributes、メッセージの言語を決めるLocaleを追加しておきます(⑤⑥)。

このリダイレクト先のエラー画面は以下のようになっています。

【リスト22-3】src/main/resources/templates/error.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <h1 th:text="#{Somthing.Wrong1}"></h1>
  <h2 th:text="#{Somthing.Wrong2}"></h2>
  <a href="/login" th:text="#{Somthing.Wrong3}"></a>
</body>
</html>
```

【リスト22-4】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
Somthing.Wrong1=操作に何らかの誤りがあるようです。
Somthing.Wrong2=もう一度、↓こちらから操作しなおしてください。
Somthing.Wrong3=ログイン画面
```

【リスト22-5】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
Somthing.Wrong1=Sorry, there seems to be an error in your operation.
Somthing.Wrong2=Please try again from here.
Somthing.Wrong3=Login page
```

これで以下のようなエラー画面を出力できます。

(2)タスク削除処理：deleteTask()

【リスト22-6】
com.example.todolist.controller.TodoListController#deleteTask()

```
    // Task削除処理
    @GetMapping("/task/delete")
    public ModelAndView deleteTask(@RequestParam(name = "task_id") int
taskId,
                                   @RequestParam(name = "todo_id") int
todoId,
                                   ModelAndView mv, // ②
                                   RedirectAttributes redirectAttributes,
Locale locale) {
        // ToDo取得
        Optional<Todo> someTodo = todoRepository.findById(todoId);
        someTodo
            .ifPresentOrElse(todo -> {
                // todoは存在する
                Integer accountId = (Integer)session.getAttribute("accountId");
                // 操作者のToDoか?
                if (todo.getOwnerId().equals(accountId)) {
                    // Taskを削除
                    taskRepository.deleteById(taskId);

                    // 削除完了メッセージをセットしてリダイレクト
                    String msg =
messageSource.getMessage("msg.i.task_deleted", null, locale);
                    redirectAttributes.addFlashAttribute("msg", new
OpMsg("I", msg));
                    mv.setViewName("redirect:/todo/" + todoId); // ①

                } else {
                    // 操作者のものでない
```

```
                    operationError(mv, redirectAttributes, locale);
                }
            }, () -> {
                // todoが存在しない
                operationError(mv, redirectAttributes, locale);
            });

        return mv;

    }
```

ifPresentOrElse()で制御するのは同じです。ただし入力画面へのリダイレクト方法をoperationError()に合わせてModelAndViewを使うようにします(①)。

```
return "redirect:/todo/" + todoId;
↓
mv.setViewName("redirect:/todo/" + todoId); // ①
return mv;
```

このためにメソッドにModelAndView型の引数を追加し、戻り値もModelAndView型へ変更します(②)。

(3)添付ファイル削除処理：deleteAttachedFile()

タスク削除の場合と同様です。

【リスト22-7】

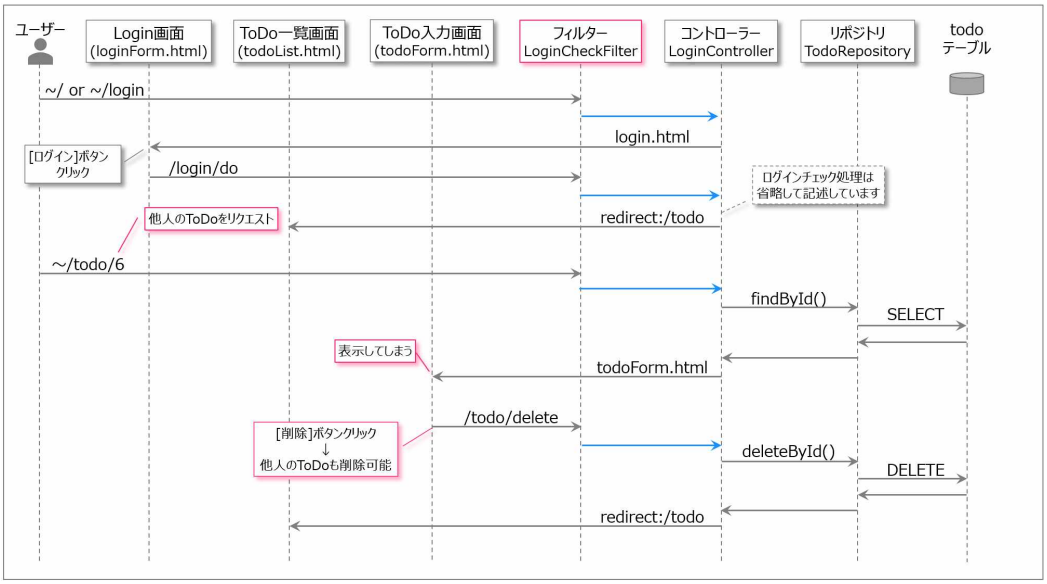
com.example.todolist.controller.TodoListController#deleteAttachedFile()

```
    // 添付ファイルを削除する
    @GetMapping("/todo/af/delete")
    public ModelAndView deleteAttachedFile(@RequestParam(name = "af_id")
int afId,
                                           @RequestParam(name =
"todo_id") int todoId,
                                           ModelAndView mv,
                                           RedirectAttributes
redirectAttributes,
                                           Locale locale) {
        // ToDo取得
        Optional<Todo> someTodo = todoRepository.findById(todoId);
        someTodo
            .ifPresentOrElse(todo -> {
                // todoは存在する
                Integer accountId = (Integer)session.getAttribute("accountId");
                // 操作者のToDoか?
                if (todo.getOwnerId().equals(accountId)) {
                    // 添付ファイルを削除
                    todoService.deleteAttachedFile(afId);
                    // attached_fileテーブルから削除
                    attachedFileRepository.deleteById(afId);

                    // 削除完了メッセージをセットしてリダイレクト
                    String msg = messageSource
                        .getMessage("msg.i.attachedfile_deleted", null, locale);
                    redirectAttributes.addFlashAttribute("msg", new
OpMsg("I", msg));
                    mv.setViewName("redirect:/todo/" + todoId);
```

```
                } else {
                    // 操作者のものでない
                    operationError(mv, redirectAttributes, locale);
                }
            }, () -> {
                // todoが存在しない
                operationError(mv, redirectAttributes, locale);
            });

        return mv;
    }
```

(4)添付ファイルのダウンロード処理：downloadAttachedFile()

URLパス変数で受け取るのは「添付ファイルのid」です。そのため下図のようにテーブルを 2 回アクセスします。



【図22-5】ファイルダウンロード時のチェック処理

例えば上図のようaccountId=1のユーザーが、attached_file.id=11の添付ファイルをリクエストしたとします。コントローラーは11をキーにattached_fileを検索し、さらにtodo_id(=2)をキーにtodoを検索します。これで得られるowner_idがaccountIdと同じなら、このユーザーの添付ファイルなので、ダウンロードします。一方これ以外ならエラー画面を表示します。

> attached_fileテーブルとtodoテーブルを結合させた**View(ビュー)**を作成すれば、テーブルアクセスを 1 回にできます。ビューを使ったアクセスは発展編で解説していますので興味がある方は御覧ください。

これをコードにすると以下のようになります。ifPresentOrElse()がネストしていますが、シーケンス図と照らし合わせれば流れが追えると思います。

【リスト22-8】
com.example.todolist.controller.DownloadController#downloadAttachedFile()

```
        :
```

```
import java.util.Locale;
import java.util.Optional;
import com.example.todolist.entity.AttachedFile;
import com.example.todolist.entity.Todo;
import com.example.todolist.repository.AttachedFileRepository;
import com.example.todolist.repository.TodoRepository;
import jakarta.servlet.http.HttpSession;
    :
@RestController
@RequiredArgsConstructor
public class DownloadController {
    private final TodoRepository todoRepository;
    private final AttachedFileRepository attachedFileRepository;
    private final DownloadService downloadService;
    private final HttpSession session;

    // 添付ファイルのダウンロード処理
    @GetMapping("/todo/af/download/{afId}")
    public void downloadAttachedFile(@PathVariable(name = "afId") int afId,
                                     HttpServletResponse response,
                                     Locale locale) {

        // 添付ファイル情報を取得
        Optional<AttachedFile> someAf =
attachedFileRepository.findById(afId);
        someAf.ifPresentOrElse(af -> {
            // 添付ファイルが存在する -> 添付先Todoの情報を取得
            Optional<Todo> someTodo =
todoRepository.findById(af.getTodoId());
            someTodo
                .ifPresentOrElse(todo -> {
                    // ToDoが存在する
                    // このセッションのaccountId取得
```

```
                        Integer accountId =
(Integer)session.getAttribute("accountId");

                        // 操作者のToDoか？
                        if (todo.getOwnerId().equals(accountId)) {
                            // 本人のものなのでdownloadする
                            downloadService.downloadAttachedFile(afId,
response);

                        } else {
                            // 操作者のToDoでない

downloadService.invalidDownloadRequest(session,response, locale);
                        }
                    }, () -> {
                        // ToDoが存在しない

downloadService.invalidDownloadRequest(session,response, locale);
                    });
            }, () -> {
                // 添付ファイルが存在しない
                downloadService.invalidDownloadRequest(session,response,
locale);
            });
        }

}
```

エラー画面の表示方法は(1)～(3)と異なります。DownloadControllerには@RestControllerを付与しているため、文字列操作でエラー画面のHTMLを組み立て、それをブラウザに送ります。これが上記エラー時に実行しているDownloadService#invalidDownloadRequest()です。

【リスト22-9】
com.example.todolist.service.DownloadService#invalidDownloadRequest()

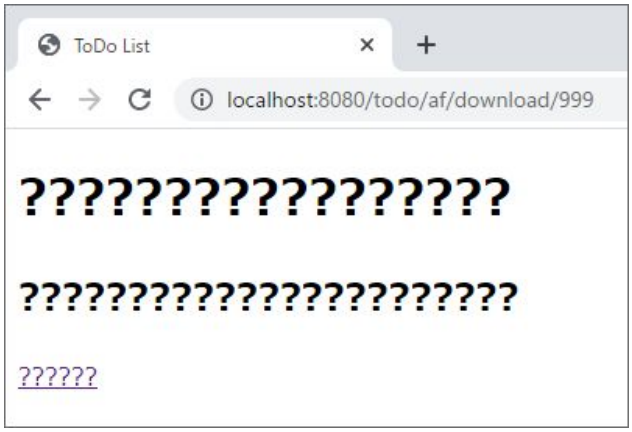
```
package com.example.todolist.service;
:
import java.io.PrintWriter;
import java.util.Locale;
import org.springframework.context.MessageSource;
import jakarta.servlet.http.HttpSession;
:
public class DownloadService {
    :
    private final MessageSource messageSource; // ①
    :
    // 不正なダウンロードリクエストの場合
    public void invalidDownloadRequest(HttpSession session,
                                       HttpServletResponse response,
                                       Locale locale) {
        // セッションを無効化
        session.invalidate();

        // メッセージを取得
        String mes1 = messageSource.getMessage("Somthing.Wrong1", null,
locale); // ②
        String mes2 = messageSource.getMessage("Somthing.Wrong2", null,
locale);
        String mes3 = messageSource.getMessage("Somthing.Wrong3", null,
locale);

        // エラー画面を出力する
        try {
            response.setContentType("text/html;charset=UTF-8"); // ③
            PrintWriter out = response.getWriter(); // ④
            out.println("<!DOCTYPE html>"); // ⑤
```

```
                out.println("<html>");
                out.println("<head>");
                out.println("<meta charset='UTF-8'>");
                out.println("<title>ToDo List</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("    <h1>" + mes1 + "</h1>");
                out.println("    <h2>" + mes2 + "</h2>");
                out.println("    <a href='/login'>" + mes3 + "</a>");
                out.println("</body>");
                out.println("</html>");

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

表示するメッセージは【リスト22-3】error.htmlと同じものです(①②)。処理はまずこれから送信するのが「UTF-8でエンコードされたhtml形式のテキスト」であることを示すContent-Typeタイプをセットします(③)。そしてブラウザへの出力先をgetWriter()で取得します(④)。あとはこのライターへprintln()を実行すれば、その内容がブラウザへ送信されます(⑤)。

> 18章のDownloadService#downloadAttachedFile()では、ファイルをダウンロードするのに出力ストリームを取得していました(getOutputStream())。これはダウンロードデータをバイト単位で出力するためです。ここでは文字(英数字は1バイト、漢字は2バイト)を出力するのでライターを使います。

> ②のsetContentType()は、ライターを取得する前、つまり③のgetWriter()の前に実行します。③getWriter()→②setContentType()にするとエラー画面が文字化けします。



これは③→②ではsetContentType()が効かず、"Content-Type: text/html;charset=ISO-8859-1"がデフォルトとして使われるためです(ISO-8859-1は、簡単に言えば半角英数字なので、全角文字が？になる)。

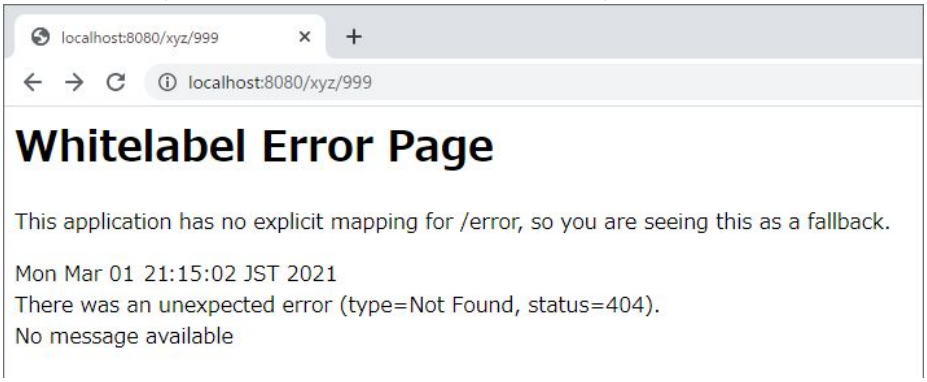
22.3 エラー画面の自動表示

ここまで操作している中で、存在しないURLパスをアクセスしたときもエラー画面(error.html)が表示されることに気付いた方がいるかもしれません。以下はhttp://localhost:8080/xyz/999という存在しないURLを入力した場合ですが、エラー画面が表示されます。しかしプログラムではこのような制御をしていません。

【図22-6】自動的に表示されたエラー画面

これはSpring Bootに「エラーが発生したとき、error.htmlが存在すれば、それを表示する」という処理が組み込まれているためです(もう少し正確に言うと「表示されるのは/errorというビューであり、Thymeleafを使っていればsrc/main/resources/templates/error.htmlがエラーページとして扱われる」ということです)。

実際同じ操作を前章のTodolist14で実行すると、デフォルトの「Whitelabel Error Page」という画面が出力されます。これはerror.htmlが無いからです。



【図22-7】Whitelabel Error Pageの例

このようにerror.htmlは、独自のものにできます。そのとき以下のようなエラー情報も利用できます。

【表22-1】error.htmlで取得可能なエラー情報

取得可能な項目	内容
timestamp	エラー発生時刻
path	例外が発生したpath
status	HTTP Status
error	エラー理由
message	例外メッセージ
exception	root exceptionのクラス名(設定されていれば)
errors	入力チェックのエラー情報
trace	例外のstack trace

これらのエラー情報をerror.htmlに追加してみます。

【リスト22-10】src/main/resources/templates/error.html

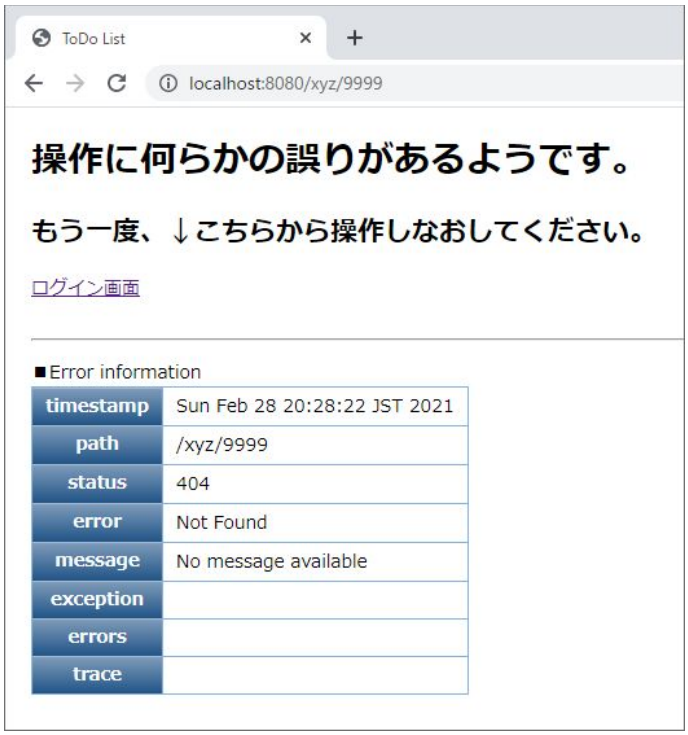
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
<body>
  <h1 th:text="#{Somthing.Wrong1}"></h1>
  <h2 th:text="#{Somthing.Wrong2}"></h2>
  <a href="/login" th:text="#{Somthing.Wrong3}"></a>
  <!-- エラー情報　ここから --->
  <div th:unless="${error}=='None'">
    <hr style="margin-top: 2em; margin-bottom: 1em;">
    <span>■Error information</span>
    <table>
      <tr>
        <th>timestamp</th>
        <td th:text="${timestamp}"></td>
      </tr>
```

```
        <tr>
          <th>path</th>
          <td th:text="${path}"></td>
        </tr>
        <tr>
          <th>status</th>
          <td th:text="${status}"></td>
        </tr>
        <tr>
          <th>error</th>
          <td th:text="${error}"></td>
        </tr>
        <tr>
          <th>message</th>
          <td th:text="${message}"></td>
        </tr>
        <tr>
          <th>exception</th>
          <td th:text="${exception}"></td>
        </tr>
        <tr>
          <th>errors</th>
          <td th:text="${errors}"></td>
        </tr>
        <tr>
          <th>trace</th>
          <td th:text="${trace}"></td>
        </tr>
      </table>
    </div>
    <!-- エラー情報　ここまで --->
</body>
</html>
```

これでブラウザのURL欄からhttp://localhost:8080/todo/create/formをリクエストすると、以下のようなエラー画面が表示されます。

【図22-8】エラー情報を追加したerror.htmlの表示例

一方、本章最初のhttp://localhost:8080/xyz/9999のように存在しないURLなら以下のようになります。これは(いわゆる)404エラーで原因が明らかです。一般的には専用のエラーページを用意しているシステムが多いようです。



【図22-9】404エラーが発生した場合のerror.html

Spring Bootでは404エラーが起きたとき、src/main/resources/templates/**error**/404.htmlというビューがあれば、それを表示します。同様にスタータスコード4xxの画面は4xx.htmlで定義しておくことができます。以下は、error.htmlを簡略化して404エラーに特化した404.htmlの例です。

【リスト22-11】src/main/resources/templates/error/404.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>ToDo List</title>
<link th:href="@{/css/style.css}" rel="stylesheet" type="text/css">
</head>
</body>
<h1 th:text="#{Somthing.Wrong404}"></h1>
<h2 th:text="#{Somthing.Wrong2}"></h2>
<a href="/login" th:text="#{Somthing.Wrong3}"></a>
</body>
```

```
</html>
```

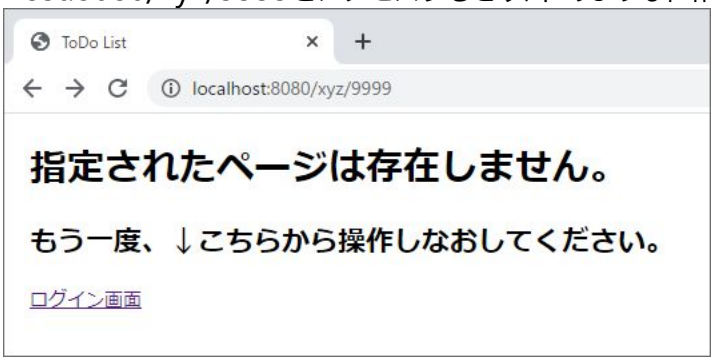
【リスト22-12】src/main/resources/i18n/FixedDisplayStrings_ja.properties(追加分)

```
Somthing.Wrong404=指定されたページは存在しません。
```

【リスト22-13】src/main/resources/i18n/FixedDisplayStrings_en.properties(追加分)

```
Somthing.Wrong404=The specified page does not exist.
```

これでhttp://localhost:8080/xyz/9999をアクセスすると以下のような画面が表示されます。



【図22-10】404.html表示例

22.4 終わりに向かって

以上、ログイン認証、アクセス制限などを解説してきましたが、考慮すべき事項はまだ数多くあります。例えば、ユーザーがブラウザの[←][→]ボタンを操作して画面遷移した場合への対処。またCSRF(cross-site request forgeries;クロスサイトリクエストフォージェリ)対策も欠かせません。ぜひ自分で調べてみてください。

補足：前書・演習課題の実装例

本章では前書の最後に出題した演習課題の実装例について解説します(プロジェクト名：Todolist7)。

さらに本編の解説量を抑えるため、ここで若干コードを追加します(プロジェクト名：Todolist7x)。

→実際のコードはサポートサイトからダウンロードできます。

本書12章のTodolist8は、このTodolist7xをコピーして作成してください。

演習課題の実装例(Todolist7)

①更新の場合、完了日が過去でもエラーにしない

完了日が過ぎたToDoを更新しようとすると「期限を設定するときは今日以降にしてください」というエラーメッセージが表示されます。そのため期限オーバーのToDoは件名、重要度、緊急度だけ直すことができません。そこで更新の場合、完了日はyyyy-mm-ddの形式チェックだけするよう変更してください。

■解説

このエラーメッセージは、ToDo入力画面の内容をチェックするTodoServiceのisValid()で出力しています。このメソッドはコントローラーTodoListControllerのcreateTodo(新規登録)、updateTodo(更新)から呼ばれています。

方法はいくつかありますが、もっとも簡単なのは「createTodo()、updateTodo()のどちらから呼び出されたのか判別できる引数(フラグ)を追加する」というものです。

```
public boolean isValid(TodoData todoData, BindingResult result) {
                              ↓
public boolean isValid(TodoData todoData, BindingResult result, boolean isCreate) {
```

isCreateには新規追加(=createTodo())ならtrue、更新(=updateTodo())ならfalseをセットします。こうすればisCreate=falseの場合、isValid()で期限の過去日付チェックは不要です。実際のロジックは、以下のようにif文を追加すればOKです。

```
//   yyyy-mm-dd形式チェック、過去日付チェック
String deadline = todoData.getDeadline();
if (!deadline.equals("")) {
        LocalDate tody = LocalDate.now();
        LocalDate deadlineDate = null;
        try {
                deadlineDate = LocalDate.parse(deadline);
                // 過去日付チェックは新規登録の場合のみ
                if (isCreate) {    // ← 追加したif文
                        if (deadlineDate.isBefore(tody)) {
                                FieldError fieldError = new FieldError(
                                        result.getObjectName(),
                                        "deadline",
```

```
                        "期限を設定するときは今日以降にしてください");
                result.addError(fieldError);
                ans = false;
            }
        }
    } catch (DateTimeException e) {
        FieldError fieldError = new FieldError(
            result.getObjectName(),
            "deadline",
            "期限を設定するときはyyyy-mm-dd形式で入力してください");
        result.addError(fieldError);
        ans = false;
    }
}
```

そしてTodoListController側は上記に合わせて２か所修正します。
まず新規登録の場合は、isValid()でtrueを渡すようにします。

```
@PostMapping("/todo/create/do")
public String createTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result,
                         Model model) {
    // エラーチェック
    boolean isValid = todoService.isValid(todoData, result, true);
                         :
```

また更新の場合はfalseです。

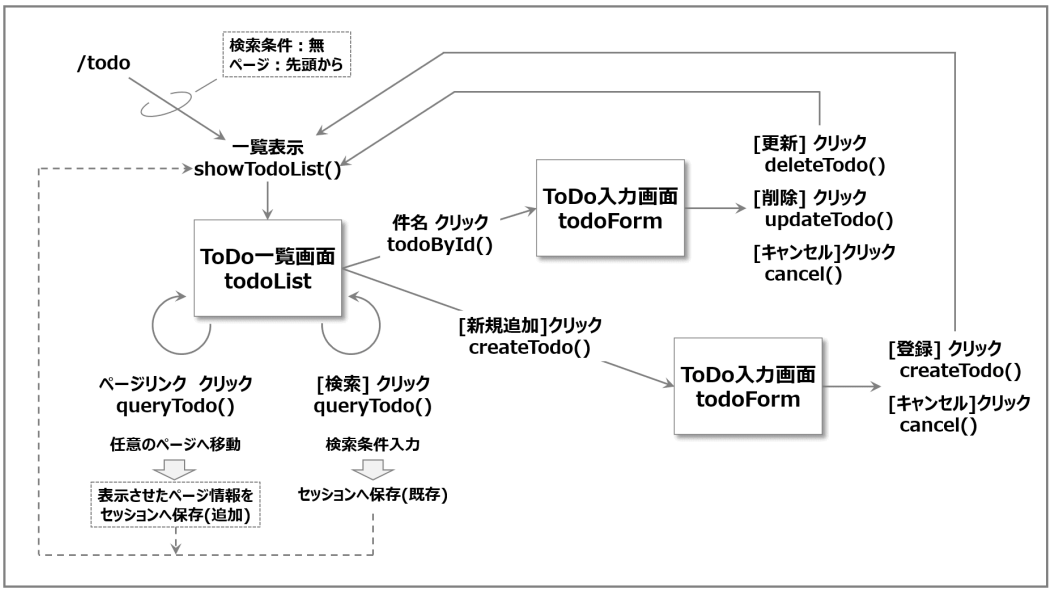
```
@PostMapping("/todo/update")
public String updateTodo(@ModelAttribute @Validated TodoData todoData,
                         BindingResult result,
                         Model model) {
    // エラーチェック
    boolean isValid = todoService.isValid(todoData, result, false);
                         :
```

②入力画面から元のページへ戻る

入力画面で[登録][更新][削除][キャンセル]ボタンを押下すると、1ページ目に戻ります。これを入力画面へ遷移する前に表示していたページへ戻るよう変更してください。

■解説

こういった課題は、まず処理の流れを視覚化すると良いでしょう。11章Todolist6を図解すると以下のようなイメージです。画面を四角形で表し、ユーザーの操作とそれを処理するメソッドは矢印付きの実線としています。



これを見るとToDo入力画面で[登録][更新][削除][キャンセル]をクリックした後、ToDo一覧画面を表示するのはshowTodoList()ということがわかります。これは各メソッドからshowTodoList()へリダイレクトしていることからも明らかです。

ToDo入力画面へ遷移する前のToDo一覧を再表示するには、遷移前に表示したときの①検索条件、②ページ位置情報が必要です。これらの情報は、画面間で共有するのでセッションに保存するのが良いでしょう。

まず①の検索条件はToDo一覧画面の検索条件入力フォームへ入力されます。これをTodolist6では[検索]ボタンがクリックされたときtodoQueryという名前でセッションに格納しています。これはそのまま使えます。

②のページ情報、つまり今何ページ目を表示しているかは、ページリンクがクリックされたとき決まります。これはqueryTodo()の引数pageableそのものです。これもセッションへ格納します。名前はprevious(前の) + pageable を短くした、"prevPageable"とします。コードで表すと次のようになります。

■com.example.todolist.controller.TodoListController#queryTodo()

```
    // ページリンク押下時
    @GetMapping("/todo/query")
    public ModelAndView queryTodo(
                            @PageableDefault(page=0, size=5, sort="id") Pageable
pageable,
                            ModelAndView mv) {
        // 現在のページ位置を保存
        session.setAttribute("prevPageable", pageable);       // ←追加
            :
        mv.setViewName("todoList");
            :
        return mv;
    }
```

この①②は上の図で言うと左側の点線矢印に当たります。

次にshowTodoList()の処理を考えます。このメソッドが呼び出されるパターンは 2 つあります。
【パターン1】ブラウザからhttp://localhost:8080/todo がアクセスされたとき
【パターン2】ToDo入力画面から遷移してきたとき

【パターン1】の場合、セッションには①検索条件、②ページ情報とも格納されていません。そのためshowTodoList()は、検索条件なし、先頭ページから表示します。そしてその内容(①②)をセッションへ保存します。
一方【パターン2】は、遷移前のToDo一覧画面を表示したときに、①②ともセッションへ格納されています。これを使い一覧を作成します。

これをコードにすると以下のようになります。

■com.example.todolist.controller.TodoListController.showTodoList()

```
    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
```

```
                        @PageableDefault(page = 0, size = 5, sort = "id") Pageable
pageable) {
        // sessionから前回の検索条件を取得
        TodoQuery todoQuery = (TodoQuery)session.getAttribute("todoQuery");
        if (todoQuery == null) {
            // なければ初期値を使う
            todoQuery = new TodoQuery();
            // セッションへ保存
            session.setAttribute("todoQuery", todoQuery);
        }

        // sessionから前回のpageableを取得
        Pageable prevPageable = (Pageable)session.getAttribute("prevPageable");
        if (prevPageable == null) {
            // なければ@PageableDefaultを使う
            prevPageable = pageable;
            // セッションへ保存
            session.setAttribute("prevPageable", prevPageable);
        }

        mv.setViewName("todoList");

        Page<Todo> todoPage = todoDaoImpl.findByCriteria(todoQuery,
prevPageable);
        mv.addObject("todoQuery", todoQuery);
        mv.addObject("todoPage", todoPage);
        mv.addObject("todoList", todoPage.getContent());

        return mv;
    }
```

一覧の検索はTodoRepository#findAll()から11章で作成したTodoDaoImpl#findByCriteria()へ変更します。これで前回の①検索条件、②ページ情報を使って検索できるようになります。

③ページリンクの範囲を制限する

ページリンクを表示中ページの前後2ページ分だけにしてください。

現状)たとえば、検索結果が12ページあったら、1～12がページリンクになる。

変更後)ページリンクを以下のようにする。

■解説

現状は#numbers.sequence()で全ページ分のリンクを生成しています。下記のように先頭は0で、最終ページは総ページ番号-1(=todoPage.getTotalPages() - 1)です。

```
<li th:each="i : ${#numbers.sequence(0, todoPage.getTotalPages() - 1)}">
```

これを「表示中ページの前後2ページに限定する」ということは、「表示中ページのページ番号-2 ～ 表示中ページのページ番号+2」へ変更すればいいわけです。ただし「0～総ページ番号-1」の範囲に収まるよう調整する必要があります。

【開始ページ番号】

「表示中ページのページ番号-2」が0未満なら0とします。if文で書くと以下のようなイメージです。

```
if (表示中ページのページ番号 - 2 < 0) {
    開始ページ番号 = 0;
} else {
    開始ページ番号 = 表示中ページのページ番号 - 2;
}
```

【終了ページ番号】

同様に「表示中ページのページ番号+2」が総ページ番号-1を超えるかどうかで処理を分岐させます。

```
if (表示中ページのページ番号 + 2 < 総ページ番号 - 1) {
    終了ページ番号 = 総ページ番号 - 1;
} else {
```

```
        終了ページ番号 = 表示中ページのページ番号 + 2;
}
```

これをtodoList.htmlで使っているオブジェクトで置き換えると以下のようになります。

【開始ページ番号】

```
if (todoPage.getNumber()-2 < 0) {
        開始ページ番号 = 0;
} else {
        開始ページ番号 = todoPage.getNumber()-2;
}
```

【終了ページ番号】

```
if (todoPage.getNumber()+2 > todoPage.getTotalPages()-1) {
        終了ページ番号 = todoPage.getTotalPages()-1;
} else {
        終了ページ番号 = todoPage.getNumber()+2;
}
```

これを以下のようにすればいいのですが、本書ではThymeleafの変数機能について説明していません。

```
<li th:each="i : ${#numbers.sequence(開始ページ番号, 終了ページ番号)}"> <!-- ① -->
```

そこで開始ページ番号, 終了ページ番号は使わず、三項演算子(?:)で直接値を戻せるようにします。

【開始ページ番号】

```
todoPage.getNumber()-2 < 0   ?   0   :   todoPage.getNumber()-2
```

【終了ページ番号】

```
todoPage.getNumber()+2 > todoPage.getTotalPages()-1 ?
todoPage.getTotalPages()-1;    :    todoPage.getNumber()+2
```

これを①へ代入すると以下のようになります。

```
<li th:each="i : ${#numbers.sequence(
      todoPage.getNumber()-2 < 0 ? 0 : todoPage.getNumber()-2,
      todoPage.getNumber()+2 > todoPage.getTotalPages()-1 ?
            todoPage.getTotalPages()-1 : todoPage.getNumber()+2)}">
```

これで「表示中ページの前後2ページに限定」できます。

> **th:with属性**を使うと変数を定義できます。興味がある方は調べてみてください。

本書開始の準備(Todolist7x)

前述の演習課題①ではtry～catch文でyyyy-mm-dd形式の日付かチェックしています。しかしこれだと、ぱっと見たとき処理の目的を把握しづらいでしょう。こういった場合、メソッドにして適切な名前を付けるわかりやすくなります。またこのメソッドは他の処理でも利用する可能性があるのでUtilsクラスへ定義します。

ここで以下のメソッドを追加します。コードの内容はサポートサイトからプロジェクトをダウンロードして確認してください。

(1)**public static** Date str2date(String s)
・文字列をyyyy-MM-dd型の日付としてjava.sql.Dateオブジェクトへ変換する。

(2)**public static** Date date2str(Date　date)
・java.sql.Dateオブジェクトをyyyy-mm-dd形式の文字列へ変換する。

(3)**public static boolean** isAllDoubleSpace(String s)
・引数が全角SPACEだけで構成されていればtrueを返す。

(4)**public static boolean** isBlank(String s)
・引数が""または半角SPACE/TABだけで構成されているならtrueを返す。

(5)**public static boolean** isTodayOrFurtureDate(String s) {
・引数が今日以降の日付を表す文字列(yyyy-MM-dd形式)ならtrueを返す。

(6)**public static boolean** isValidDateFormat(String s)
・引数がyyyy-mm-dd形式の日付と解釈できればtrueを返す。
→実装例①の日付チェックをメソッドにしたもの。

これらのメソッドを使うと演習課題①のコードは以下のように書き直せます。メソッド名からどのようなチェックをしているのか把握しやすくなっていると思います(戻り値がboolean型のメソッドはis～()という名前にするのが一般的です)。

```
// 期限が""ならチェックしない
String deadline = todoData.getDeadline();
if (!deadline.equals("")) {
```

```
        // yyyy-mm-dd形式チェック
        if (!Utils.isValidDateFormat(deadline)) {
            FieldError fieldError = new FieldError(
                    result.getObjectName(),
                    "deadline",
                    "期限を設定するときは今日以降にしてください");
            result.addError(fieldError);
            ans = false;

        } else {
            // 過去日付チェックは新規登録の場合のみ
            if (isCreate) {
                // 過去日付ならエラー
                if (!Utils.isTodayOrFurtureDate(deadline)) {
                    FieldError fieldError = new FieldError(
                            result.getObjectName(),
                            "deadline",
                          "期限を設定するときはyyyy-mm-dd形式で入力してください");
                    result.addError(fieldError);
                    ans = false;
                }
            }
        }
    }
```

さらにSTSでisValidDateFormat()にマウスカーソルを置くと、以下のようにメソッドの仕様がポップアップウィンドウ(ホーバー)で表示されます。

表示されているのはisValidDateFormat()の冒頭に/** ～ */で記載したものです。

```
/**
 * 引数がyyyy-mm-ddの日付と解釈できればtrueを返す
 *
 * @param str チェック対象
 * @return true:yyyy-mm-ddと解釈できる、false:解釈できない or 引数がnull or ""
 */
public static boolean isValidDateFormat(String s) {
    :
}
```

これは**JavaDoc**と言うJavaのソースコードからHTML形式のAPI仕様書を生成するためのものです。STSではJavaDocが記述されているメソッドの呼び出し箇所にマウスカーソルを置くと、その内容を上記のように表示します。ソースコードを読まなくてもメソッドがどういうものか知ることができるので非常に便利です(本節でUtilsクラスに追加したメソッドにはJavaDocを記載しています)。

これらのメソッドを使ってTodoServiceクラスを書き直したのがTodolist7xです。本書はこの内容をベースに進めていきます。

> JavaDocの書き方、利用法については各自調べてみてください。

参考資料

書籍

　この本を読んだ後、さらにSpring Bootを学びたい方には、以下の3冊をお薦めします。
　ただしいずれもSpring Boot**3**ではなく、SPring Boot**2**用なので留意してください。
　(書籍情報は2022年12月時点のもの)。

[Spring Boot 2 応用: REST x Swagger UI、MyBatisからAWSへのデプロイまで]

作者:原田 けいと　(著), 竹田 甘地 (著), Robert Segawa (著) / 発売日: 2020/12/28 / 価格：700円
★次に読むならこのシリーズ。本書では扱っていない項目、あるいは同じ項目でもまた別の角度から解説されており、理解度を深めることができます。

[Spring Boot 2 プログラミング入門]

作者:掌田津耶乃 / 発売日: 2018/1/30 / 価格：3,080円(単行本), 2,772円(Kindle版)

★レベルアップを目指すならこの本。「オリジナルのバリデーターを作る」など、Spring Bootの使いこなす上で有用な情報が多数書かれています。ただ掲載されているプログラムリストが見にくいのと、文章が少々わかりにくいのが残念。

Spring徹底入門 Spring FrameworkによるJavaアプリケーション開発(大型本)

作者:株式会社NTTデータ / 発売日: 2016/07/21 / 価格：4,400円(大型本), 3,960円(Kindle版)

★Spring Bootのベースとなっているspring Frameworkに関する書籍。Spring Bootの根本原理を理解したいならこの本は欠かせません。本格的にやるなら手元に置いておきたい1冊。ただし「徹底入門」とあるが入門者用ではない。ある程度知っている人のための本です。

サイト

インターネット上にも数多くの情報源があります。Spring Boot関連で日頃筆者がよく利用させてもらっているのは、以下のサイトです。

Spring Boot
https://spring.io/projects/spring-boot
★Spring Boot開発元のサイト。

Qiita
https://qiita.com/
★プログラマー向け技術情報共有サービス。このなかにSpring Bootに関する記事も数多く含まれている。ただし内容は高度なものが多い印象。

StackOverflow
https://stackoverflow.com/　英語
https://ja.stackoverflow.com/　日本語
★プログラミングに関するQ&Aサイト。英語版は圧倒的なボリュームを持つ。エラーメッセージをキーにしてGoogleで検索すると、ここにたどり着くことが多い印象。

TERASOLUNA Server Framework for Java (5.x) Development Guideline
https://terasolunaorg.github.io/guideline/5.5.1.RELEASE/ja/index.html#
★TERASOLUNAは、株式会社NTTデータの開発している比較的規模が大きなシステム開発手順、フレームワーク、サポートのブランド名です。
「TERASOLUNA Server Framework for Java」はオープンソース化されたフ

レームワークでSpring Frameworkを使っています。このガイドラインはSpringの知識だけでなく、Webアプリケーションを構築する上で示唆に富む内容を数多く含んでおり参考になります。

英語のサイトの方も多いですが、ChromeでGoogle翻訳の拡張機能でページ全体を翻訳すると大体の意味はつかめます。意味不明なところはDeepL(https://www.deepl.com/ja/translator)を使うと、良い結果が得られることもあります。

奥付

菊田 英明(きくた ひであき)

Java言語と出会ったのは1995年の終わりごろ。JDKはまだβ版だった。当初は「趣味」でJavaプログラムを書いていたが、いつのまにか仕事もJava一色となる。その後はWebアプリケーションシステムの開発に従事する。某エンジニアリング会社勤務を経て2019年4月より個人事業主。近年は新入社員向けJava導入教育の講師も請け負っている。

■保有する資格

情報処理技術者試験

プロジェクトマネージャ

アプリケーションエンジニア

プロダクションエンジニア

データベーススペシャリスト

オンライン情報処理技術

基本情報処理技術者

Sun Certified Programmer for the Java Platform

■著書

「実践 JDBC―Javaデータベースプログラミング術 」(オーム社)

「SE・プログラマスタートアップテキストJSP 基礎」(技術評論社)

「基本情報技術者 らくらく突破 Java」(共著、技術評論社)

「Spring Boot3で始めるWebアプリケーション開発入門(基礎編)」(Amazon Kindle)

「Spring Bootで始めるWebアプリケーション開発入門(実戦編)」(Kindle)

表紙デザイン：後藤あゆみ

Spring Boot3で始めるWebアプリケーション開発入門(応用編)

2022年12月27日　初版発行
2022年12月30日　21.3に補足を追加

著者　　菊田英明
発行者　菊田英明