最先端をゆくシステム開発の現場で必要な知識を解説

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

ポートフォリオのその先へ!

多対多の関連、複合主キー、トランザクション管理
排他制御、ロック、論理削除、監査証跡の出力

発展編

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、『基礎編』)」
・「Spring Boot3で始めるWebアプリケーション開発入門(応用編)(以下、『応用編』)」
の続編です。『基礎編』はSpring Bootの基礎知識、『応用編』は(いわゆる)ポートフォリオを作るレベルの内容でしたが、本書はその先、実際のシステム開発で必要とされる知識を解説していきます。
想定している読者は、『基礎編』『応用編』に続き、本書も
・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。

本書の構成

『基礎編』『応用編』を通してToDoアプリケーション(以下、ToDoアプリ)を作成してきましたが、本書でさらに機能を追加します。内容は「**ポートフォリオでは使われないけど、実務では必須**」と思われるものを中心に取り上げています。そういった意味で「**実践を超えた実戦的な知識**」が多く含まれています。

章	タイトル	備考
	はじめに	本章
1章	テーブルの内容からリストボックスを作成する	ToDoアプリの復習
2章	複合主キー	
3章	多対多(n:n)の関連	1:nで表せないデータ(関連)の扱い
4章	Viewを利用したテーブルの検索	
5章	トランザクション管理	同時に同じデータを更新しても破綻しないために
6章	楽観ロックによる排他制御(1)	
7章	楽観ロックによる排他制御(2)	
8章	論理削除	

9章	監査情報の出力	「監査」への対応
10章	監査テーブルの作成	
–	参考情報	

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

「Spring Boot3で始めるWebアプリケーション開発入門(基礎編)」および
「Spring Boot3で始めるWebアプリケーション開発入門(応用編)」の内容

※『基礎編』『応用編』を経由せず本書に取り組むと、かなり苦労すると思います。それは題材とするToDoアプリが、Javaプログラムだけで、**すでに約1,900行**あるためです(コメント、空行除く)。これは実務でも同じですが、元のプログラムが何をやっているかわからないと、余計に難しく感じます。可能であれば『基礎編』『応用編』に目を通された上でチャレンジすることをお勧めします。

・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OKです。
もし不安でしたら都度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

各章は「先輩」と「自分」の会話で始まります。

自分：『応用編』で作成したToDoアプリを社内に公開した人
先輩：ToDoアプリに対して忌憚のない意見を出してくる貴重なユーザー

先輩はToDoアプリの使い勝手に対して、いろいろ言ってきますが、その中に追加すべき機能の要件が含まれています。それを分析し、実現していく、といったストーリーで進めます。

各章初めには作成するプロジェクト名なども記載しています。本書は『応用編』の最後に作成したプロジェクトTodolist15をスタート地点としています。第1章はこれをコピーしてTodolist16を作成し、必要なファイルを追加、変更していく、という手順で進めるとよいでしょう。

また次の章は前の章をコピーして進められるようになっています。プロジェクトのコピー方法については、『基礎編』7章末尾の「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。

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『基礎編』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『基礎編』2章や、インターネットの情報などを参考にしてください。

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

プログラムのインポート

以下の手順で『応用編』の最後に作成したプロジェクトTodolist15をSTSへ取り込みます。1章のTodolist16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。また他のプロジェクトを取り込む場合も、この手順を参考にしてください。

※プロジェクト取り込み方法は何種類かありますが、ここでは影響しない無難な方法を使います。

1)上記サポートサイトから本書の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。
『応用編』ではworkspace**2**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>¥workspace3**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Todolist15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。

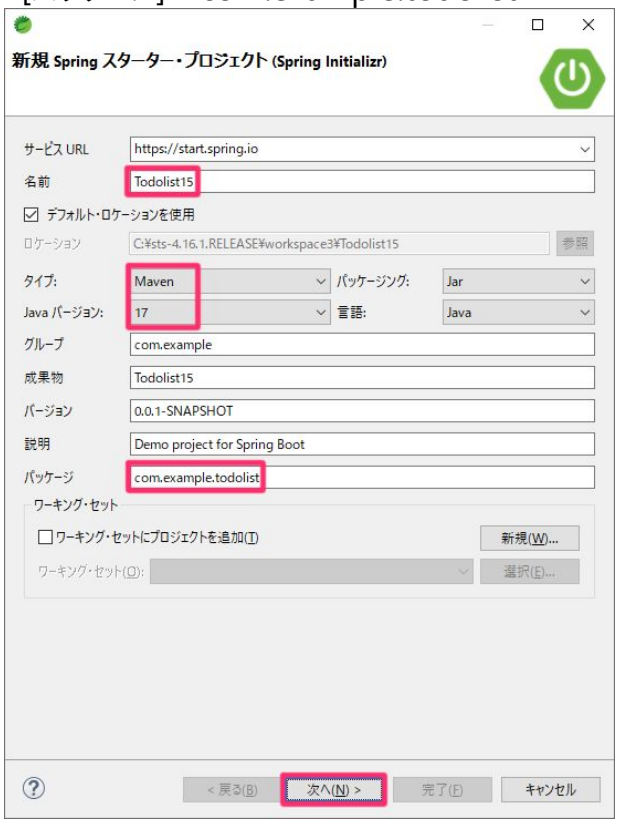
以下を入力して[次へ(N)]ボタンをクリックする。

・[名前]：Todolist15

・[タイプ]：Maven

・[Javaバージョン]：17

・[パッケージ]：com.example.todolist

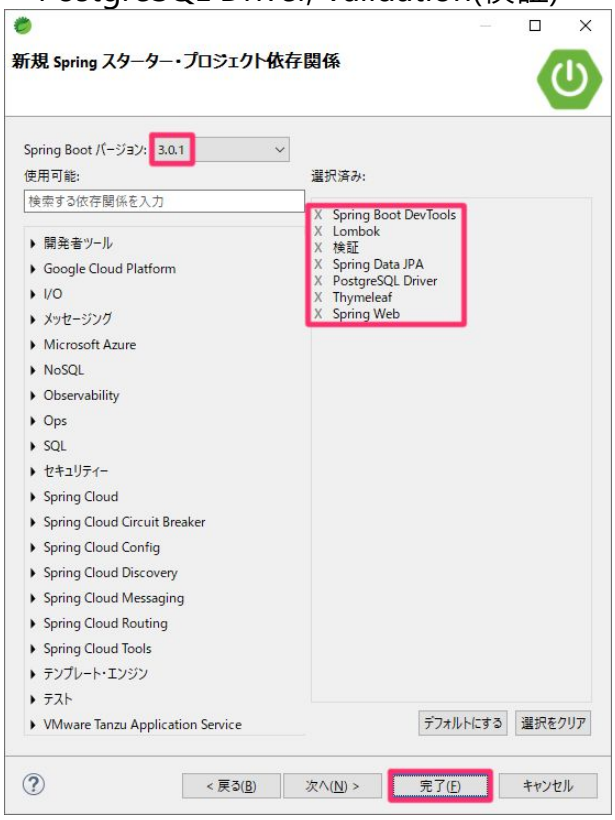


3-3)依存関係を設定する

[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。

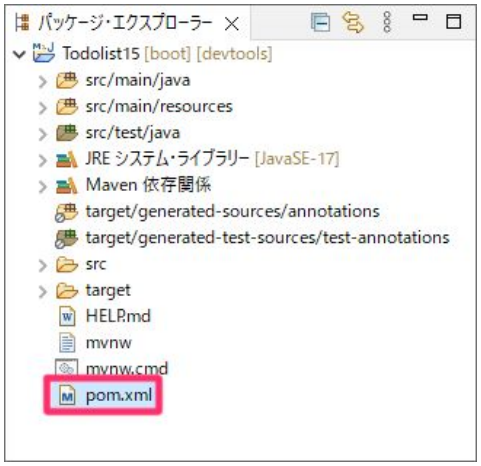
さらに以下を選択し、[完了(F)]ボタンをクリックする。

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



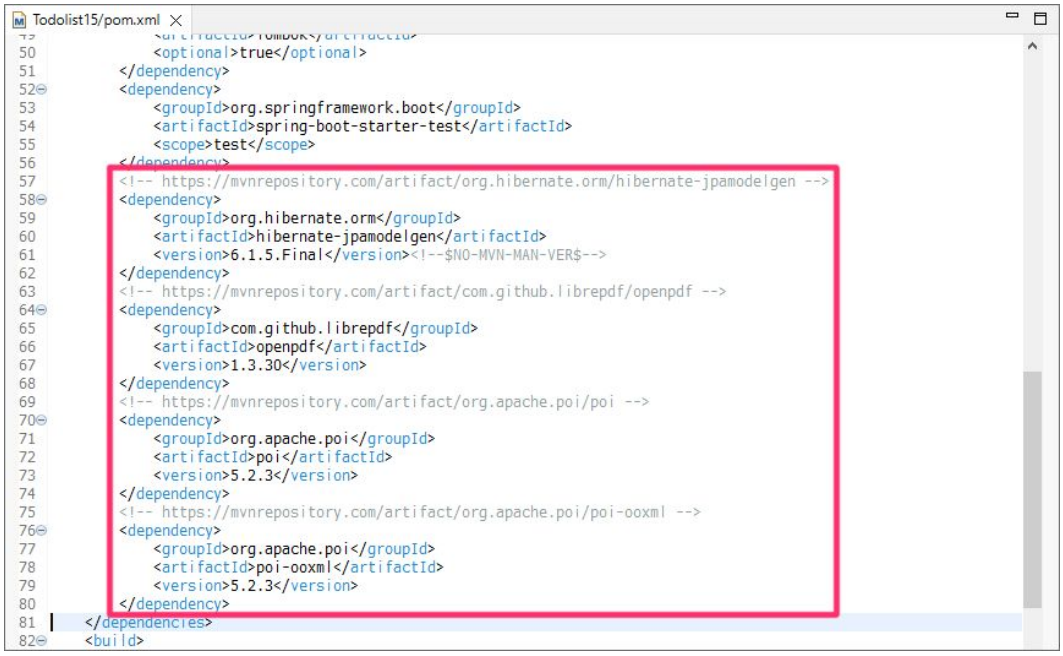
4)Mavenリポジトリ(メタクラス生成ツールhibernate-jpamodelgenなど)の取得

4-1)パッケージ・エクスプローラーでTodolist15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
Todolist15 [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)。
→このタイミングで実行に必要なjarファイルがダウンロードされる(ダウンロードされていなければ)。

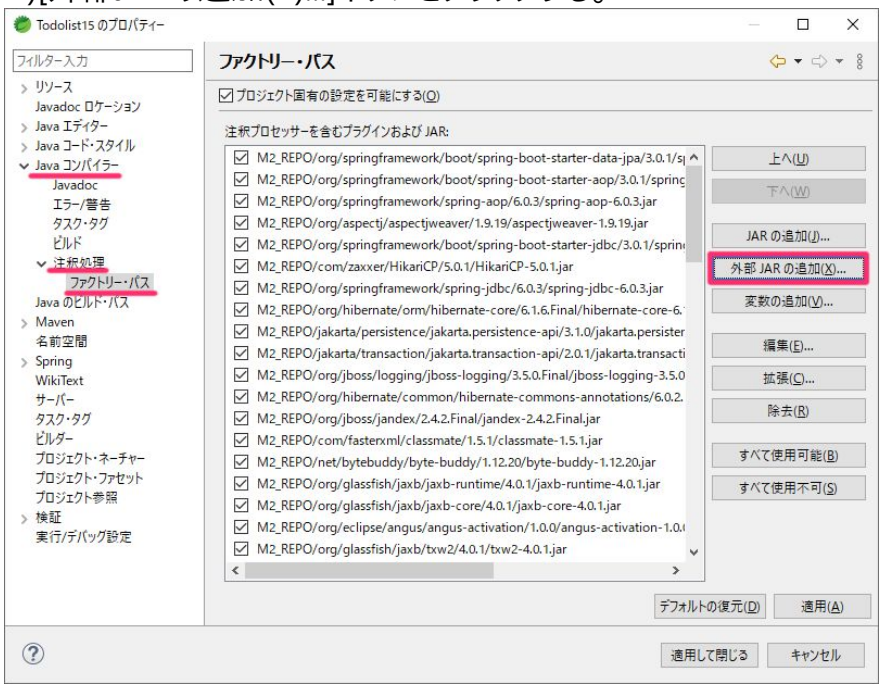
```
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-jpamodelgen -->
<dependency>
	<groupId>org.hibernate.orm</groupId>
	<artifactId>hibernate-jpamodelgen</artifactId>
	<version>6.1.5.Final</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<!-- https://mvnrepository.com/artifact/com.github.librepdf/openpdf -->
<dependency>
	<groupId>com.github.librepdf</groupId>
	<artifactId>openpdf</artifactId>
	<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>
```


Todolist15/pom.xml
<optional>true</optional>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-jpamodelgen -->
<dependency>
<groupId>org.hibernate.orm</groupId>
<artifactId>hibernate-jpamodelgen</artifactId>
<version>6.1.5.Final</version><!--$NO-MVN-MAN-VER$-->
</dependency>
<!-- https://mvnrepository.com/artifact/com.github.librepdf/openpdf -->
<dependency>
<groupId>com.github.librepdf</groupId>
<artifactId>openpdf</artifactId>
<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>

4-3)パッケージ・エクスプローラーでTodolist15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]を選択する。

4-4)[外部JARの追加(X)...]ボタンをクリックする。



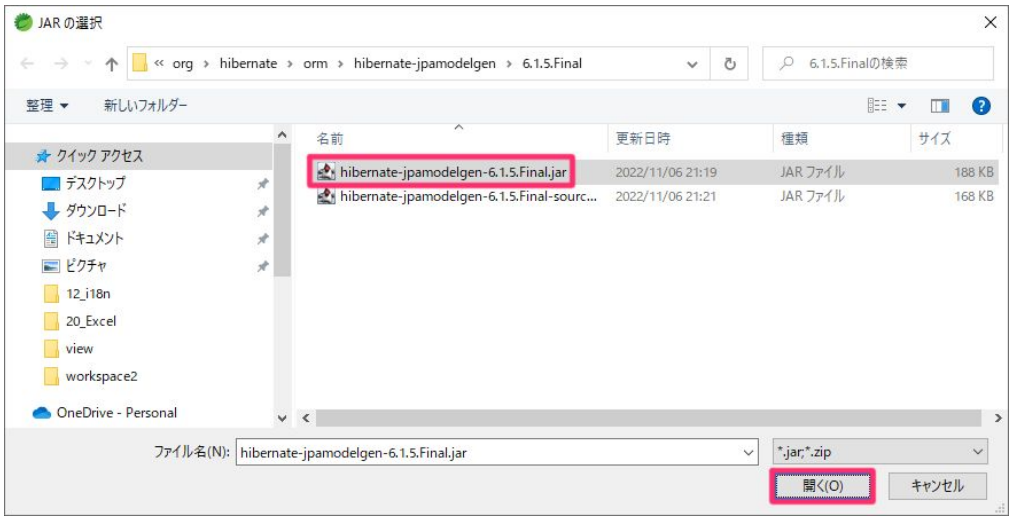
4-5)hibernate-jpamodelgenのjarを選択し[開く(O)]ボタンをクリックする。

■格納場所:

C:¥Users¥<ユーザー名>¥.m2¥repository¥org¥hibernate¥orm¥hibernate-jpamodelgen¥**6.1.5.Final**

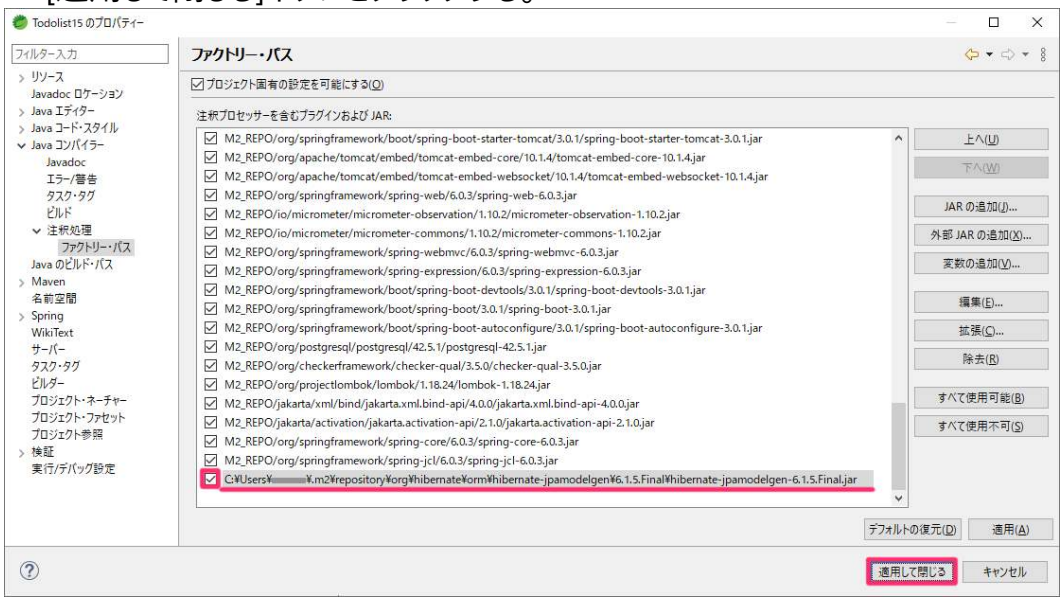
■ファイル名:

hibernate-jpamodelgen-**6.1.5.Final**.jar

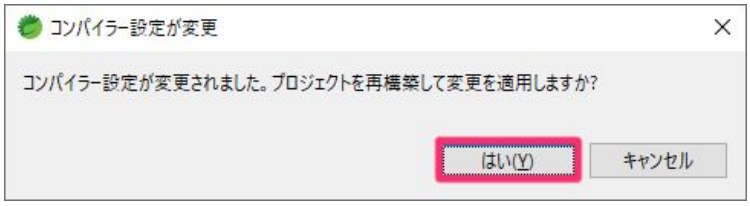

JAR の選択
org › hibernate › orm › hibernate-jpamodelgen › 6.1.5.Final
6.1.5.Finalの検索
整理
新しいフォルダー
クイック アクセス
デスクトップ
ダウンロード
ドキュメント
ピクチャ
12_i18n
20_Excel
view
workspace2
OneDrive - Personal
名前
更新日時
種類
サイズ
hibernate-jpamodelgen-6.1.5.Final.jar
2022/11/06 21:19
JAR ファイル
188 KB
hibernate-jpamodelgen-6.1.5.Final-sourc...
2022/11/06 21:21
JAR ファイル
168 KB
ファイル名(N):
hibernate-jpamodelgen-6.1.5.Final.jar
.jar;.zip
開く(O)
キャンセル

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

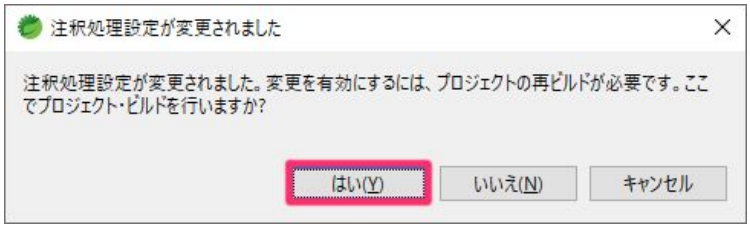
[適用して閉じる]ボタンをクリックする。



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」ダイアログが表示されたら[はい(Y)]ボタンをクリックする。



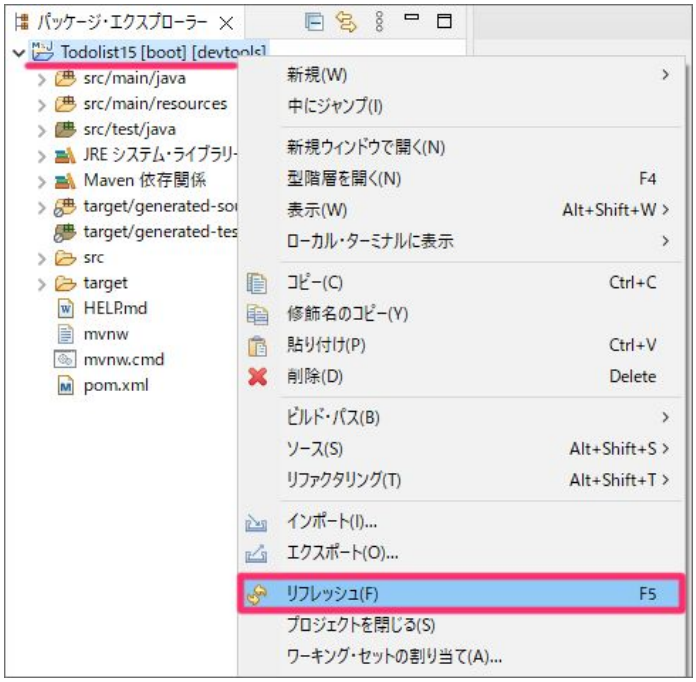
5)ソースコードのコピー

5-1)**エクスプローラー**で<STS_DIR>¥workspace3¥Todolist15を開く。

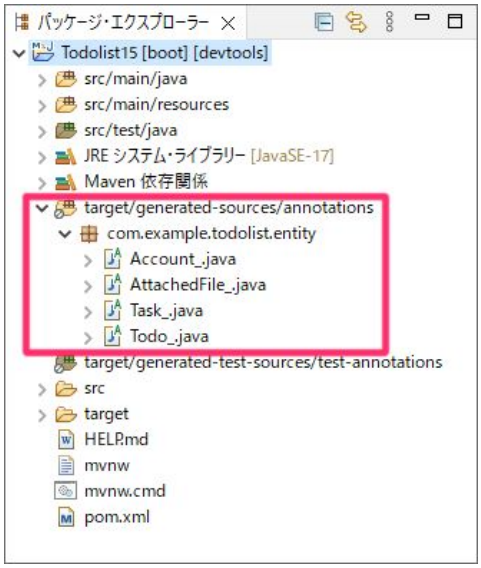
→STSのパッケージ・エクスプローラーではないの注意

5-2)1)の解凍結果に含まれるTodolist15下のsrcフォルダを上記5-1)へ上書きコピーする。

5-3)**パッケージ・エクスプローラー**でプロジェクトTodolist15を右クリック > [リフレッシュ(F)]をクリックする。



5-4)target/genarated-sources/annotaions下にTodo_.javaなどのメタクラス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)]ボタンをクリックしてみてください。

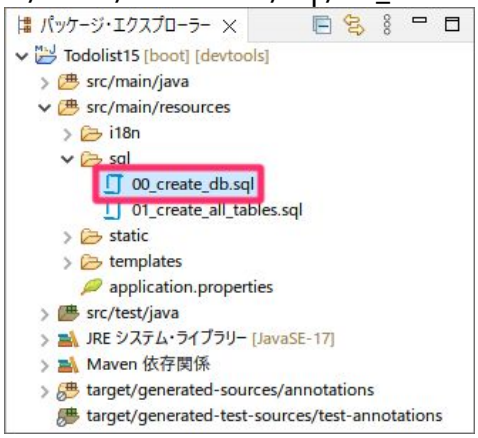
データベース

PostgreSQLを新たにインストールした場合、本書用のDB(tododb)を作成します。
本書で使用するDBは以下のような仕様です。

・DB名：tododb
・所有者：todouser
・所有者のパスワード：pass
・データベースの文字コード：UTF8

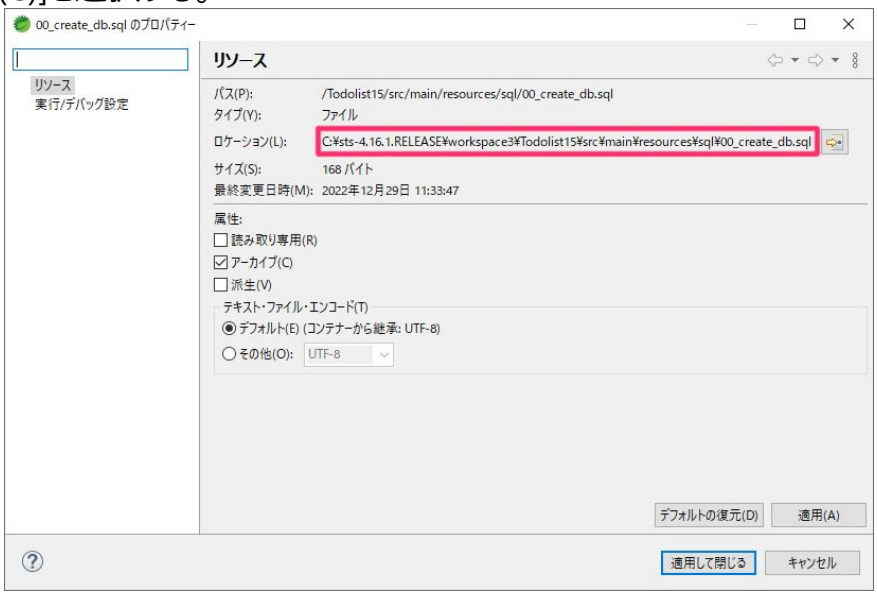
※『基礎編』『応用編』から継続される方は作成済のため、**本項の操作は不要**です。
次の「本書開始の準備(テーブル)」へ進んでください。

■データベース作成手順
・データベースを作成するSQLファイル(スクリプト)は00_create_db.sqlです。
・全プロジェクトに含めてあります(/<プロジェクト名>/src/main/resources/sql/00_create_db.sql)



作成手順は以下の通りです。
1)以下どちらかの方法で、プロパティダイアログを表示する
・00_create_db.sqlを選択し、[Alt]+[Enter]を押す　または
・00_create_db.sql右クリック→プロパティ(R)を選択

2)「プロパティー」ダイアログの[ロケーション(L)]の内容をマウスで範囲選択後、右クリック>[コピー(C)]を選択する。



※このダイアログは[キャンセル]ボタンをクリックして閉じる。

3)コマンドプロンプトを開く。

4)以下のコマンドを実行する

```
psql -U postgres -f <コピーしたSQLファイルのフルパス>
```

→PostgreSQLへpostgres(スーパーユーザー)として接続し、-fで指定されたファイルを実行する

例.

```
psql -U postgres -f    C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist15¥src¥main¥resources¥sql¥
00_create_db.sql
```

※1行で入力してください

※実行するとpostgresのパスワードを求められるので、**インストール時に設定したもの**を入力してください。

■実行例1：tododb/todouser が存在しなかった場合

```
>psql -U postgres -f C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist15¥src¥main¥resources¥
sql¥00_create_db.sql
ユーザー postgres のパスワード:
psql:C:/sts-
4.16.1.RELEASE/workspace3/Todolist15/src/main/resources/sql/00_create_db.sql:
1
: NOTICE:  データベース"tododb"は存在しません、スキップします
DROP DATABASE
psql:C:/sts-
4.16.1.RELEASE/workspace3/Todolist15/src/main/resources/sql/00_create_db.sql:
2
: NOTICE:  ロール"todouser"は 存在しません、スキップします
DROP ROLE
CREATE ROLE
CREATE DATABASE

>
```

■実行例2：tododb/todouser が存在した場合

```
>psql -U postgres -f C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist15¥src¥main¥resources¥
sql¥00_create_db.sql
ユーザー postgres のパスワード:
DROP DATABASE
DROP ROLE
CREATE ROLE
CREATE DATABASE

>
```

どちらも場合も、「CREATE ROLE」「CREATE DATABASE」が表示されればOKです。

テーブル

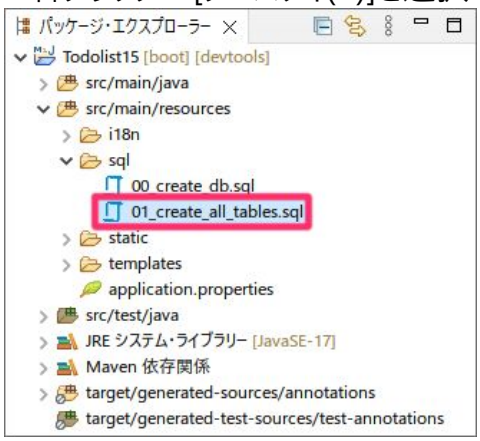
テーブルの作成、変更(列の追加、など)は、以下の手順で実行してください。

1)変更方法を決める
　2 つのやり方があります。
　a. 前の章との違い(差分)だけ適用する。
　　・本文の説明に合わせて追加・変更していきます。ご自分で登録したデータは、そのまま残ります。
　　→ただし、本文中のスクリーンショットと内容が一致しません。
　　・説明文中、および各章**冒頭の「SQLファイル」を実行**してください。

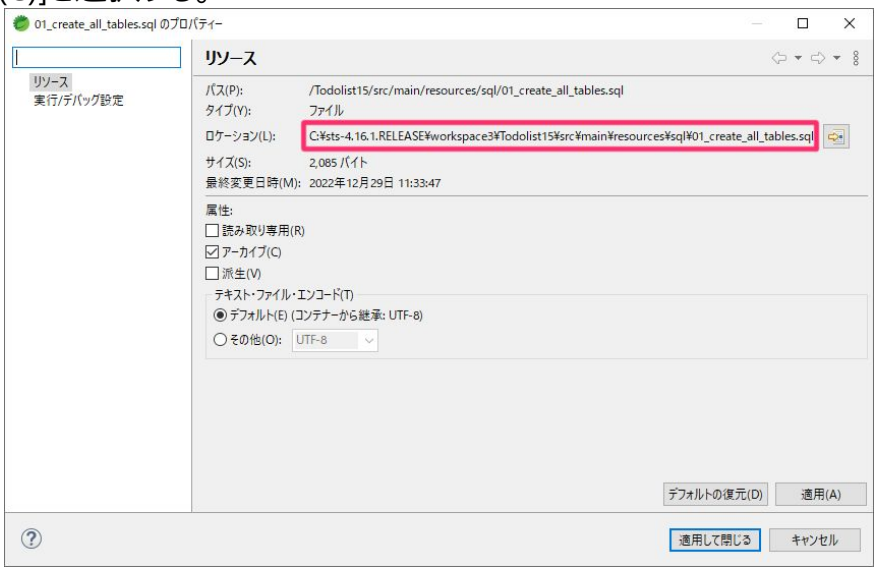
　b. 全テーブル再作成
　　・その章での追加・変更部分を含んだ内容で、すべてのテーブルを作り直す。
　　・何度でも実行できますが、データが都度初期状態に戻るので注意してください。
　　・各プロジェクトの**sql¥01_create_all_tables.sqlを実行**してください。
　→以下、こちらの方法で説明します。

2)実行するSQLファイルのプロパティダイアログを開く
　・プロジェクト・エクスプローラーでファイルを選択し、[Alt]+[Enter]を押す　または

・右クリック→[プロパティ(R)]を選択



3)「プロパティー」ダイアログの[ロケーション(L)]の内容をマウスで範囲選択後、右クリック>[コピー(C)]を選択する。



※このダイアログは[キャンセル]ボタンをクリックして閉じる。

4)以下のコマンドを実行する

```
psql -U todouser -d tododb -f <コピーしたSQLファイルのフルパス>
```

→PostgreSQLへユーザーtodouserとして接続し、データベースtododbに-fで指定されたファイルを実行する

例.

```
psql -U todouser -d tododb -f C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist15¥src¥main¥resources¥
sql¥01_create_all_tables.sql
```

※1行で入力してください
※実行するとtodouserのパスワードを求められるので、**pass**を入力してください。

■実行例(パスワードはpass)

```
>psql -U todouser -d tododb -f C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist15¥src¥
main¥resources¥sql¥01_create_all_tables.sql
ユーザー todouser のパスワード:
DROP TABLE
CREATE TABLE
INSERT 0 1
:
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1

>
```

■補足

本文中のSQLファイルには記載していませんが、漢字(いわゆる全角文字)を含むSQL文がある場合は、sqlフォルダのファイル(*.sql)の１行目には**¥encoding UTF8**を挿入してあります。

例.10_create_category.sql

```
¥encoding　UTF8;
DROP　TABLE　IF　EXISTS　category;
CREATE　TABLE　category
(
　　id　　SERIAL　PRIMARY KEY,
　　name　　TEXT
);
INSERT　INTO　category(name)　VALUES('仕事');
INSERT　INTO　category(name)　VALUES('勉強');
INSERT　INTO　category(name)　VALUES('レジャー');
```

これは本書のダウンロードサイトで公開しているSQLファイル(*.sql)の文字コードが**UTF-8**なのに対し、Windowsのpsqlは**Shift-JIS**前提だからです(データベースのテーブルの文字コードはUTF-8です)。このギャップを埋めるのが、前述の「¥encoding UTF8;」です。これで「**このファイルの文字コードはUTF-8**」ということをpsqlへ知らせ、適宜文字コード変換をしてもらいます。

ただし、¥encoding UTF8がある場合、**psqlのメッセージが文字化けします**。

→以下のように“NOTICE”なら問題ありません。

```
C:¥>psql -U todouser -d tododb -f C:¥sts-
4.16.1.RELEASE¥workspace3¥Todolist19¥src¥main¥resou
rces¥sql¥01_create_all_tables.sql
ユーザー todouser のパスワード:
psql:C:/sts-
4.16.1.RELEASE/workspace3/Todolist19/src/main/resources/sql/01_create_all_tab
l
es.sql:7: NOTICE:　綱・・綱悶NDROP TABLE蟄伜恵縺励∪縺帙ｓ繧√ｾせ繧ｭ綱・・縺励∪縺・
```

```
CREATE TABLE
    :
```

メッセージ内容を調べたい方は、以下のようにしてください。

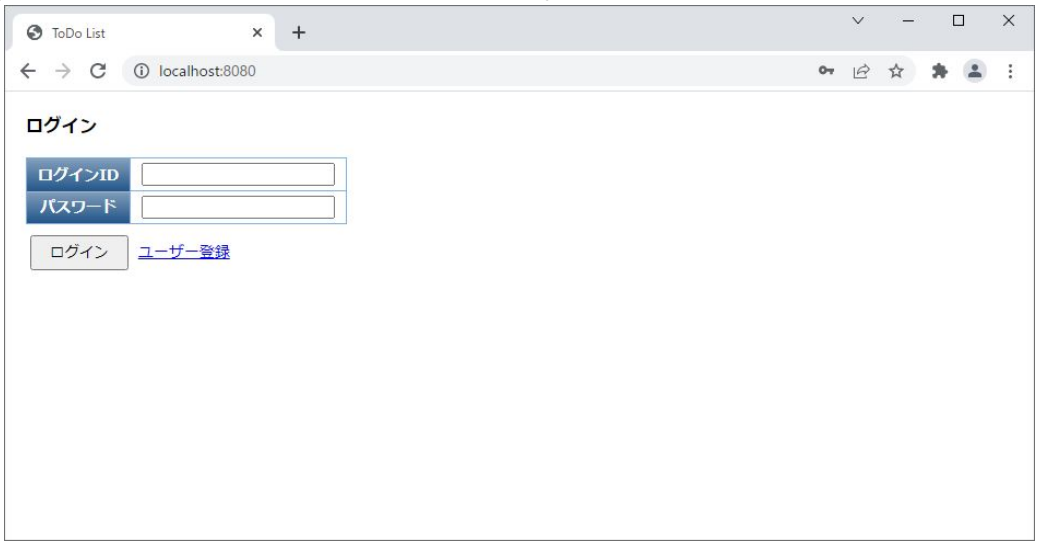
1)psqlへログイン後、SQLファイルの「¥encoding UTF8」以外の行をコピー＆ペーストして実行する。

2)PostgreSQLの設定を変更し、メッセージを英語にする。

→「psql メッセージ 文字化け」で検索すると変更方法の解説記事が見つかるので、それを参考にしてください。

動作確認

1)Todolist15を起動し、http://localhost:8080/todoへアクセスする。
2)ログイン画面が表示されることを確認する。



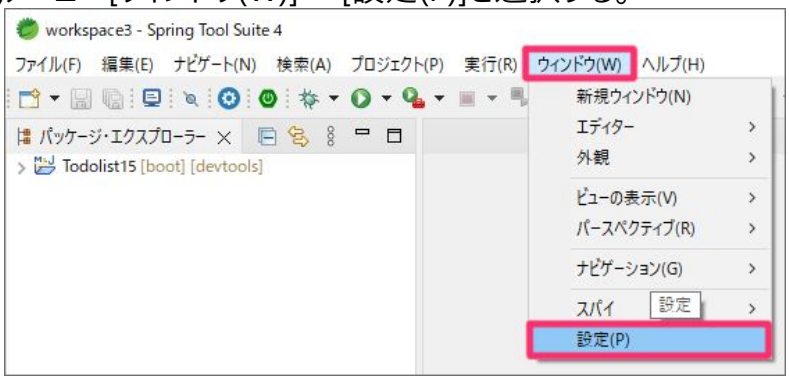
3)ログインID：**okada**, パスワード：**s6rizqfk**　を入力　→　[ログイン]ボタンをクリックする。
4)ToDo一覧が表示されることを確認する。

※上記操作で下記「Ansiコンソール」ダイアログが表示された場合

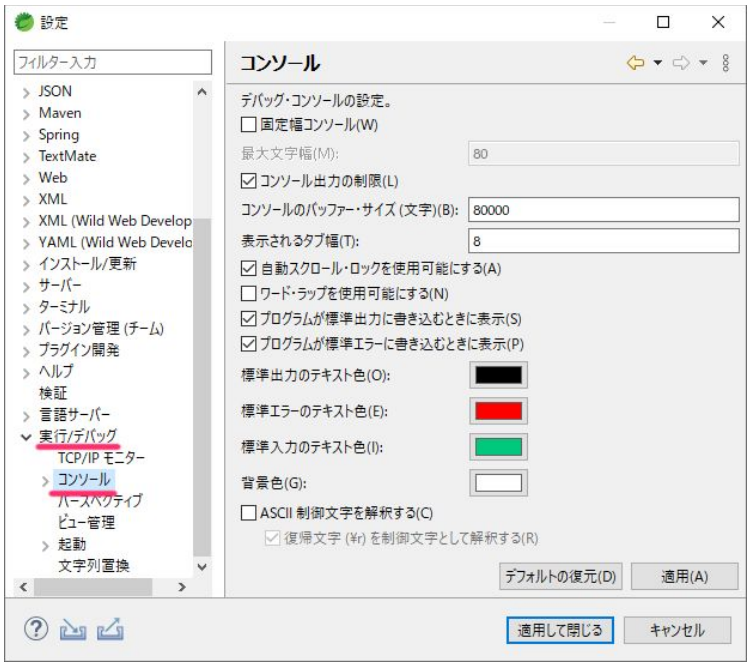
・[Never remind me again]をクリックすると、このダイアログは今後表示されない。
　→コンソールへの出力が若干遅くなる状態のまま
・[後で通知]をクリックするとダイアログは消えるが、以下の設定を行わないと、また表示される。

設定の手順は以下の通り。

1)メニュー [ウィンドウ(W)] > [設定(P)]を選択する。



2)[実行/デバッグ] > [コンソール]を選択する。


設定
フィルター入力
JSON
Maven
Spring
TextMate
Web
XML
XML (Wild Web Develop
YAML (Wild Web Develo
インストール/更新
サーバー
ターミナル
バージョン管理 (チーム)
プラグイン開発
ヘルプ
検証
言語サーバー
実行/デバッグ
TCP/IP モニター
コンソール
パースペクティブ
ビュー管理
起動
文字列置換
コンソール
デバッグ・コンソールの設定。
固定幅コンソール(W)
最大文字幅(M):
80
コンソール出力の制限(L)
コンソールのバッファー・サイズ (文字)(B):
80000
表示されるタブ幅(T):
8
自動スクロール・ロックを使用可能にする(A)
ワード・ラップを使用可能にする(N)
プログラムが標準出力に書き込むときに表示(S)
プログラムが標準エラーに書き込むときに表示(P)
標準出力のテキスト色(O):
標準エラーのテキスト色(E):
標準入力のテキスト色(I):
背景色(G):
ASCII 制御文字を解釈する(C)
復帰文字 (¥r) を制御文字として解釈する(R)
デフォルトの復元(D)
適用(A)
適用して閉じる
キャンセル

3)以下のどちらかを設定後、[適用して閉じる]をクリックする。

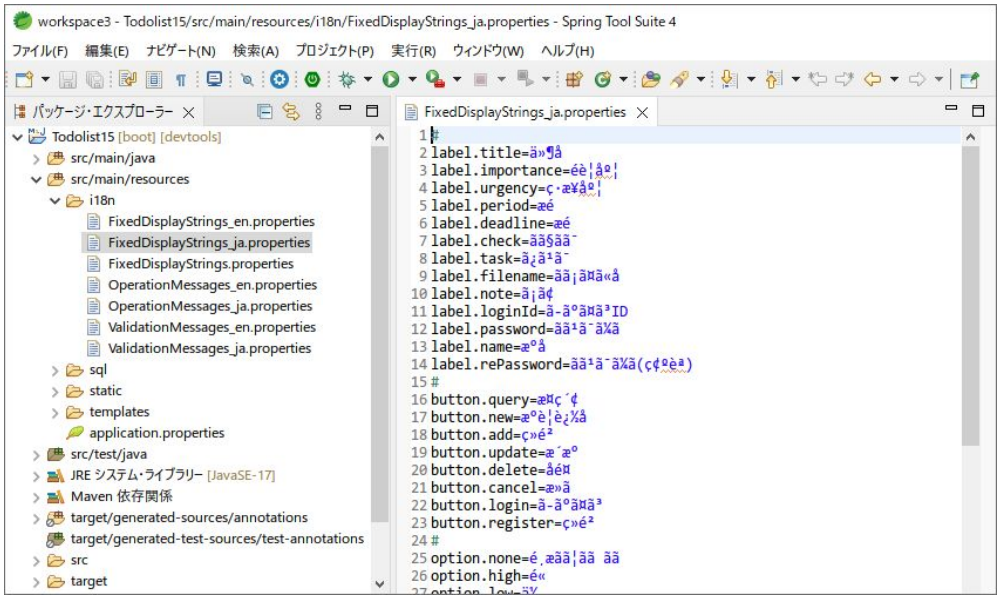
・[コンソール出力の制限(L)]をチェックしたまま[コンソールのバッファー・サイズ(文字)(B)]にダイアログの提案した値を入力

→ここでは1000000を入力

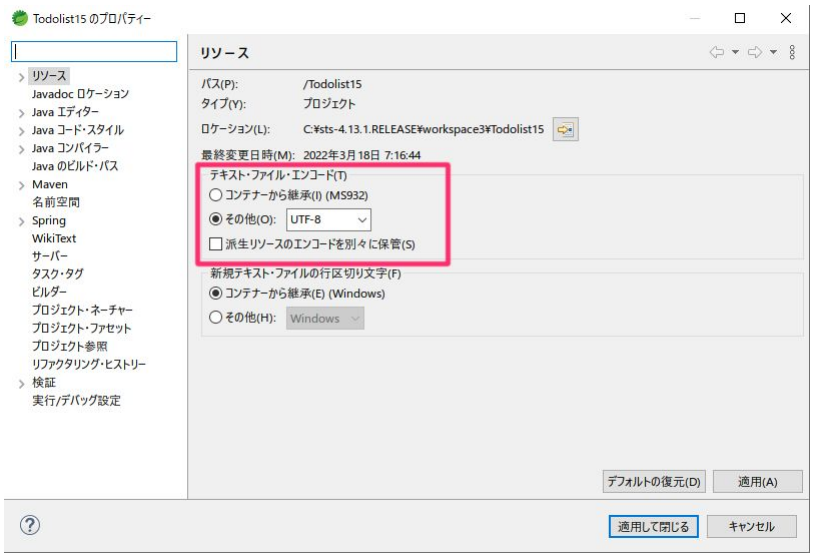
・[コンソール出力の制限(L)]をクリアする

プロパティファイルの文字化け対策

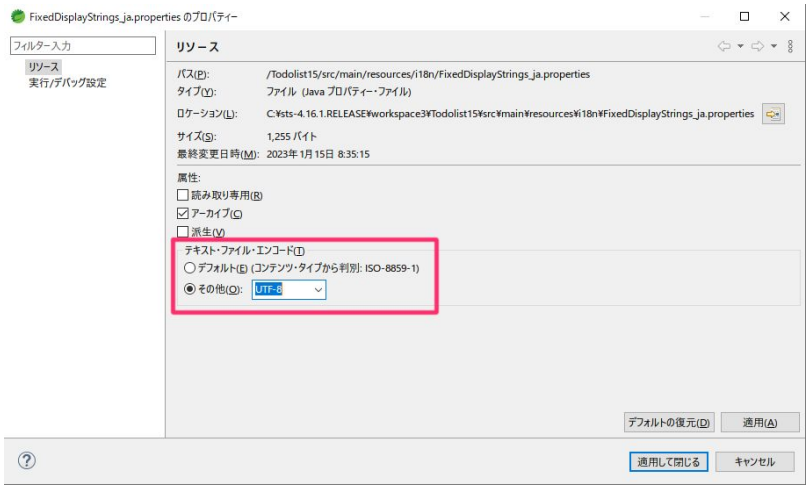
インストール直後のSTSでプロパティファイル(拡張子properties)を開くと、日本語が文字化けします。



プロジェクトのエンコーディングは、下記のようにUTF-8ですが、これはプロパティファイルへ反映されません。


Todolist15 のプロパティー
リソース
Javadoc ロケーション
Java エディター
Java コード・スタイル
Java コンパイラー
Java のビルド・パス
Maven
名前空間
Spring
WikiText
サーバー
タスク・タグ
ビルダー
プロジェクト・ネーチャー
プロジェクト・ファセット
プロジェクト参照
リファクタリング・ヒストリー
検証
実行/デバッグ設定
リソース
パス(P): /Todolist15
タイプ(Y): プロジェクト
ロケーション(L): C:¥sts-4.13.1.RELEASE¥workspace3¥Todolist15
最終変更日時(M): 2022年3月18日 7:16:44
テキスト・ファイル・エンコード(T)
コンテナーから継承(I) (MS932)
その他(O): UTF-8
派生リソースのエンコードを別々に保管(S)
新規テキスト・ファイルの行区切り文字(F)
コンテナーから継承(E) (Windows)
その他(H): Windows
デフォルトの復元(D)
適用(A)
適用して閉じる
キャンセル

解消策としては、プロパティファイルが少なければ、以下のようにファイル単位でエンコーディングをUTF-8へ変更する方法もあります。



しかし変更したファイルをコピーすると、エンコーディングがデフォルトのISO-8859-1に戻るため、また文字化けします。

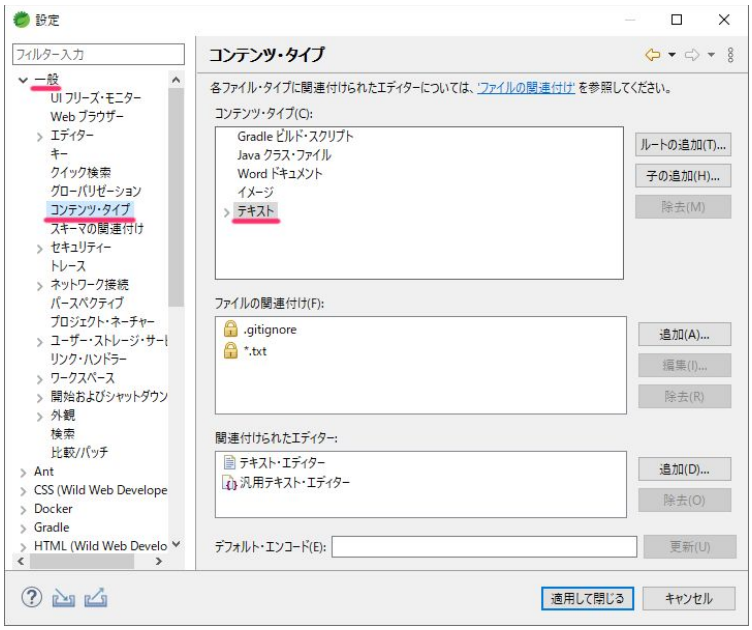
そこで以下のような方法で、STSに「**プロパティファイル(拡張子properties)はUTF-8で扱う**」ことを設定します。

設定の手順は以下の通りです。

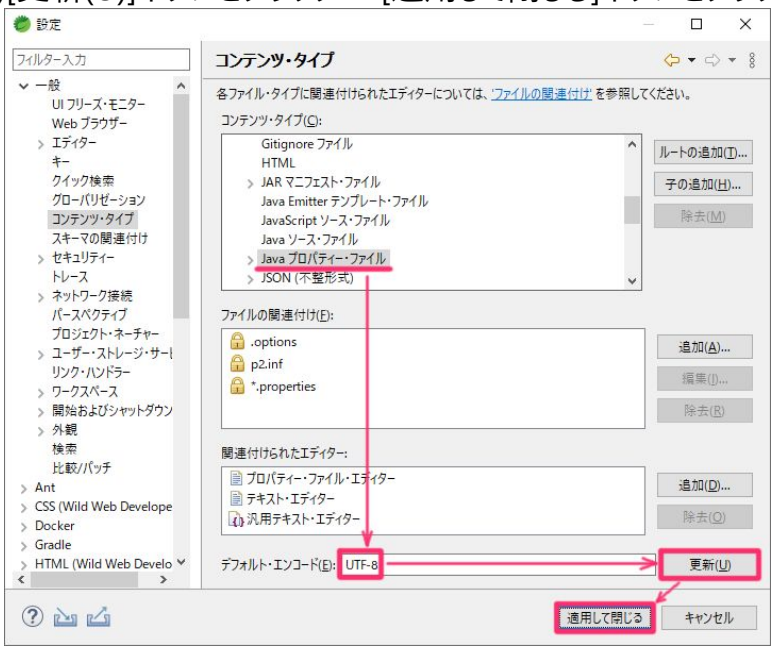
1)メニュー [ウィンドウ(W)] > [設定(P)]を選択する。

2)左のツリーメニュー［一般］ > ［コンテンツ・タイプ］を選択する。

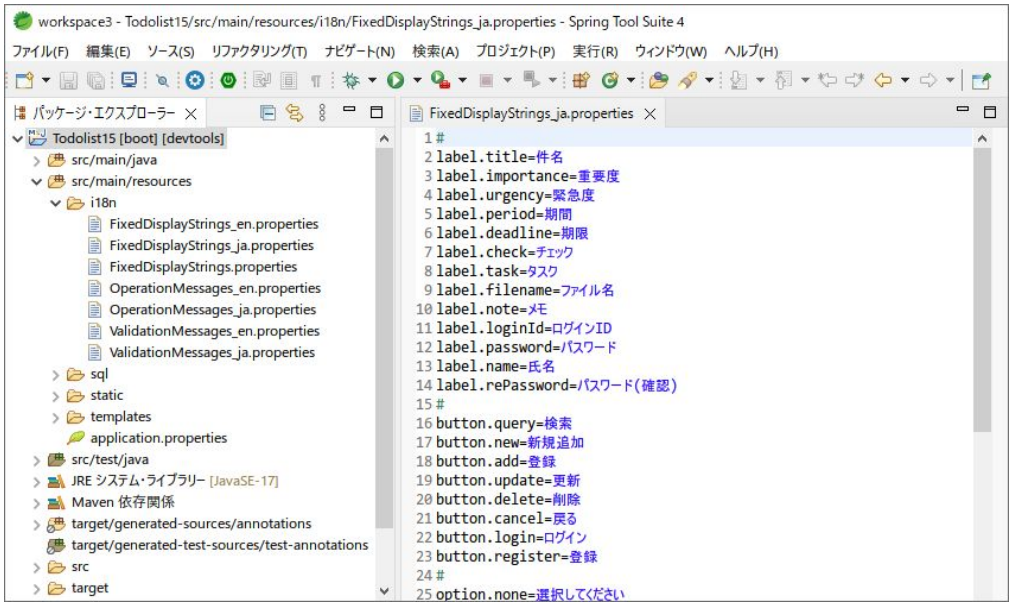
3)右側のコンテンツ・タイプ(C)の中にある[テキスト]を展開する(行頭の>をクリックする)



4)[Javaプロパティー・ファイル]を選択 > [デフォルトエンコーディング(E)]を UTF-8 へ変更。
5)[更新(U)]ボタンをクリック > [適用して閉じる]ボタンをクリックする。



これで文字化けが解消します。



では本書の内容に入っていきましょう。

1. テーブルの内容からリストボックスを作成する

プロジェクト名	Todolist16
作成ファイル	com.example.todolist.entity.Category.java com.example.todolist.repository.CategoryRepository.java
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java src/main/resources/templates/todoForm.html src/main/resources/templates/todoList.html src/main/resources/i18n/FixedDisplayStrings_en.properties src/main/resources/i18n/FixedDisplayStrings_ja.properties src/main/resources/i18n/ValidationMessages_en.properties src/main/resources/i18n/ValidationMessages_ja.properties
SQLファイル	src/main/resources/sql/10_create_category.sql src/main/resources/sql/11_add_categoryId_to_todo.sql

ある日オフィスの片隅で...

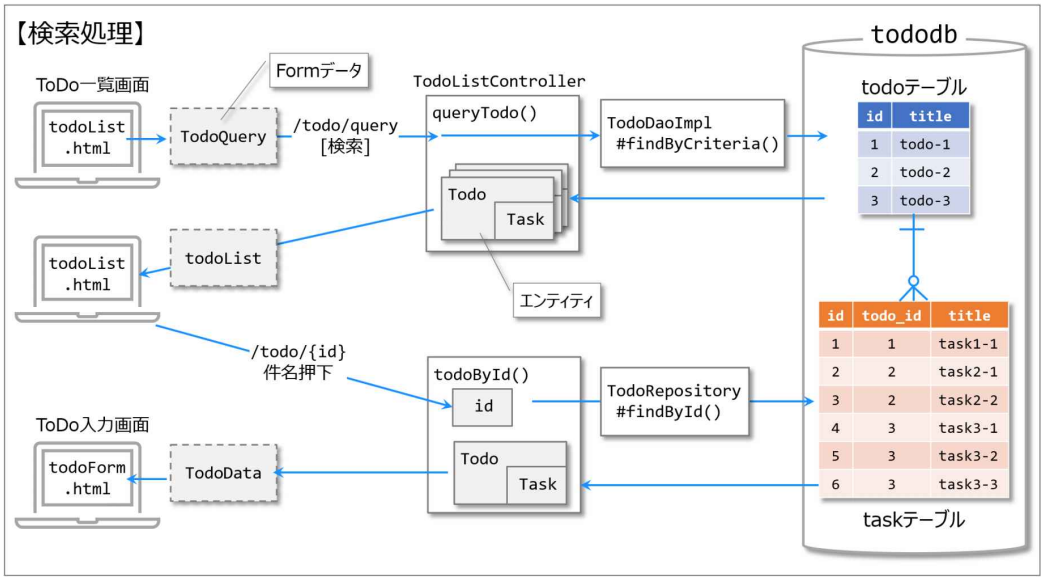
先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「仕事のToDoもプライベートなやつも区別しないんだな」
自分「？」
先輩「分けられない？」
自分「タブを複製して表示するとかでは？」
先輩「うーん、タブも悪くないけど、まとめて見たい時もあるな...」
自分「...(どっちやねんw)」
先輩「『仕事』『プライベート』、みたいなカテゴリ付けられない？」
自分「属性を追加しろと？」
先輩「そうなるかな。まぁいい感じで頼むわ...」ｽﾀｺﾗｻｯｻ
自分「はぁ...」

本章ではToDoアプリへ「カテゴリ」というリストボックスを追加しながら、アプリ全体の構成を復習していきます。また「**アプリケーションスコープ**」という仕組みも取り入れます。

1.1 ToDoアプリの全体構成(復習)

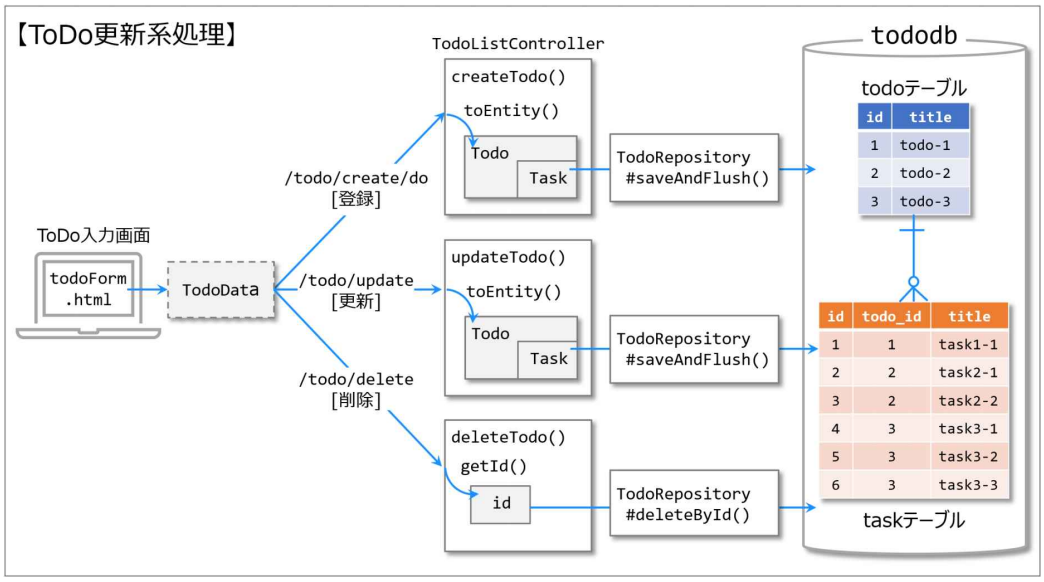
ToDoのデータはtodoテーブル/taskテーブルで管理します。「１つのToDoは、0個以上のタスクを持つ」ので、1:nの関係です。プログラムではテーブルに対応するTodoエンティティ/Taskエンティティを定義し、アノテーション(@OneToMany, @ManyToOne)で関係を表します。

検索処理は、画面に入力された検索条件をTodoQueryというFormデータクラスのオブジェクトに格納し、処理を制御するコントローラークラスTodoListControllerへ渡します。検索結果がToDo一覧のときはtodoListという名前のオブジェクトを画面(テンプレート)へ返します。またToDo１件だけならTodoDataを使います。この中にはToDoだけでなく、関連するタスクの情報も含んでいます。このように画面からの入力、画面への出力にはFormデータが介在しています。



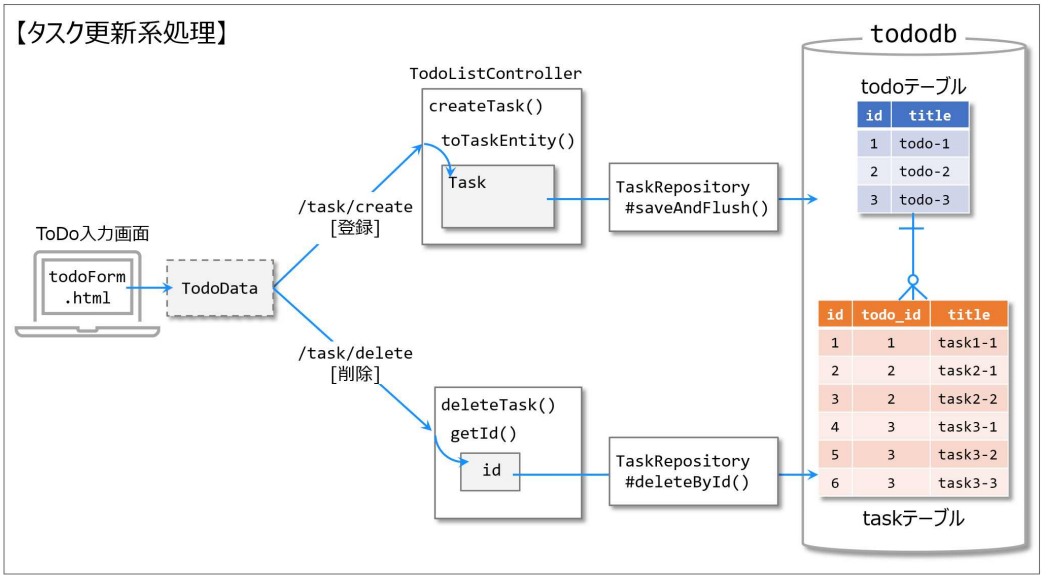
【図1-1】ToDoアプリの構成(1/3)

ToDoの更新系処理(追加、変更、削除)も、画面に入力された内容をTodoDataへ格納します。これには変更されたタスクの情報も含まれており、テーブルtodo/taskをまとめて更新します。



【図1-2】ToDoアプリの構成(2/3)

タスクの更新系処理も同じ構造ですが、追加/削除だけです。更新は前述のToDoと一緒に行います。また画面からTodoDataを受け取りますが、使用するのはその中のタスクデータの部分だけです。



【図1-3】ToDoアプリの構成(3/3)

　これらの図を見ると、ToDoアプリへ項目(ここでは「カテゴリ」)を追加するには、画面/テーブルだけでなく、それらをつなぐエンティティ、Formデータにも項目追加が必要、ということがわかると思います。これはコントローラーやバリデーションの処理にも影響を与えます。

> ➡ToDoアプリの詳細については『基礎編』『応用編』を参照してください。

> ToDoアプリにはLoginやPDF/Excel出力機能などもありますが、本書では変更しないので、ここでは割愛しています。

1.2 カテゴリリストの概要

本章ではToDoのカテゴリを、リストボックス(<select>)で入力できるようにします。
すでにToDoアプリでは、「緊急度」をリストボックスで入力していますが、選択肢はリソースファイル(FixedDisplayStrings_*.properties)で定義しています。

■ToDo入力画面(src/main/resources/templates/todoForm.html)の緊急度入力欄

```
<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>
```

これに対し本章で追加するカテゴリの選択肢は、テーブルで管理します。テーブルにデータを追加すれば、それも選択肢となるようにします。

■本章で追加するカテゴリ入力欄(後述)

```
<select name="categoryId">
  <option th:each="c : ${application.categoryList}" th:value="${c.id}"
th:text="${c.name}"
          th:selected="${c.id} == *{categoryId}"></option>
</select>
```

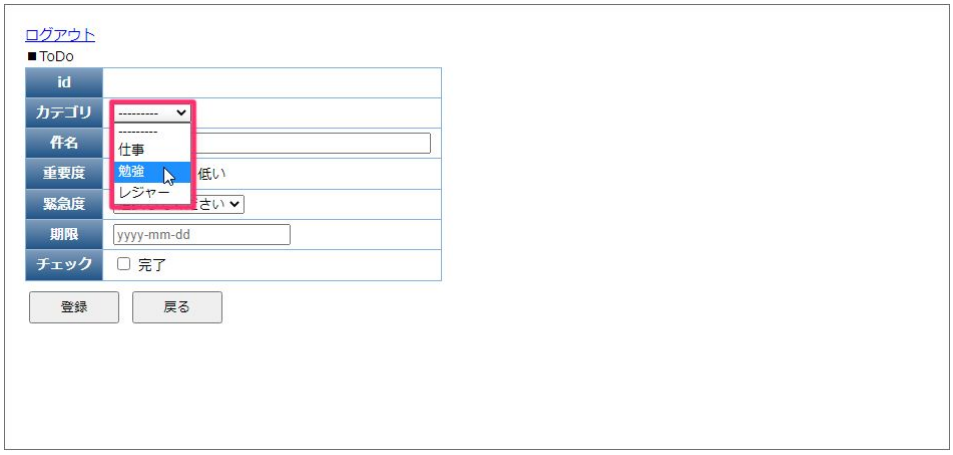
以下が本章終了時のToDoアプリです。詳細は次節以降で説明します。ここではイメージだけ掴んでください。

(1)テーブルに登録してあるカテゴリの選択肢

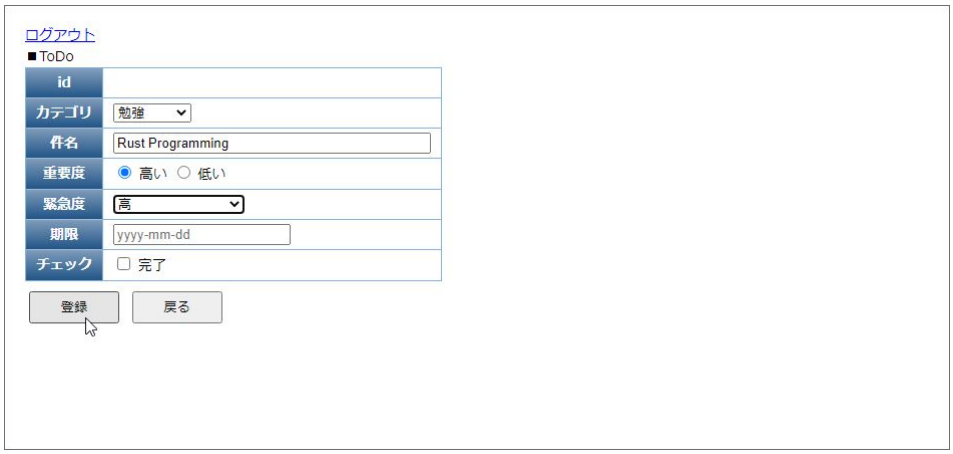
```
tododb=> SELECT * FROM category;
 id |   name
----+----------
  1 | 仕事
  2 | 勉強
  3 | レジャー
(3 行)
```

(2)ToDo一覧にカテゴリ列が追加されている

(3)カテゴリをリストボックスで選択(選択肢はテーブルの内容)



(4)他の項目も入力→[登録]



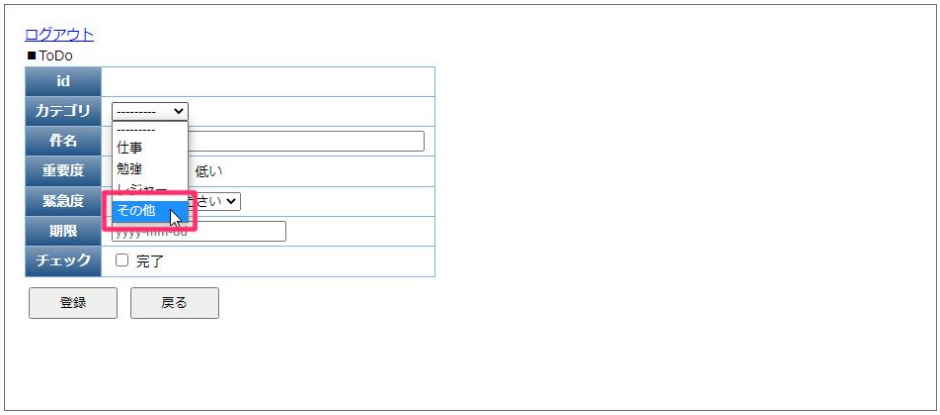
(5)選択したカテゴリが設定された

(6)ToDo一覧にも反映される

(7)カテゴリを追加→ToDoアプリ再起動

```
tododb=> INSERT INTO category (name) VALUES('その他');
INSERT 0 1
tododb=> SELECT * FROM category;
 id |   name
----+----------
  1 | 仕事
  2 | 勉強
  3 | レジャー
  4 | その他
(4 行)
```

(8)再度ログインすると、追加したカテゴリも選択肢になっている



1.3 categoryテーブル追加/todoテーブル変更

最初にカテゴリを格納する**category**というテーブルを追加します。内容は下表のようにシンプルです。

【表1-1】category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)
name	カテゴリ名	TEXT		

categoryテーブルを作成するSQLは以下のようにします。

【リスト1-1】src/main/resources/sql/10_create_category.sql

```
DROP    TABLE    IF EXISTS category; --①
CREATE TABLE    category
(
        id              SERIAL PRIMARY KEY,
        name            TEXT
);
INSERT INTO category(name)    VALUES('仕事'); --②
INSERT INTO category(name)    VALUES('勉強');
INSERT INTO category(name)    VALUES('レジャー');
```

①categoryテーブル削除

・何回でも作り直せるようにするため、最初に削除します。

・ただし初回はテーブルが無いので、"DROP TABLE"だけではエラーになります。それを避けるのに**"IF EXISTS"**を付加して「**もしcategoryテーブルが存在するなら**DROPする」とします。

②カテゴリ追加

・idはSERIAL型なので指定不要。値は1～(自動採番)。

➡詳細は『基礎編』「6.1 テーブルの作成」参照

■実行結果

```
tododb=> SELECT * FROM category;
 id |   name
----+----------
  1 | 仕事
  2 | 勉強
  3 | レジャー
(3 行)
```

次にtodoテーブルへ、categoryのidの値を格納する**category_id**という列を追加します。これはcategoryとtodoが1:nの関係で、todoがそのn側だからです。

> ・1 つのcategoryから見ると、対応するtodoは0個以上ある(1:n)。
> ・1 つのtodoから見ると、対応するcategoryは 1 つだけ(1:1)。
> ・よってcategoryとtodoは1:nの関係にある。
> このときn側(=todo)には関連する1側(=category)の主キーの値を持たせる。
> →todoとtaskが1:nの関係で、task(=n側)がtodo(=1側)のidをtodo_idとして持つのと同じ。
>
> ➡詳細は『応用編』「14. エンティティの関連」参照

すでに存在するテーブルへ列を追加するときは、**ALTER TABLE文**を使います。

【リスト1-2】src/main/resources/sql/11_add_categoryId_to_todo.sql

```
ALTER TABLE todo ADD COLUMN category_id INTEGER; --①
UPDATE todo SET category_id=1; --②
ALTER TABLE todo ALTER COLUMN category_id SET NOT NULL; --③
```

①todoテーブルへcategory_id列(INTEGER型)追加

■形式

```
ALTER TABLE テーブル名 ADD COLUMN 追加する列名 型名;
```

②category_idの初期値設定

・【リスト1-1】でINSERTしたcategory.idの値(1～3)なら何でもよいのですが、ここでは1とし、
カテゴリ「仕事」と関連付けます。
・この初期値設定をしないと、次のNOT NULL制約が付与できません(エラーになる)。

③category_idに**NOT NULL**制約を付与

・これでcategory_idがNULLのレコードは、todoテーブルへ追加(INSERT)できなくなります。
→category_idは必須入力項目となる

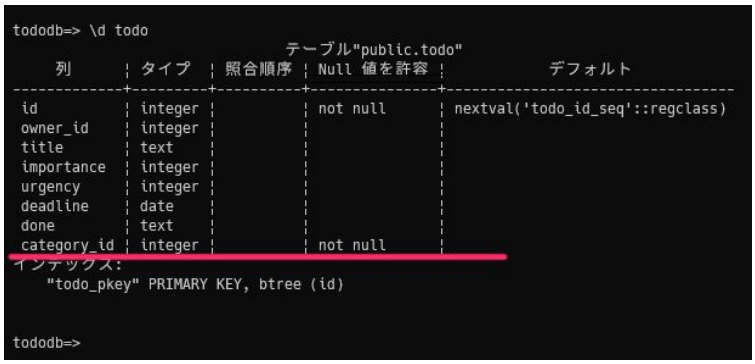
■形式

```
ALTER　TABLE テーブル名　ADD　COLUMN　付与する列名　SET　NOT NULL;
```

> これ以降、既存レコードのcategory_idをNULLへ変更することもできなくなります。
> 例. 以下のUPDATE文を実行するとエラーになる(③のNOT NULL制約に反するため)。
> 　UPDATE todo SET category_id=null;

psqlの**¥d**コマンドでtodoテーブルの定義を表示すると、category_idがNOT NULL制約付きで追加されたことを確認できます。

■実行結果



```
tododb=> \d todo
                          テーブル"public.todo"
     列      | タイプ  | 照合順序 | Null 値を許容 |            デフォルト
-------------+---------+----------+---------------+----------------------------------
 id          | integer |          | not null      | nextval('todo_id_seq'::regclass)
 owner_id    | integer |          |               |
 title       | text    |          |               |
 importance  | integer |          |               |
 urgency     | integer |          |               |
 deadline    | date    |          |               |
 done        | text    |          |               |
 category_id | integer |          | not null      |
インデックス:
    "todo_pkey" PRIMARY KEY, btree (id)


tododb=>
```

では以下のようなSELECT文で、todoとcategoryが紐付けられることを確認します。

■確認SQL

```
SELECT　todo.*,　category.*
  FROM　todo
  JOIN　category　ON　todo.category_id = category.id
  ORDER　BY　todo.id;
```

■実行結果

```
tododb=>   SELECT  todo.*,  category.*
tododb->     FROM  todo
tododb->     JOIN  category  ON  todo.category_id = category.id
tododb->    ORDER  BY  todo.id;
 id | owner_id | title  | importance | urgency |  deadline  | done | category_id | id | name
----+----------+--------+------------+---------+------------+------+-------------+----+------
  1 |        1 | todo-1 |          0 |       0 | 2023-10-01 | N    |           1 |  1 | 仕事
  2 |        1 | todo-2 |          0 |       1 | 2023-10-02 | Y    |           1 |  1 | 仕事
  3 |        1 | todo-3 |          1 |       0 | 2023-10-03 | N    |           1 |  1 | 仕事
  4 |        1 | todo-4 |          1 |       1 | 2023-10-04 | Y    |           1 |  1 | 仕事
(4 行)


tododb=>
```

このようにSQLレベルでは確認できたので、次はエンティティで関連付けられるようにします。

> 列を追加するのではなく「テーブル削除(DROP)→列を追加した定義で再作成(CREATE TABLE)」する方法もあります。しかし登録されていたデータも失われるため、何かと不便です。そこで実務では、上記のようにALTER TABLEで列追加することもよくあります。

1.4 Categoryエンティティ追加/Todoエンティティ変更

categoryテーブルに対応する**Category**エンティティを追加します。

> ➡エンティティについては『基礎編』「6.3 エンティティ」参照

【リスト1-3】com.example.todolist.entity.Category.java

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

import java.util.ArrayList;
import java.util.List;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OrderBy;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Entity
@Table(name = "category")
@Data
@ToString(exclude = "todoList")
@NoArgsConstructor
public class Category {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
	private Integer id;
```

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

    @OneToMany(mappedBy = "category")   // ①
    @OrderBy("id asc")
    private List<Todo> todoList = new ArrayList<>();

    public Category(Integer id) {   // ②
     this.id = id;
    }

    public Category(Integer id, String name) {   // ②
     this.id = id;
     this.name = name;
    }
}
```

①Todoエンティティとの関連付け

・CategoryとTodoが1:nの関係で、このCategoryが1側になることを**@OneToMany**で表します(後述)。

②コントローラークラスなどで使用するコンストラクター(後述)

あわせてリポジトリも追加します。

【リスト1-4】com.example.todolist.repository.CategoryRepository.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.Category;

@Repository
public interface CategoryRepository extends JpaRepository<Category, Integer> {
    List<Category> findAllByOrderById();   // ①
```

```
}
```

①検索用メソッドの宣言

・categoryテーブルから全レコード取得し、idの昇順で並べ替えた結果をListにして返すメソッドの宣言

→これもコントローラークラスで使います。

> ➡メソッド名と検索機能の関係については『基礎編』「9. 入力された条件で検索する」参照

もう一方のTodoエンティティにも、【リスト1-2】のcategory_idに対応するプロパティを追加します。

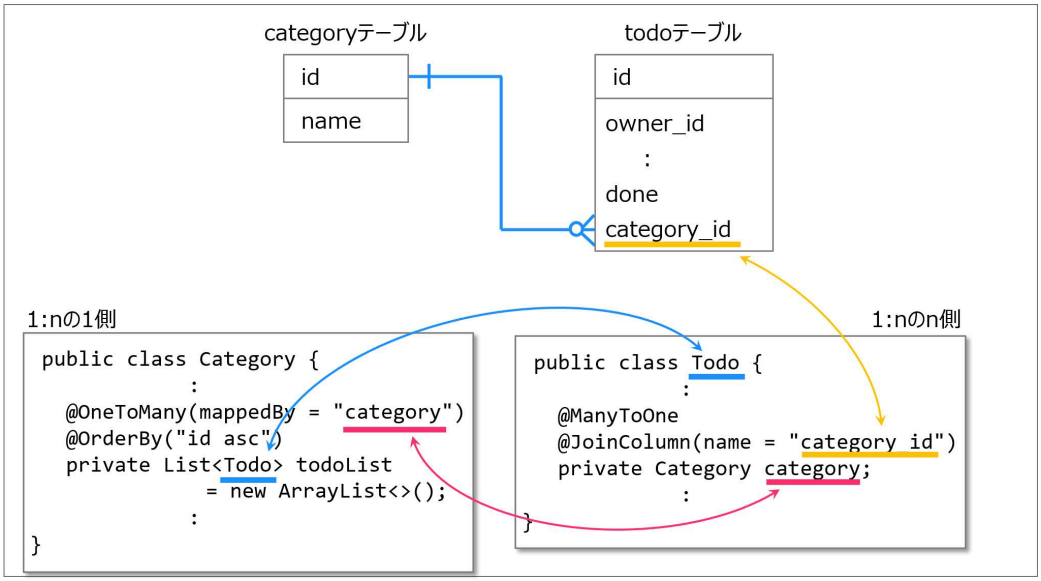
【リスト1-5】com.example.todolist.entity.Todo.java

```
                    :
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
                    :
public class Todo {
                    :
    private Integer ownerId;

    //  ----- 追加ここから ↓ ↓ ↓ -----
    @ManyToOne   // ①
  @JoinColumn(name = "category_id")
    private Category category;
    //  ----- 追加ここまで ↑ ↑ ↑ -----

    @OneToMany(mappedBy = "todo", cascade = CascadeType.ALL)
                    :
}
```

①Categoryエンティティとの関連付け

・CategoryとTodoが1:nの関係で、このTodoがn側になることを**@ManyToOne**で表します。

このようにTodoとCategoryはアノテーションで関連付けています。下図はその部分を抜き出したものでです。



【図1-4】Todo/Categoryの関連付け定義

■Categoryエンティティ

1)**@OneToMany**アノテーションで、1:nの1側であることを表している。

2)mappedBy属性には、このCategoryを参照するTodo側のプロパティ名を指定する。

→cascade属性は指定しない(CategoryからTodoへ波及させる操作を想定していないため)

3)List<Todo>により、Todoエンティティを持てるようにする(0個以上)。

→todoListに対応する列はcategoryテーブル上に無いので@columnは不要

■Todoエンティティ

1)**@ManyToOne**アノテーションで、1:nのn側であることを表している。

2)**@JoinColumn**のname属性は、関連するCategoryがcategory_id列の値で決まることを表している。

→このように外部キーを持つ側を「**所有者側**」、反対側を「**被所有者側**」と呼ぶことがある。

→3章 【図3-6】の解説参照

3)関連付けられたカテゴリの内容は、categoryプロパティに格納される。

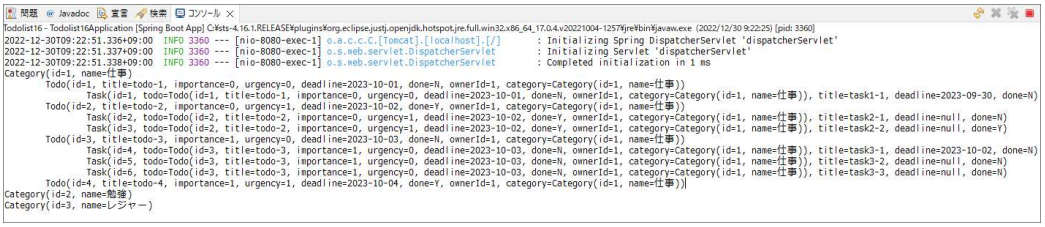
これでTodo～Categoryの関連付けは完了です。以下のようなコードをTodoListControllerへ追加すると、ログイン後、STSのコンソールに関連付けされた内容が表示されます。

【リスト1-6】com.example.todolist.controller.TodoListController.java

```
                :
import com.example.todolist.entity.Category;
import com.example.todolist.repository.CategoryRepository;
                :
public class TodoListController {
                :
    private final CategoryRepository categoryRepository;
                :
    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                :
        mv.addObject("todoList", todoPage.getContent()); // 検索結果

        // ---------- 追加ここから ↓ ↓ ↓ ----------
        List<Category> _categoryList = categoryRepository.findAllByOrderById();
        List<Todo> todoList;
        List<Task> taskList;
        for (Category category : _categoryList) {
            System.out.println(category);

            todoList = category.getTodoList();
            for (Todo todo : todoList) {
                System.out.println("¥t" + todo);

                taskList = todo.getTaskList();
                for (Task task : taskList) {
                    System.out.println("¥t¥t" + task);
                }
            }
        }
        // ---------- 追加ここまで ↑ ↑ ↑ ----------
        return mv;
```

```
}
```

■実行結果(STSコンソールへの出力)　ログインID：okada, パスワード：s6rizqfkでログインした場合



このようにCategoryからTodoが参照でき、またTodoはCategoryオブジェクトを持っているのがわかります。

例.　category=Category(id=1, name=仕事)

1.5 カテゴリをToDo一覧へ表示する

このカテゴリー名をToDo一覧(todoList.html)へ表示するには、以下の行を追加します。

【リスト1-7】src/main/resources/templates/todoList.html

```
                :
<!-- 検索結果エリア -->
<table border="1">
  <tr>
    <th>id</th>
    <th th:text="#{label.category}"></th>  <!-- ①追加 -->
    <th th:text="#{label.title}"></th>
                :
  </tr>
  <tr th:each="todo:${todoList}"><!-- ③補足 -->
    <!-- id -->
    <td th:text="${todo.id}"></td>
    <!-- カテゴリ -->
    <td th:text="${todo.category.name}"></td>  <!-- ②追加 -->
                :
```

①見出し追加

・表示文字列はリソースファイルに定義します。

【リスト1-8】src/main/resources/i18n/FixedDisplayStrings_en.properties

```
label.category=Category
```

【リスト1-9】src/main/resources/i18n/FixedDisplayStrings_ja.properties

```
label.category=カテゴリ
```

②カテゴリ名表示

・カテゴリ名は**${todo.category.name}**になります。

・③のth:eachによって、todoはtodoList(=List<Todo>型)の各要素を表すのでTodo型です。このうちカテゴリは、【リスト1-5】で追加したCategory型プロパティcategoryが持っており、カテゴリ名はその中のnameプロパティです。
→上記STSへの出力結果を参照

> ➡th:eachの詳細は『基礎編』「5.2　数当てゲームでセッションを学ぶ」参照

これで一覧にカテゴリが表示されます。またtodoの値を変えると、それに応じたカテゴリ名になることも確認できます。

例. `UPDATE todo SET category_id=2;`
を実行すると、一覧のカテゴリはすべて「勉強」になる。

1.6 カテゴリを入力する

次はToDo入力画面(todoForm.html)でカテゴリを選べるようにします。そのためにcategoryテーブルの内容を、リストボックスの選択肢とします(冒頭の実行例参照)。このデータはToDo入力画面を表示する前に、コントローラーでcategoryテーブルから取得します。

【リスト1-10】com.example.todolist.controller.TodoListController.java

```
    // ToDo入力フォーム表示
    @PostMapping("/todo/create/form")
    public ModelAndView createTodo(ModelAndView mv) {
        mv.setViewName("todoForm");
        mv.addObject("todoData", new TodoData());

        // ---------- 追加 ここから ↓ ↓ ↓ ----------
        List<Category> categoryList = categoryRepository.findAllByOrderById(); // ①
        mv.addObject("categoryList", categoryList);   // ②
        // ---------- 追加 ここまで ↑ ↑ ↑ ----------

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

①categoryテーブルの内容を全件/id順に取得

・【リスト1-4】でリポジトリに宣言したメソッドを使っています。

②検索結果を入力画面へ渡す

・少々紛らわしいですが、第二引数(categoryList)が検索結果、第一引数("categoryList")はその名前です。下記todoForm.htmlに出てくる**${categoryList}は第一引数と対応**しています。

→第一引数を"xyz"などとしてもかまわないが、その場合はtodoForm.htmlでも${xyz}とする。

ToDo入力画面では、これをもとにリストボックス(<select>)を作ります。

【リスト1-11】src/main/resources/templates/todoForm.html

```
                :
```

```
■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}">
    </td>
  </tr>
  <!-- ========== カテゴリ追加 ここから ↓ ↓ ↓ ========== -->
  <!-- カテゴリ -->
  <tr>
    <th th:text="#{label.category}"></th>
    <td>
      <select name="categoryId">    <!-- ① -->
        <option th:each="c : ${categoryList}" th:value="${c.id}"
th:text="${c.name}"
          th:selected="${c.id} == *{categoryId}"></option>    <!-- ② -->
      </select>
      <div th:if="${#fields.hasErrors('categoryId')}" th:errors="*{categoryId}"
        th:errorclass="red"></div>
    </td>
  </tr>
  <!-- ========== カテゴリ追加 ここまで ↑ ↑ ↑ ========== -->
  <!-- 件名 -->
          :
```

①カテゴリの値を表す名称

・リストボックスで選択されたカテゴリ(のvalueの値)が、"categoryId"という名前でコントローラーへ送信されるようにします。

・この値を使うには、Formデータ(TodoData)へ、同名のプロパティを追加する必要があります(後述)。

②カテゴリの選択肢を作成する

・<option>タグは以下のようになっています。

【表1-2】<option>タグの内容

属性	内容
th:each="c : ${categoryList}"	・【リスト1-10】から渡されたcategoryListの要素数分<option>タグを生成する。 ・各要素は c で表す。 →categoryListはList<Category>型なのでcは Category型オブジェクト
th:value="${c.id}"	・選択されたときcategoryIdに設定し、サーバーへ送信する値 →categoryテーブルのid列の値(1,2,3..)とする
th:text="${c.name}"	・選択肢として表示する文字列 →categoryテーブルのname列の値(仕事、勉強、...)とする
th:selected="${c.id} == *{categoryId}"	・コントローラーから送られたcategoryIdと一致するものを選択状態 にする →ToDo更新、入力エラー時用

> ➡<form>～</form>内で使用するフォーム部品の詳細は『基礎編』「4.Thymeleafでフォーム操作」参照

【図1-2】のように入力されたデータは、TodoDataオブジェクト経由でコントローラーが受け取ります。ここにカテゴリ用プロパティも追加します。

【リスト1-12】com.example.todolist.form.TodoData.java

```
public class TodoData {
                    :
    private String done;
```

```
    @Min(value = 1)    // ②追加
      private Integer categoryId;    // ①追加

      @Valid
      private List<TaskData> taskList;
                        :
      //   ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
      public Todo toEntity() {
                        :
            todo.setDone(done);
            todo.setCategory(new Category(categoryId));    // ③追加(選択されたカテゴリ)
                        :
      }

      // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
      public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
            // Todo部分
                        :
            this.done = todo.getDone();
            this.categoryId = todo.getCategory().getId();    // ④追加(現在のカテゴリ)
                        :
      }
       :
}
```

①カテゴリの選択値

・【リスト1-11】と同じ名前(**categoryId**)にすることで、ハンドラーメソッドの**@ModelAttribute**により送信されたデータの中からカテゴリ(のid値)がここへ設定されます。

②バリデーション

・categoryIdが1を下回るなら、バリデーションエラーとします。

・【リスト1-1】のように、category.idは1以上です。よって1未満は「カテゴリを選択しなかった場合」です。

【リスト1-13】src/main/resources/i18n/ValidationMessages_en.properties

```
#Category
Min.todoData.categoryId=Select the category.
```

【リスト1-14】src/main/resources/i18n/ValidationMessages_ja.properties

```
#カテゴリ
Min.todoData.categoryId=カテゴリを選択してください
```

③TodoDataからTodoオブジェクトを作成

・【リスト1-5】【図1-4】のように、Todoではカテゴリをオブジェクトで表します。そのためcategoryIdを引数にして、Categoryのコンストラクターを呼び出します。

→todoテーブルにはcategory_idしかないので、カテゴリ名は不要

④入力画面へ表示するデータ作成

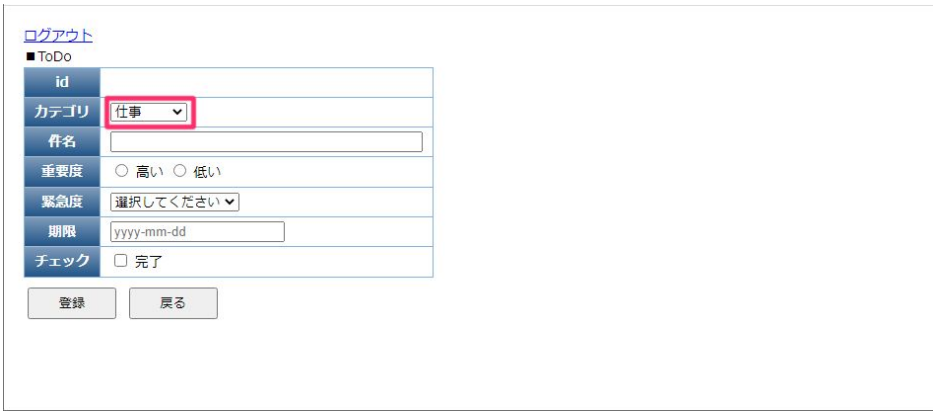
・③とは反対にTodoのCategoryオブジェクトからidを取得し、これをcategoryIdへセットします。

→この値が【リスト1-11】【表1-2】の*{categoryId}として使われます。

1.7 アプリケーションスコープ

これでカテゴリを選択・表示できるようになりました。しかし以下のような問題があります。

【問題１】「仕事」が初期値(デフォルト値)になってしまう。



【図1-5】「仕事」が初期値(デフォルト値)になってしまう

【解決方法】

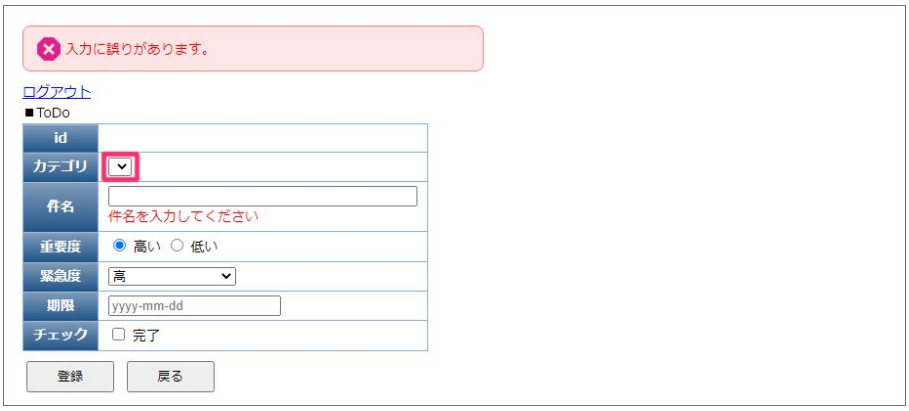
categoryテーブルの検索結果に、デフォルトの選択肢を追加します。

【リスト1-15】com.example.todolist.controller.TodoListController.java

```
                                :
    // ---------- 追加 ここから ↓ ↓ ↓ ----------
    List<Category> categoryList = categoryRepository.findAllByOrderById();
    categoryList.add(0, new Category(0, "---------"));    // デフォルトの選択肢を追加
    mv.addObject("categoryList", categoryList);
    // ---------- 追加 ここまで ↑ ↑ ↑ ----------
                                :
```

コンストラクターに渡している0は、category.idには存在しない値です(【リスト1-1】実行結果参照)。これをcategoryListの先頭要素となるようadd()で追加します(add()の引数0は先頭に追加するの意)。もしユーザーがデフォルトのままとしたらcategoryId=0なので、【リスト1-12】の@Min(value = 1)によりバリデーションエラーとなります。

【問題２】入力エラーのとき、カテゴリの選択肢が消えてしまう。



【図1-6】入力エラーのとき、カテゴリの選択肢が消えてしまう。

【問題３】ToDo一覧で件名をクリックして遷移したとき、カテゴリが消えている。選択肢も設定されていない。

【図1-7】ToDo一覧で件名をクリックして遷移したとき、カテゴリが消えている

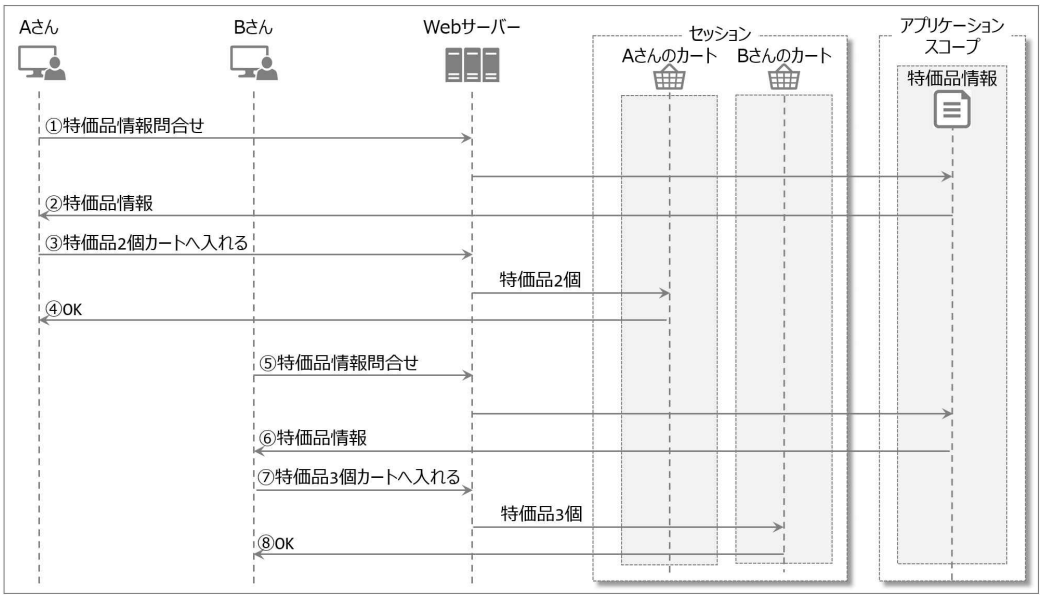
【解決方法】

この２つの原因は同じです。カテゴリの取得・選択肢設定は、【リスト1-10】[新規追加]クリック時のcreateTodo()にしかありません。これ以外の遷移では選択肢に何もセットされません。

解決策としては同じ処理を追加すればいいのですが、同じコードが点在するのは避けたいところです。またcategoryの内容はほとんど変わらないでしょうから、頻繁に検索するのも無駄です。

そこでログインした直後に 1 回だけ取得することにします。保存先はセッションにもできますが、本章のカテゴリはシステムの全ユーザーで共通です。システムで 1 つだけ持ち、全ユーザーで共有した方がよいでしょう。こういった場合、セッションではなく「**アプリケーションスコープ**」が適しています。

セッションスコープは「ブラウザ毎」の専用領域でしたが、「アプリケーションスコープ」はそれより広く、**すべてのブラウザから参照できます**。イメージとしては下図のようになります。



【図1-8】アプリケーションスコープ

セッションにある「Aさんのカート」「Bさんのカート」は、「Aさん」「Bさん」のブラウザからのみアクセスできます。それに対してアプリケーションスコープにある「特価品情報」は、どちらからでもアクセスできます。このように**システムで共有したいデータの格納先には、アプリケーションスコープが適しています**。

なおアプリケーションスコープのデータは、Webシステム(ToDoアプリ)を停止するまで存在し続けます。セッションのように「タイムアウト」という考えはありません。

> ➡セッションについては『基礎編』「5.セッション操作」参照

ToDoアプリの場合、カテゴリをアプリケーションスコープへ格納するのは、Login成功後に実行するshowTodoList()の中がいいでしょう。

【リスト1-16】com.example.todolist.controller.TodoListController.java

```
                    :
```

```
import jakarta.servlet.ServletContext;
                    :
public class TodoListController {
                    :
    private final ServletContext application;    // ①追加
                    :
    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                    :
    // ---------- 追加 ここから ↓ ↓ ↓ ----------
    @SuppressWarnings("unchecked")
    List<Category> categoryList
                                = (List<Category>)
application.getAttribute("categoryList");
    if (categoryList == null) {    // ②
        categoryList = categoryRepository.findAll();
        categoryList.add(0, new Category(0, "---------"));
        application.setAttribute("categoryList", categoryList);
    }
    // ---------- 追加 ここまで ↑ ↑ ↑ ----------
     return mv;
    }
                    :
    // ToDo入力フォーム表示
    @PostMapping("/todo/create/form")
    public ModelAndView createTodo(ModelAndView mv) {
        mv.setViewName("todoForm");
        mv.addObject("todoData", new TodoData());

        // ---------- こちらは不要 ここから ↓ ↓ ↓ ----------
        //List<Category> categoryList =
categoryRepository.findAllByOrderById();
```

```
            //categoryList.add(0, new Category(0, "---------"));    // デフォルトの選択肢を追加
            //mv.addObject("categoryList", categoryList);
            // ---------- こちらは不要 ここまで ↑ ↑ ↑ ----------

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

①アプリケーションスコープを追加

・アプリケーションスコープはServletContext(jakarta.servlet.ServletContext)型のプロパティとして操作します。これもSpring Bootに、コンストラクターインジェクションで設定してもらいます。

②カテゴリデータの有無チェック

・アプリケーションスコープへのデータ登録は、セッションと同様にsetAttribute()を使います。ただしこの処理は、アプリケーションスコープにカテゴリが無かった時だけ実行するようにします。こうすればToDoアプリ起動後、一番最初にLogin成功したユーザーが、ToDo一覧を表示するときにだけcategoryを検索します。

→それ以降のLoginでは、アプリケーションスコープに存在するため②でfalseとなる。

> ➡コンストラクターインジェクションについては『基礎編』「6.5 コントローラー」参照

これにあわせて入力画面では、カテゴリをアプリケーションスコープから取得するように変更します。

【リスト1-17】src/main/resources/templates/todoForm.html

```
    <!-- ========== カテゴリ追加 ここから ↓ ↓ ↓ ========== -->
    <!-- カテゴリ -->
    <tr>
      <th th:text="#{label.category}"></th>
      <td>
        <select name="categoryId">    <!-- ① -->
          <!--                              application.    追加する -->
          <option th:each="c : ${application.categoryList}" th:value="${c.id}"
              th:text="${c.name}"
              th:selected="${c.id} == *{categoryId}"></option>    <!-- ② -->
```

```
            </select>
            <div th:if="${#fields.hasErrors('categoryId')}" th:errors="*{categoryId}"
                th:errorclass="red"></div>
        </td>
    </tr>
    <!-- ========== カテゴリ追加 ここまで ↑ ↑ ↑ ========== -->
```

${categoryList}を${**application**.categoryList}へ変更しています。ここで追加した"**application**"が、アプリケーションスコープを表しています。

> ほかにもToDoアプリのメインクラスであるcom.example.todolist.Todolist16Applicationで、カテゴリを取得する方法などもあります。興味がある方は調べてみてください。

2. 複合主キー

プロジェクト名	Todolist17
作成ファイル	com.example.todolist.entity.CategoryKey.java
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Category.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java com.example.todolist.repository.CategoryRepository.java src/main/resources/templates/todoForm.html src/main/resources/i18n/ValidationMessages_en.properties src/main/resources/i18n/ValidationMessages_ja.properties
SQLファイル	src/main/resources/sql/20_recreate_category.sql src/main/resources/sql/21_add_category_to_todo.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「カテゴリをロケール別にできない？」
自分「！？」
先輩「ヨセミテ支社の連中に頼まれて"Preparing for Halloween"追加したら...」
自分「...(ヨセミテ？どこだよ！)」
先輩「日本支社の画面にも表示されるんだよ！」
自分「...(つーか、なんで先輩がカテゴリ追加できんの？)」
先輩「なんとかして。頼む...」ｽﾀｺﾗｻｯｻ
自分「はぁ...(その前に先輩から操作権限剥奪っと)」

2.1 自然キーと複合主キー

先輩の要件を実現する最も簡単な方法は「categoryテーブルにロケールを表す列(たとえばlocale列)を追加し、ブラウザのロケールと一致するものを選択肢とする」ことでしょう。

【表2-1】ロケールを追加したcategory

id	name	locale
1	仕事	ja
2	勉強	ja
3	レジャー	ja
4	Leisure	en_US
5	Study	en_US
6	Job	en_US
7	Preparing for Halloween	en_US

しかし、一歩推し進めて「『勉強』に関連するToDoを検索しよう」とすると難しくなります。なぜなら「勉強」にはロケール違いで「Study」があるからです。そのため、以下のようなSELECT文になるでしょう。

```
SELECT  category.name,  todo.*
  FROM  todo
  JOIN  category  ON  (category.name = '勉強'  OR category.name =
'Study')
   AND todo.category_id = category.id;
```

これだとロケールが増えると面倒です。そこで「codeで種別を表す」としたらどうでしょう。たとえば次のような形です。

【表2-2】codeを意味付けしたcategoryテーブル

code	locale	name

10	ja	仕事
20	ja	勉強
30	ja	レジャー
30	en_US	Leisure
20	en_US	Study
10	en_US	Job
90	en_US	Preparing for Halloween

ここでは10を「仕事」、20を「勉強」という具合にcodeを「**意味付け**」しています。そして一意となるように主キーは(code, locale)のペアとします。もしフランス支社、ドイツ支社用に「勉強」というカテゴリを追加するならcodeは20です。それ以外の値ではcodeの意味が保てなくなります。

【表2-3】codeを意味付けしたcategoryテーブル(2)

code	locale	name
20	fr	Étude
20	de	Studie

このcodeのように「意味付けされたキー」を「**自然キー**(Natural key；ナチュラルキー)」と言います。また(code, locale)のように、複数項目で構成されたキーを「**複合キー**(Composite key;コンポジットキー)」と言います。

これに対しtodo/taskテーブルのidは、SERIAL型として機械的に採番(1～)されたものであり、意味を持ちません。こういった「意味付けされていないキー」は「**代理キー**(Surrogate key;サロゲートキー)」と呼ばれています。

テーブルを設計するとき「自然キー」「代理キー」のどちらが良いか?(どちらにすべきか?)は難しい問題です。それは「(システムの要件など前提条件が違うので)一概には言えない」ためです。また以下のような折衷案もあります。

【表2-4】代理キーと自然キーの折衷案

id	code	locale	name

1	10	ja	仕事
2	20	ja	勉強
3	30	ja	レジャー
4	30	en_US	Leisure
5	20	en_US	Study
6	10	en_US	Job
7	90	en_US	Preparing for Halloween
8	20	fr	Étude
9	20	de	Studie

・代理キーid - **PRIMARY KEY制約**を付与する
・自然キー(code, locale) - **UNIQE制約**を付与する

> PRIMARY KEY制約 = 一意制約 + NOT NULL制約
> UNIQUE制約 = 一意制約(→ NULL値も使用可)

Spring Bootで新規開発するなら「代理キー」(あるいは「折衷案」)が、適しているように感じます。しかし「テーブルのフォーマットを変更しないこと」といった要件のある「システム移行」では、自然キー/複合キーも必要となるでしょう。

本章では複合主キーをSpring Bootで使う方法を解説します。例としてcategoryの主キーを【表2-2】のように(id, locale)のペアへ変更していきます。

2.2　複合主キーの定義

(1) 複数列からなるPRIMARY KEY制約

まずcategoryテーブルを以下のように再作成します。

【リスト2-1】src/main/resources/sql/20_recreate_category.sql

```
DROP　TABLE　IF　EXISTS category;
CREATE TABLE category
(
　　code　　TEXT,
　　locale　　TEXT,
　　name　　TEXT,
　　PRIMARY KEY(code, locale) -- ①
);
INSERT INTO category(code, locale, name)　VALUES('10', 'ja', '仕事');
INSERT INTO category(code, locale, name)　VALUES('20', 'ja', '勉強');
INSERT INTO category(code, locale, name)　VALUES('30', 'ja', 'レジャー');
INSERT INTO category(code, locale, name)　VALUES('10', 'en_US', 'Job');
INSERT INTO category(code, locale, name)　VALUES('20', 'en_US', 'Study');
INSERT INTO category(code, locale, name)　VALUES('30', 'en_US', 'Leisure');
```

①主キーの定義

・ここまでに作成したテーブルの主キーは、すべて"id SERIAL PRIMARY KEY"でした。これはidという列を定義し、同時にPRIMARY KEY制約を付与する書き方です。一方このcategoryテーブルでは、主キーとするcodeとlocaleを定義してから「**codeとlocaleの組み合わせがPRIMARY KEYである**」という書き方をしています。

　→前者を「**列制約**」(=特定の列に対するもの)、後者を「**表制約**」(=列のグループに対するもの)と言います。

次にtodoテーブルを変更し、codeとlocaleでcategoryテーブルと関連付けます。

【リスト2-2】src/main/resources/sql/21_add_category_to_todo.sql

```
ALTER TABLE todo DROP COLUMN category_id;  -- ①

ALTER TABLE todo ADD COLUMN category_code TEXT;  -- ②
ALTER TABLE todo ADD COLUMN category_locale TEXT;

UPDATE todo SET category_code='10';  -- ③
UPDATE todo SET category_locale='ja';

ALTER TABLE todo ALTER COLUMN category_code SET NOT
NULL;  -- ④
ALTER TABLE todo ALTER COLUMN category_locale SET NOT
NULL;
```

①todoテーブルからcategory_id列を削除

■形式

```
ALTER TABLE テーブル名 DROP COLUMN 削除する列名;
```

②category_code, category_locale列を追加

③仮の値を設定

④category_code, category_locale列にNOT NULL制約を付与
→必須項目とする。

■確認用SQL

```
SELECT todo.*, category.name
  FROM todo
  JOIN category ON todo.category_code = category.code
   AND todo.category_locale = category.locale
ORDER BY todo.id;
```

■実行例

```
tododb=>   SELECT  todo.*, category.name
tododb->     FROM  todo
tododb->     JOIN  category  ON  todo.category_code = category.code
tododb->      AND  todo.category_locale = category.locale
tododb->    ORDER  BY  todo.id;
 id | owner_id | title  | importance | urgency |  deadline  | done | category_code | category_locale | name
----+----------+--------+------------+---------+------------+------+---------------+-----------------+------
  1 |        1 | todo-1 |          0 |       0 | 2023-10-01 | N    | 10            | ja              | 仕事
  2 |        1 | todo-2 |          0 |       1 | 2023-10-02 | Y    | 10            | ja              | 仕事
  3 |        1 | todo-3 |          1 |       0 | 2023-10-03 | N    | 10            | ja              | 仕事
  4 |        1 | todo-4 |          1 |       1 | 2023-10-04 | Y    | 10            | ja              | 仕事
(4 行)


tododb=>
```

これでtodoとcategoryを(categoryの複合主キー)使って結合できることが確認できました。これをプログラムコードへ反映していきます。

(2) 複合主キークラスの定義

プログラム側は、まずcategoryテーブルの複合主キーに対応するクラスを作ります。
ここまでの主キーは、以下のようなInteger型プロパティとして定義していました。

```
    @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;
```

しかし複合(主)キーは、それを表す専用クラスが必要です。作成方法には
・**@Embeddable**を使う
・**@IdClass**を使う
の 2 通りありますが、本書では前者を使います。

> @Embeddableと@IdClassを比べると、前者はアクセスするときの記述量が多くなりますが、必要な定義を1箇所にまとめられる、という利点があります。興味がある方は、両者の違いについて調べてみてください。

以下がcategoryテーブルの複合主キー(code, locale)を表すクラスです。

【リスト2-3】com.example.todolist.entity.CategoryKey.java

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

import java.io.Serializable;
import jakarta.persistence.Column;
import jakarta.persistence.Embeddable;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Embeddable    // ①
@Data    // ②-b
@NoArgsConstructor    // ②-c
@AllArgsConstructor
public class CategoryKey implements Serializable{    // ②-a
	private static final long serialVersionUID = 1L;    // ②-a

	@Column(name = "code")    // ③
	private String code;

	@Column(name = "locale")    // ③
	private String locale;
}
```

①他のエンティティに埋め込み可能(=embeddable)であることを表す(**@Embeddable**)

・このクラスは後述Categoryの中に埋め込んで使います。こういったクラスには @Embeddableを付与します。

②①の関連定義

@Embeddableは以下を要求するので、それと合致するようにします。

a. java.io.Serializableを実装すること。

→serialVersionUIDを定義するだけでよい(②-a)。

b. hashCode(), equals()を定義すること。

→主キーの一意性を保証するため。Lombokの**@Data**で自動生成(②-b)

c. 引数無しコンストラクターを持つこと。

→無いと実行時エラーになる。Lombokの**@NoArgsConstructor**で自動生成(②-c)

③キーを構成するプロパティ定義

→対応するテーブルの列名を@Columnで指定するのはidの場合と同じ。

(3) 複合主キーを持つエンティティ

Categoryエンティティに、従来の主キーidに替わって複合主キーCategoryKey(【リスト2-3】)を埋め込みます。

【リスト2-4】com.example.todolist.entity.Category.java

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

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import jakarta.persistence.Column;
import jakarta.persistence.EmbeddedId;
import jakarta.persistence.Entity;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OrderBy;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Entity
@Table(name = "category")
@Data
```

```
@ToString(exclude = "todoList")
@NoArgsConstructor
public class Category implements Serializable {    // ①
    private static final long serialVersionUID = 1L;// ①

    //@Id
    //@GeneratedValue(strategy = GenerationType.IDENTITY)
    //@Column(name = "id")
    //private Integer id;
    //          ↓
    @EmbeddedId // ②
    private CategoryKey pkey;    // ③

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

    @OneToMany(mappedBy = "category")
    @OrderBy("id asc")
    private List<Todo> todoList = new ArrayList<>();

    //public Category(Integer id) {
    //      this.id = id;
    //}
    //          ↓
    public Category(String code, String locale, String name) {    // ④
        this(code, locale);
        this.name = name;
    }

    //public Category(Integer id, String name) {
    //      this.id = id;
    //      this.name = name;
```

```
    //}
    //          ↓
    public Category(String code, String locale) {    // ④
        this.pkey = new CategoryKey(code, locale);
        this.name = "";
    }
}
```

①埋め込む側もSerializableを実装
②主キーが「埋め込み型の複合主キー」であることを指定(**@EmbeddedId**)
③複合主キークラス(CategoryKey)
④主キーの型が変わった(Integer → CategoryKey)ため変更。

さらにリポジトリを変更します。これも主キーがidから複合主キーに替えたためです。

【リスト2-5】com.example.todolist.repository.CategoryRepository

```
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.Category;
import com.example.todolist.entity.CategoryKey;

@Repository
public interface CategoryRepository extends JpaRepository<Category,
CategoryKey> {    // ①
    //List<Category> findAllByOrderById(); // 削除

    // localeで検索 -> codeの昇順(小->大)
    public List<Category> findByPkey_localeOrderByPkey_code(String
locale);    // ②

    // code,localeで検索 -> 高々 1 件なのでOrderBy不要
```

```
    public Category findByPkey(CategoryKey categoryKey);    // ③
}
```

①主キーのデータ型を変更(Integer→CategoryKey)
②ロケールによる検索メソッド
・SELECT * FROM category WHERE locale='ja' ORDER BY code に相当するメソッドを宣言します。
・メソッド名に含まれる"_"は、以下のような構造を表しています。
Pkey_locale →「Pkeyのlocale」, Pkey_code →「Pkeyのcode」
→PkeyはCategoryエンティティのpkey(【リスト2-4】③)を表しておりCategoryKey型
テーブルの列に対応するのは、その中のcode, locale(これらに@columnが付与されている)。
③主キーによる検索メソッド
・引数がCategoryKey型であることに注意。

次にこのCategoryと1:nの関係を持つTodoを変更します。

【リスト2-6】com.example.todolist.entity.Todo.java

```
                    :
import jakarta.persistence.JoinColumns;
                    :
public class Todo {
                    :
    //@ManyToOne
    //@JoinColumn(name = "category_id")
    //private Category category;
    //          ↓
    @ManyToOne
    @JoinColumns ({    // ①
        @JoinColumn(name="category_code", referencedColumnName = "code"),    // ②
        @JoinColumn(name="category_locale", referencedColumnName = "locale"),    // ②
```

```
        })
        private Category category;
                        :
}
```

①結合項目が複数あることの指定(**@JoinColumns**)

・結合項目(**@JoinColumn**)が複数ある場合は、それら全体を@JoinColumnsで囲みます。

②結合先テーブル列名の指定

・@JoinColumnを@JoinColumns内に書くときは、name属性とreferencedColumnName属性の両方を指定します。

name - 結合元の列(todo側)

referencedColumnName - nameの結合先列(category側)

(4) カテゴリの表示・入力

これで複合主キーを使ってTodoとCategoryを関連付けできました。ここから表示するための変更を加えていきます。まずカテゴリを取得する部分です。

【リスト2-7】com.example.todolist.controller.TodoListController.java

```
                        :
import org.springframework.context.i18n.LocaleContextHolder;
                        :
        //private final ServletContext application;    // 不要
                        :
        // ToDo一覧表示
        @GetMapping("/todo")
        public ModelAndView showTodoList(ModelAndView mv,
                        :
            // カテゴリ取得
            //@SuppressWarnings("unchecked")
            //List<Category> categoryList
```

```
            //                  = (List<Category>)
application.getAttribute("categoryList");
            //if (categoryList == null) {
            //      categoryList = categoryRepository.findAll();
            //      categoryList.add(0, new Category(0, "---------"));
            //      application.setAttribute("categoryList", categoryList);
            //}
            //          ↓
            @SuppressWarnings("unchecked")
            List<Category> categoryList
                              =
(List<Category>)session.getAttribute("categoryList");
            if (categoryList == null) {
                  String locale = LocaleContextHolder.getLocale().toString(); //
①
                  categoryList =
categoryRepository.findByPkey_localeOrderByPkey_code(locale);
                  categoryList.add(0, new Category("", locale, "---------"));
                  session.setAttribute("categoryList",categoryList);    //②
            }

            return mv;
      }
      :
```

①ブラウザのロケール取得

・【リスト2-5】で宣言したfindByPkey_localeOrderByPkey_code()の第一引数には、ブラウザのロケールを渡しますが、ここでは **LocaleContextHolder**(org.springframework.context.i18n.LocaleContextHolder)経由で取得したものを使っています。これを使うとハンドラーメソッドの引数にLocaleを追加しなくてもロケールを入手できます。

> ➡Localeについて『応用編』「12.国際化対応」参照

②格納先をセッションへ変更

・**categoryListの格納先を**(アプリケーションスコープから)**セッションへ変更**します。これはもしToDoアプリ起動後最初にLoginしたユーザーがフランス支社の人だったら、locale='fr'のカテゴリが選択され、それを全ユーザーで共有してしまうためです。つまり日本支社の人の画面にも、フランス語のカテゴリが表示されます。これを避けるため、**カテゴリは操作者のロケールに合わせて検索し、セッションへ格納する**ようにします。

これを画面に反映させます。

【リスト2-8】src/main/resources/templates/todoForm.html

```
                    :
■ToDo
<!-- ToDo入力エリア -->
<table>
  <!-- id -->
                    :
  <!-- カテゴリ -->
  <tr>
    <th th:text="#{label.category}"></th>
    <td>
<!--- 変更前 ここから↓ ↓ ↓
==========================================================
===========
      <select name="categoryId">
        <option th:each="c : ${application.categoryList}" th:value="${c.id}"
                th:text="${c.name}"
                th:selected="${c.id} == *{categoryId}"></option>
      </select>
      <div th:if="${#fields.hasErrors('categoryId')}" th:errors="*{categoryId}"
           th:errorclass="red"></div>
ここまで ↑ ↑ ↑
==========================================================
================== --->
```

```
<!--- 変更後 ここから↓ ↓ ↓
================================================================
======= --->
        <select name="categoryCode">
          <option th:each="c : ${session.categoryList}"
th:value="${c.pkey.code}"
                  th:text="${c.name}"
                  th:selected="${c.pkey.code} == *{categoryCode}">
</option><!--①②-->
        </select>
        <div th:if="${#fields.hasErrors('categoryCode')}" th:errors="*
{categoryCode}"
             th:errorclass="red"></div>
<!--- 変更後 ここまで↑ ↑ ↑
================================================================
======= --->
    </td>
              :
```

①カテゴリの取得先変更

・${**application**.categoryList}→${**session**.categoryList}とし、セッションからカテゴリを取得します。

②主キー変更

・主キーが${c.**id**}→${c.**pkey.code**}のように変化しています。これは「Categoryオブジェクトの中のPkeyオブジェクトの中のcode」というクラス階層に合わせた変更です。

最後にTodoDataを以下のように変更します。

【リスト2-9】com.example.todolist.form.TodoData.java

```
                  :
import org.springframework.context.i18n.LocaleContextHolder;
                  :
public class TodoData {
                  :
```

```
    @Min(value = 1)
    //private Integer categorId;
    //          ↓
    private String categoryCode;    // ①
                    :
    //   ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
        todo.setDone(done);
        //todo.setCategory(new Category(categoryId));
        //          ↓
        String locale = LocaleContextHolder.getLocale().toString();
        todo.setCategory(new Category(categoryCode, locale));    // ②選択されたカテゴリ
                    :
    }

    // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
    public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
                    :
        this.done = todo.getDone();
        //this.categoryId = todo.getCategory().getId();
        //          ↓
        this.categoryCode = todo.getCategory().getPkey().getCode();    // ③現在のカテゴリ
                    :
    }
     :
}
```

①カテゴリの型変更

・【リスト2-8】のように、ToDo入力画面ではidに代わりcodeを選択するので、それに合わせて変更します。

②Categoryのコンストラクター変更

・ここもLocaleContextHolder経由でロケールを取得しています。

③画面表示するcodeを渡す

・「Categoryオブジェクトの中のPkeyオブジェクトの中のcode」という具合に１レベル階層が深くなっています。

さらにFormデータのプロパティ名を変えたので(categoryId→categoryCode)、これに合わせてバリデーションメッセージの定義を変更します。

【リスト2-10】src/main/resources/i18n/ValidationMessages_en.properties

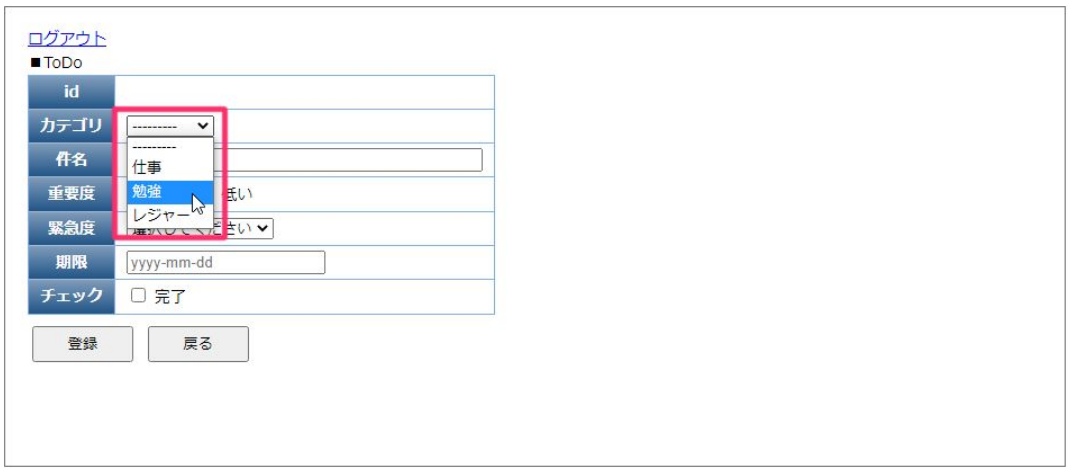
```
#Category
#Min.todoData.categoryId=Select the category.
#    ↓
Min.todoData.categoryCode=Select the category.
```

【リスト2-11】src/main/resources/i18n/ValidationMessages_ja.properties

```
#カテゴリ
#Min.todoData.categoryId=カテゴリを選択してください
#    ↓
Min.todoData.categoryCode=カテゴリを選択してください
```

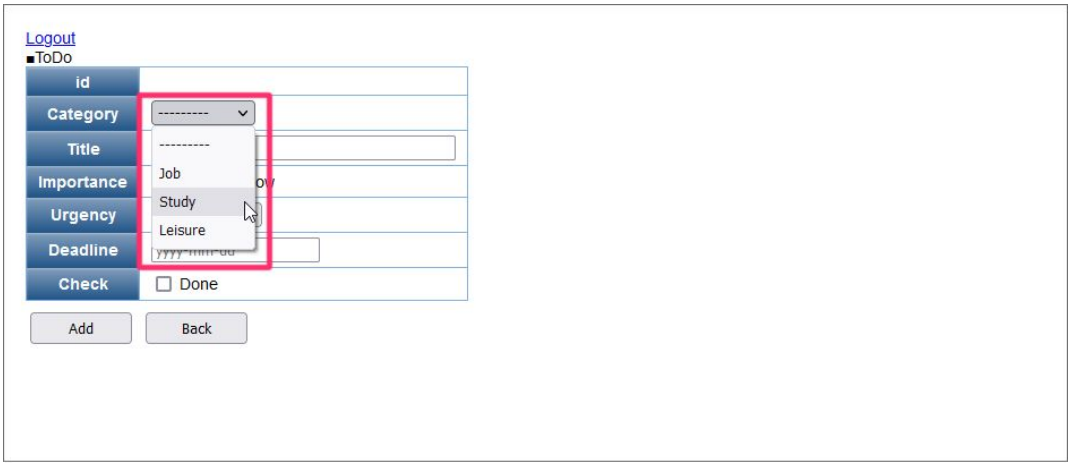
categoryの複合主キー化に伴う修正は以上です。

実行するとロケールがjaの場合は、何も変わっていません(前章と同じ)。



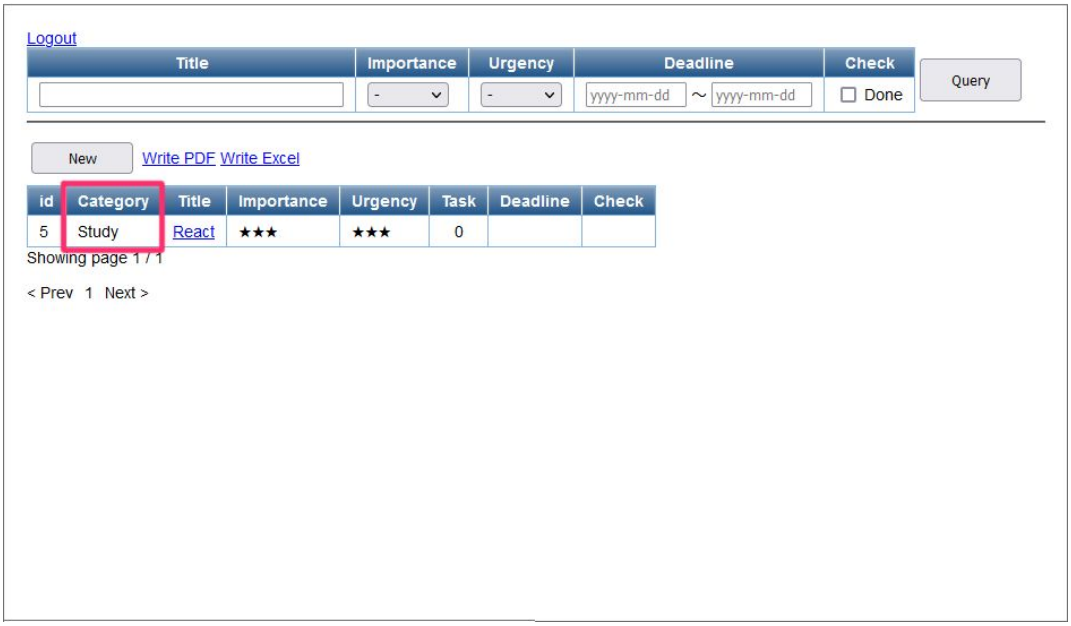
【図2-1】ロケール="ja"の場合(変化なし)

以下はロケールを"en_US"にしたFirefoxの場合です。カテゴリの選択肢が変わっていることを確認できます。



【図2-2】ロケール="en_US"の場合(1)

ToDoを登録すると、一覧にもそのロケールのカテゴリが表示できるようになります。



【図2-3】ロケール="en_US"の場合(2)

3. 多対多(n:n)の関連

プロジェクト名	Todolist18
作成ファイル	com.example.todolist.entity.Groups.java com.example.todolist.entity.GroupsAccount.java com.example.todolist.repository.GroupsRepository.java com.example.todolist.repository.GroupsAccountRepository.java
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.dao.TodoDaoImpl.java com.example.todolist.entity.Account.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java src/main/resources/templates/todoForm.html src/main/resources/templates/todoList.html src/main/resources/i18n/FixedDisplayStrings_en.properties src/main/resources/i18n/FixedDisplayStrings_ja.properties src/main/resources/i18n/OperationMessages_en.properties src/main/resources/i18n/OperationMessages_ja.properties
SQLファイル	src/main/resources/sql/30_reate_groups.sql src/main/resources/sql/31_add_groupId_to_todo.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「全部オープンなんだよな...」
自分「？」
先輩「こっそりBBQ大会のToDo登録したら全世界の支社にバレてさ...」
自分「...(何してんの！)」
先輩「クローズドグループ作れない？」
自分「！？」
先輩「社内のインフォーマルなコミュニケーションも活性化できると思うんだよなぁ〜」
自分「...(何その大義名分...)」

先輩「よろしくね！」ｽﾀｺﾗｻｯｻ
自分「はぁ...(また項目追加するか)」
こういった「安請け合い」が実は大変で、のちのち後悔するというのは実務あるある...。

3.1 多対多の関係

先輩の要件は「ToDoを共有できるグループ」ですが、これを以下のような仕様で実装していきます。

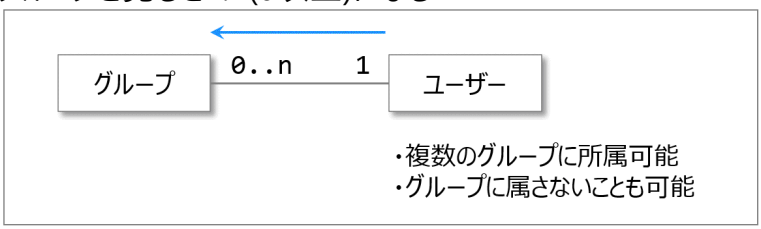
【仕様１】ユーザーは複数のグループに所属できる。
【仕様２】グループには複数のユーザーが所属できる。
【仕様３】ToDoにグループが設定されていたら、そのグループ所属のユーザーで共有できる。
【仕様４】共有したToDoは、ToDo登録者と共有グループ所属のユーザーが閲覧・編集できる。
【仕様５】ToDoの削除は登録者のみ可、タスクは登録者と共有グループ所属のユーザーのみ可とする。

> このように「簡単そうに見えても、実現するには色々なことを決めなければならない」、ということは実務でもよくあります。

さらに【仕様１】【仕様２】を分析すると、以下のようにするのが自然でしょう。

【仕様１´】１人のユーザーは、複数のグループに所属できる。またグループに所属しないこともあり得る。

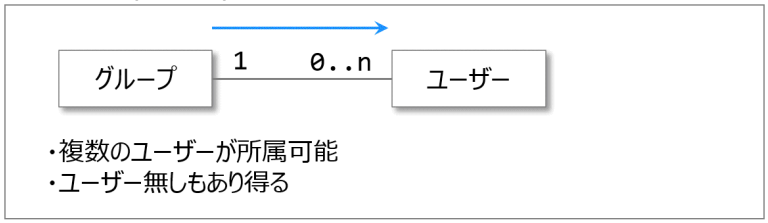
→ユーザーからグループを見ると1:n(0以上)になる



【図3-1】ユーザーからグループ(1:n)

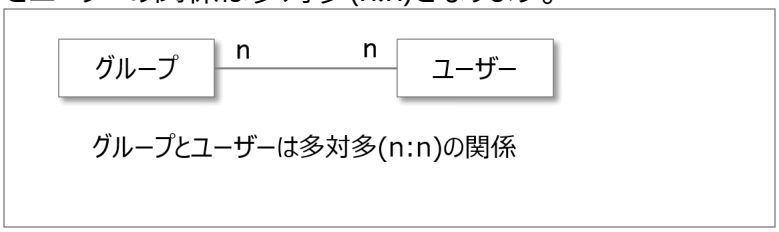
【仕様２´】１つのグループには、ユーザーが複数人所属する。またユーザーが所属していないグループもあり得る。

→グループから見ると1:n(0以上)になる。



【図3-2】グループからユーザー(1:n)

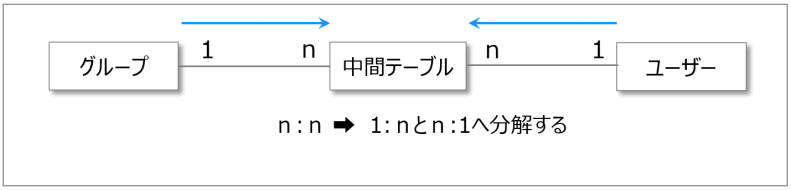
以上よりグループとユーザーの関係は多対多(n:n)となります。



【図3-3】n:nの関係

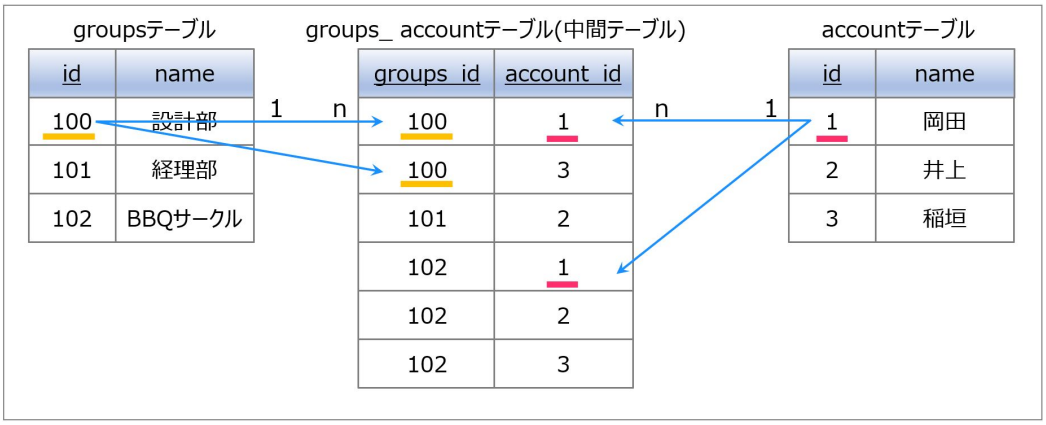
> ➡この表記法については『応用編』「14.2 多重度」参照

実はこういったn:nの関連は、正規化の観点からすると、それらのテーブルだけでは表せません。一般的には「**中間テーブル**」**を導入**し、1:n, n:1へ分解します。



【図3-4】中間テーブルの導入

では「中間テーブル」とはどういうものでしょうか？基本的には下図のように、2 つのテーブルの主キーを集めたものとします。



【図3-5】中間テーブルの導入(2)

この例では、中間テーブルを使い

・グループ「設計部」には、ユーザー「岡田」さん、「稲垣」さんが所属する
・ユーザー「岡田」さんは、グループ「設計部」「BBQサークル」に所属する
といったことを表しています。

> 中間テーブルに、キー以外の情報を持たせる場合もあります。興味がある方は調べてみてください。

これで以下のような検索が行えます。

■設計部に所属するユーザー

```
SELECT  account.*
  FROM  account
  JOIN  groups_account  ON  groups_account.account_id  =  account.id
   AND  groups_account.groups_id  =  100;
```

■岡田さんが所属するグループ

```
SELECT  groups.*
  FROM  groups
  JOIN  groups_account  ON  groups_account.groups_id  =  groups.id
   AND  groups_account.account_id = 1;
```

このように**n:n(多対多)の関係を、中間テーブルを介して1:n(１対多)同士へ分解**できます。
また「所属するユーザーがいないグループ」「グループに属していないユーザー」は、それぞれgroupsテーブル/accountテーブルにのみ存在するもの、として表せます。

3.2 中間テーブルの定義

ここからグループとユーザーをn:nの関係にしていきます。
まずグループを表すgroupsテーブルと中間テーブルgroups_accountを追加します。

【リスト3-1】src/main/resources/sql/30_create_groups.sql

```
DROP TABLE IF EXISTS    groups;
CREATE TABLE groups
(
   id          SERIAL    PRIMARY KEY,
   name        TEXT      NOT NULL
);
SELECT    SETVAL('groups_id_seq', 100, false);    -- ①
INSERT INTO groups(id, name)    VALUES(0, '');
INSERT INTO groups(name) VALUES('設計部');
INSERT INTO groups(name) VALUES('経理部');
INSERT INTO groups(name) VALUES('BBQサークル');

DROP    TABLE    IF EXISTS groups_account;
CREATE    TABLE groups_account
(
   id                  SERIAL     PRIMARY KEY,    -- ②
   groups_id        INTEGER    NOT NULL,
   account_id       INTEGER    NOT NULL,
   UNIQUE(groups_id, account_id)    -- ③
);

INSERT INTO groups_account(groups_id, account_id) VALUES (100,1);     --- ④
INSERT INTO groups_account(groups_id, account_id) VALUES (100,3);
INSERT INTO groups_account(groups_id, account_id) VALUES (101,2);
INSERT INTO groups_account(groups_id, account_id) VALUES (102,1);
INSERT INTO groups_account(groups_id, account_id) VALUES (102,2);
INSERT INTO groups_account(groups_id, account_id) VALUES (102,3);
```

①groups.idには100～が設定されるようにする

・SERIAL型の列を定義すると **テーブル名_SERIAL型の列名_seq** という名前の「**シーケンス** (sequence)」が作成されます。これはレコードを 1 行だけ持つ特別なテーブルで、この中に「いくつまで採番したか？」という情報を持っています。通常は1から採番しますが、SETVAL関数でそれを変更できます。

・引数は以下のようになっています

'groups_id_seq' - シーケンス名

100 - セットする値

false - セットした値(=100)から使い始めることを表す

これで設計部がid=100, 経理部がid=101, ...となります。

> ここで100～としたのは、単に(グループの2桁に対して)グループを3桁で表したいからです。

②主キーの定義

・(group_id, account_id)も主キーになり得ますが、その場合新たな複合主キーを表すクラスが必要になります。それを避けるため、ここでは主キーをidとしSERIAL型とします。

③UNIQUE制約

・重複した(group_id, account_id)が登録できないようUNIQUE制約を付与します。

④【図3-5】準拠のデータを登録

後述の説明で使用します。

これで以下のSELECT文を実行すると、ユーザーとグループがn:nで表せていることがわかります。

```
SELECT   a.name AS ユーザー名,   g.name AS グループ名
  FROM   groups g
  JOIN   groups_account ga   ON   g.id = ga.groups_id
  JOIN   account a   ON   ga.account_id = a.id
ORDER   BY ga.groups_id, ga.account_id;
```

■実行結果

```
tododb=>   SELECT  a.name AS ユーザー名,  g.name AS グループ名
tododb->     FROM  groups g
tododb->     JOIN  groups_account ga  ON  g.id = ga.groups_id
tododb->     JOIN  account a  ON  ga.account_id = a.id
tododb->    ORDER  BY  ga.groups_id, ga.account_id;
 ユーザー名 | グループ名
------------+------------
 岡田 是則  | 設計部
 稲垣 絵美  | 設計部
 井上 俊憲  | 経理部
 岡田 是則  | BBQサークル
 井上 俊憲  | BBQサークル
 稲垣 絵美  | BBQサークル
(6 行)


tododb=>
```

3.3 グループ-ユーザーの関連付け(@ManyToMany)

ここからn:nの関連をエンティティに定義していきます。
まず中間テーブルのエンティティを作成します。こちらは「テーブルに対応するただのエンティティ」といった感じで、特に目新しい新しい要素はありません。

【リスト3-2】com.example.todolist.entity.GroupsAccount.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 = "groups_account")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GroupsAccount {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Integer id;

	@Column(name = "groups_id")
	private Integer groupsId;

	@Column(name = "account_id")
```

```
    private Integer accountId;
}
```

【リスト3-3】com.example.todolist.repository.GroupsAccountRepository

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

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

@Repository
public interface GroupsAccountRepository extends
JpaRepository<GroupsAccount, Integer> {
}
```

次にGroupsエンティティを作成します。この中でAccountエンティティとn:nの関連であることを**@ManyToMany**で表します。

【リスト3-4】com.example.todolist.entity.Groups.java

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

import java.util.ArrayList;
import java.util.List;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OrderBy;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```
import lombok.ToString;

@Entity
@Table(name = "groups")
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString(exclude = { "accountList" })
public class Groups {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id")
	private Integer id;

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

	// こちらを被所有者側にする
	@ManyToMany(mappedBy = "groupsList")
	@OrderBy("id asc")
	private List<Account> accountList = new ArrayList<>();

	public Groups(Integer id, String name) {
		this.id = id;
		this.name = name;
	}

	public Groups(Integer id) {
		this.id = id;
	}
}
```

【リスト3-5】com.example.todolist.repository.GroupsRepository

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

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

@Repository
public interface GroupsRepository extends JpaRepository<Groups, Integer> {
}
```

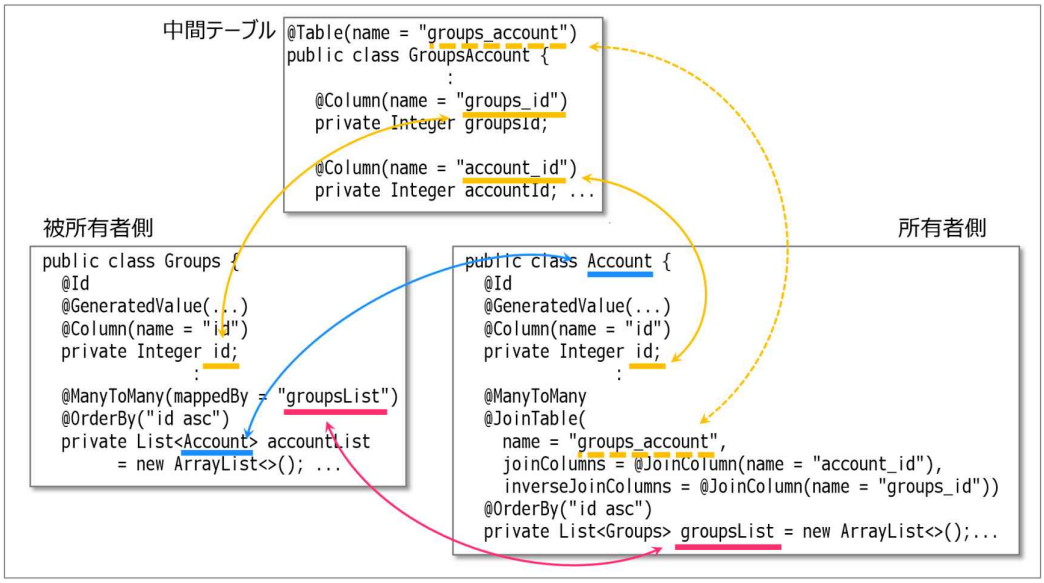
反対側のAccountエンティティにも**@ManyToMany**を追加し、Gropusとn:nであることを表します。

【リスト3-6】com.example.todolist.entity.Account.java

```
                    :
import java.util.ArrayList;
import java.util.List;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OrderBy;
import lombok.ToString
                    :
@Entity
@Table(name = "account")
@Data
//@AllArgsConstructor    // 削除
@NoArgsConstructor
@ToString(exclude = { "password", "groupsList" })    // 追加
public class Account {
                    :
    private String password;

    // ---------------- 追加ここから ↓ ↓ ↓ --------------
    // こちらを所有者側にする
  @ManyToMany
```

```
    @JoinTable(name = "groups_account",
                    joinColumns = @JoinColumn(name = "account_id"),
                    inverseJoinColumns = @JoinColumn(name = "groups_id"))
    @OrderBy("id asc")
    private List<Groups> groupsList = new ArrayList<>();

    public Account(Integer id, String loginId, String name, String password) {
        this.id = id;
        this.loginId = loginId;
        this.name = name;
        this.password = password;
    }

    public Account(Integer id) {
        this.id = id;
    }
    // ---------- 追加ここまで ↑ ↑ ↑ ----------
}
```

以下はGrpups - GroupsAccount - Accountの関連定義を抜き出したものです。



【図3-6】Grpups - GroupsAccount - Accountの関連

Groupsに「**被所有者側**」、Accountに「**所有者側**」とありますが、これはテーブルの外部キー情報をどちらが持っているか？を表しています。@OneToMany/@ManyToOneの場合は、@ManyToOne側が所有者です(Todo - Taskは、Taskが所有者で、テーブルの列todo_idを@JoinColumnで指定)。

@ManyToManyの場合は、どちらも所有者になれますが、ここではAccountとします。

@ManyToManyの所有者側エンティティには、**@JoinTable**で、中間テーブルの情報を記述します。

【表3-1】@JoinTableの属性

属性名	属性値
name	・中間テーブル名
joinColumns	・中間テーブルで所有者側の主キーを格納している列 所有者側、つまりこのAccountの主キーはaccount_id列が持っている →これを**@JoinColumn**で指定する
inverseJoinColumns	・中間テーブルで被所有者側の主キーを格納してる列 被所有者側、つまりGroupsの主キーはgroups_id列が持っている →これを**@JoinColumn**で指定する

もう一方のGroupsには、@ManyToManyのmappedBy属性で、このGroupsを参照するAccount側のプロパティを指定します(→@OneToManyと同じ考え方です)。

これでn:nの関連付けは完了です。

以下のコードをTodoListControllerへ追加後、ログイン画面からログインすると、グループ～ユーザーの関連がSTSのコンソールタブへ出力されます。

【リスト3-7】com.example.todolist.controller.TodoListController.java

```
        :
import com.example.todolist.entity.Account;
import com.example.todolist.entity.Groups;
import com.example.todolist.repository.AccountRepository;
import com.example.todolist.repository.GroupsRepository;
        :
```

```
public class TodoListController {
                    :
    private final GroupsRepository groupsRepository;      //  追加
    private final AccountRepository accountRepository;    //  追加
                    :
    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                    :
        //  ---------- 追加ここから ↓ ↓ ↓ ----------
        System.out.println("■groups -> account");
        List<Groups> groupsList = groupsRepository.findAll();
        for (Groups groups : groupsList) {
            System.out.println(groups);

            List<Account> accountList = groups.getAccountList();
            for (Account account : accountList) {
                System.out.println("¥t" + account);
            }
        }

        System.out.println("■account -> groups");
        List<Account> accountList = accountRepository.findAll();
        for (Account account : accountList) {
            System.out.println(account);

            List<Groups> groupsAccountList = account.getGroupsList();
            for (Groups groups : groupsAccountList) {
                System.out.println("¥t" + groups);
            }
        }
        //  ---------- 追加ここまで ↑ ↑ ↑ ----------

        return mv;
```

```
    }
```

■実行結果例

```
■groups -> account
Groups(id=0, name=)
Groups(id=100, name=設計部)
    Account(id=1, loginId=okada, name=岡田 是則)
    Account(id=3, loginId=inagaki, name=稲垣 絵美)
Groups(id=101, name=経理部)
    Account(id=2, loginId=inoue, name=井上 俊憲)
Groups(id=102, name=BBQサークル)
    Account(id=1, loginId=okada, name=岡田 是則)
    Account(id=2, loginId=inoue, name=井上 俊憲)
    Account(id=3, loginId=inagaki, name=稲垣 絵美)
■account -> groups
Account(id=1, loginId=okada, name=岡田 是則)
    Groups(id=100, name=設計部)
    Groups(id=102, name=BBQサークル)
Account(id=2, loginId=inoue, name=井上 俊憲)
    Groups(id=101, name=経理部)
    Groups(id=102, name=BBQサークル)
Account(id=3, loginId=inagaki, name=稲垣 絵美)
    Groups(id=100, name=設計部)
    Groups(id=102, name=BBQサークル)
```

以上の追加・変更をしても、ユーザー登録はこれまで通り行えます(Accoutテーブルには変更を加えていないため)。本来であればグループを作成する機能、およびグループとユーザーを関連付ける機能なども必要ですが、本書では省略します。興味がある方は、是非チャレンジしてみてください。

3.4 ToDo-グループの関連付け

これで最初に書いた【仕様１】【仕様２】がクリアできました。次に【仕様３】を実装します。

【仕様３】ToDoにグループが設定されていたら、そのグループで共有できる。

この「共有」を表す方法はいくつか考えられますが、ここでは以下のようにします。
【仕様3-1】todoテーブルにgroups_id列を追加する。
【仕様3-2】共有しないToDoは、groups_id=0とする。
【仕様3-3】共有するToDoは、そのグループを表すgroups.idの値をgroups_idへ設定する。

【表3-2】ToDo共有の仕様

■todoテーブル

id	...	owner_id	groups_id 【仕様3-1】	...	備考
10		1	0		account.id=1が登録者のTodo →groups_id=0なので共有しない(【仕様3-2】)
11		1	102		account.id=1が登録者のTodo groups.id=102のグループで共有する(【仕様3-3】) ↓ SELECT account_id FROM groups_account WHERE groups_id=102 の結果が共有するユーザーとなる ↓ 【図3-5】【リスト3-1】の場合、結果は 1, 2, 3なので「岡田」「井上」「稲垣」で共有する

上記仕様に従いtodoテーブルにgroups_idを追加します。これでToDo-グループを関連付けます。

【リスト3-8】src/main/resources/sql/31_add_groupId_to_todo.sql

```
ALTER TABLE todo ADD COLUMN groups_id INTEGER;
```

```
UPDATE  todo  SET groups_id  =  100;
ALTER  TABLE todo  ALTER COLUMN groups_id  SET  NOT NULL;
```

Todoエンティティにも対応するプロパティを追加し、関連するGroupsエンティティを取得できるよう@ManyToOneを付与します。

【リスト3-9】com.example.todolist.entity.Todo.java

```
public class Todo {
                    :
    private List<Task> taskList = new ArrayList<>();

    // ---------- 追加ここから ↓ ↓ ↓ ----------
    @ManyToOne
    @JoinColumn(name = "groups_id")
    private Groups groups;
    // ---------追加ここまで ↑ ↑ ↑ ----------

    // Todoへの参照設定
                    :
}
```

グループとToDoは1:nの関係です。ToDoはn側なので@ManyToOneによってグループと関連付けます。これでToDoを取得すると、同時にグループの情報も取得できます。

反対にグループからToDoを取得することは想定していないので、グループ側はそのままとします。

これでTodo→Groupsを関連付けできたのでToDo一覧へ表示できるようにします。

【リスト3-10】src/main/resources/templates/todoList.html

```
                    :
  <!-- 検索結果エリア -->
  <table border="1">
    <tr>
      <th>id</th>
      <th th:text="#{label.category}"></th>
      <th th:text="#{label.group}"></th>    <!-- 追加 -->
```

```
        <th th:text="#{label.title}"></th>
                    :
    </tr>
    <tr th:each="todo:${todoList}"><!-- ②補足 -->
        <!-- id -->
        <td th:text="${todo.id}"></td>
        <!-- カテゴリ -->
        <td th:text="${todo.category.name}"></td>
        <!-- グループ -->
        <td th:text="${todo.groups.name}"></td>  <!-- ①追加 -->
        <!-- 件名 -->
                    :
```

①ToDoのグループ名

・②のth:eachよりtodoはtodoList(=List<Todo>型)の各要素なのでTodo型です。グループ名は【リスト3-9】で追加したGroups型プロパティgroupsが持っており、カテゴリ名はその中のnameプロパティです。

【リスト3-11】src/main/resources/i18n/FixedDisplayStrings_en.properties

```
label.group=Group
```

【リスト3-12】src/main/resources/i18n/FixedDisplayStrings_ja.properties

```
label.group=グループ
```

以上でToDo一覧にグループ名が表示できるようになります。


ログアウト
件名
重要度
緊急度
期限
チェック
検索
yyyy-mm-dd
yyyy-mm-dd
完了
新規追加
PDF出力 Excel出力
id
カテゴリ
グループ
件名
重要度
緊急度
タスク
期限
チェック
1
仕事
UI設計
★★★
★★★
0
2023-10-01
2
レジャー
BBQサークル
定例BBQ大会開催
★★★
★
2
2023-10-02
1 / 1 ページを表示中
←前 1 次→

【図3-7】グループの表示

3.5 グループの入力

今度はToDo入力・編集時、グループを指定できるようにします。**選択肢は「そのユーザーが属しているグループ」**であり、これは【リスト3-6】で追加したAccount.groupsListが持っています。

「ユーザーが所属するグループ」も、前章のカテゴリ同様**ユーザーごとのデータ**です。またグループの情報は、ToDoアプリを使っている間(=セッション中)は変更しないことにすれば、取得タイミングはLogin成功後の1回で十分です。また格納先も**セッション**がよいでしょう。

【リスト3-13】com.example.todolist.controller.TodoListController

```
      :
import java.util.ArrayList;
      :
      // ToDo一覧表示
      @GetMapping("/todo")
      public ModelAndView showTodoList(ModelAndView mv,
                        :
            // ---------- 追加ここから ↓ ↓ ↓ ----------
            // 所属グループ取得
            @SuppressWarnings("unchecked")
            List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
            if (groupsList == null) {
                  Account myAccount = accountRepository.findById(accountId).get();
// ①
                  groupsList = new ArrayList<>();
                  groupsList.addAll(myAccount.getGroupsList());    // ②
                  session.setAttribute("groupsList", groupsList);
            }
            // ---------- 追加ここまで ↑ ↑ ↑ ----------

            return mv;
      }
```

①ユーザーの情報を取得する

・セッションに格納しているaccountIdをキーにして、このユーザーのAccountエンティティを取得する。このとき関連付けられたGroupsエンティティが、Account.groupsListへセットされます。

→get()はfindById()の戻り値がOptional<Account>のため必要。
セッションにaccountIdがあるということは、Loginに成功している。よってこのfindById()はnullにならない(=accountに該当レコードが存在する)という前提で直接get()を実行している。

②List#addAll()は、引数をすべて追加するメソッド

これをToDo入力画面(todoForm.html)では以下のように表示します。

【リスト3-14】src/main/resources/templates/todoForm.html

```
                    :
■ToDo
<!-- ToDo入力エリア -->
<table>
                    :
  <!-- カテゴリ -->
                    :
  <!-- ========== 追加ここから ↓ ↓ ↓
========================================================== -->
  <!-- グループ -->
  <tr>
    <th th:text="#{label.group}"></th>
    <td>
      <select name="groupsId">
        <option value="0">----------</option>    <!-- ① -->
        <option th:each="g : ${session.groupsList}" th:value="${g.id}"
                th:text="${g.name}" th:selected="${g.id} == *{groupsId}">
</option>
      </select>
    </td>
  </tr>
```

```
  <!-- ========== 追加ここまで ↑ ↑ ↑
================================================== -->
  <!-- 件名 -->
                    :
```

①グループを選択しない場合用に追加する。

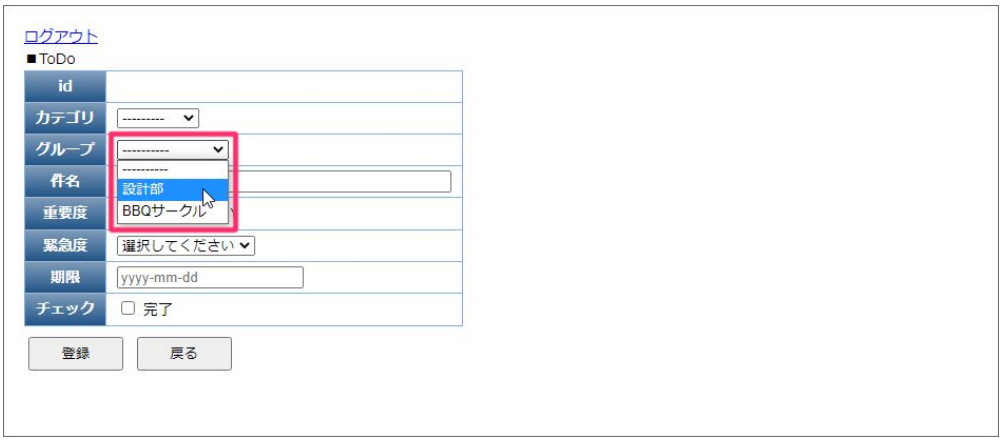
あとはこのgroupsIdをTodoDataへ定義します。これでTodoにグループを設定できるようになります。

【リスト3-15】com.example.todolist.form.TodoData.java

```
    :
import com.example.todolist.entity.Groups;
    :
public class TodoData {
                    :
    private List<TaskData> taskList;

    private Integer groupsId;    // ①追加

    private TaskData newTask;
                    :
    //   ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
        todo.setCategory(new Category(categoryCode, locale));
        todo.setGroups(new Groups(groupsId));    // 追加
                    :
    }
                    :
    // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
    public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
                    :
        this.categoryCode = todo.getCategory().getPkey().getCode();
        this.groupsId = todo.getGroups().getId();    // 追加
                    :
```

```
        }
    }
```

①グループは必須項目ではないので、バリデーションでチェックしない。



【図3-8】グループの入力

3.6 グループによる閲覧制限

次に【仕様４】を実装します。

【仕様４】共有したToDoは、ToDo登録者と共有グループ所属のユーザーが閲覧・編集できる。

まずToDoを閲覧(検索)できる条件ですが、もう少し細かく書くと以下のようになります。
1) todoのowner_idが自分のaccountのid
または
2) todoのgroup_idが自分の所属するグループのいずれか

たとえばユーザーがaccount.id=3なら、表示するToDoは以下のようなSELECT文で表せます。

```
SELECT    todo.*
  FROM    todo
  WHERE    todo.owner_id = 3
    OR    todo.groups_id    IN ( SELECT    groups_account.groups_id
                                  FROM    groups_account
                                 WHERE    groups_account.account_id = 3);
```

このOR以降が新しく必要となる部分です。これをTodoDaoImpl#findByCriteria()に追加します。

【リスト3-16】com.example.todolist.dao.TodoDaoImpl.java(変更前)

```
public Page<Todo> findByCriteria(TodoQuery todoQuery, Integer accountId,
                                 Pageable pageable) {

    CriteriaBuilder builder = entityManager.getCriteriaBuilder();
    CriteriaQuery<Todo> query = builder.createQuery(Todo.class);
    Root<Todo> root = query.from(Todo.class);
    List<Predicate> predicates = new ArrayList<>();
                :
```

```
        // 完了
        if (todoQuery.getDone() != null && todoQuery.getDone().equals("Y")) {
            predicates.add(
                builder.and(builder.equal(root.get(Todo_.DONE),
todoQuery.getDone())));
        }

        // 所有者
        predicates.add(builder.and(builder.equal(root.get(Todo_.OWNER_ID),
accountId)));

        // SELECT作成
        Predicate[] predArray = new Predicate[predicates.size()];
                    :
}
```

↓

【リスト3-17】com.example.todolist.dao.TodoDaoImpl.java(変更後)

```
                    :
import jakarta.persistence.criteria.Subquery;
import com.example.todolist.entity.GroupsAccount;
import com.example.todolist.entity.GroupsAccount_;
                    :
public Page<Todo> findByCriteria(TodoQuery todoQuery, Integer accountId,
                                 Pageable pageable) {

        CriteriaBuilder builder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Todo> query = builder.createQuery(Todo.class);
        Root<Todo> root = query.from(Todo.class);
        List<Predicate> predicates = new ArrayList<>();
                    :
        // 完了
        if (todoQuery.getDone() != null && todoQuery.getDone().equals("Y")) {
            predicates.add(
```

```
                builder.and(builder.equal(root.get(Todo_.DONE),
todoQuery.getDone())));
        }

        // -------------------- 変更ここから ↓ ↓ ↓ --------------------
        // 所属グループを求めるサブクエリ
        Subquery<Integer> subquery = query.subquery(Integer.class);    // ①
        Root<GroupsAccount> subqueryRoot =
subquery.from(GroupsAccount.class); // ②
        subquery.select(subqueryRoot.get(GroupsAccount_.GROUPS_ID))
                .where(builder
                    .equal(subqueryRoot.get(GroupsAccount_.ACCOUNT_ID),
accountId));    // ③

        // 所有者 or 所属グループのもの
        predicates.add(
            builder.and(
                builder.or(    // ④
                    builder.equal(root.get(Todo_.OWNER_ID), accountId),
                    root.get(Todo_.GROUPS).in(subquery)    // ⑤
                )
            )
        );
        // -------------------- 変更ここまで ↑ ↑ ↑ --------------------

        // SELECT作成
        Predicate[] predArray = new Predicate[predicates.size()];
                    :
}
```

➡Criteria APIについては『基礎編』「10.4 Criteria APIによる動的クエリの実行」参照

①Subqueryを取得

・前述のSELECT文のように問合せ(query)の中で、さらに問合せ(sub query)を実行する場合は、**Subquery**クラス(jakarta.persistence.criteria.Subquery)を使います。これはCriteriaQuery#subquery()で取得します。
・ここではInteger型であるgroups_account.groups_idを検索するので、それを戻り値の総称型<Integer>と引数Integer.classで表します。

②検索対象エンティティの指定
・GroupsAccountエンティティ、つまりgroups_accountテーブルであることを指定します。

③SELECT文作成
・これでsubqueryは、以下のようなSELECT文を表します。

```
SELECT groups_id FROM groups_account WHERE account_id = accountId
```

④③の結果をTodoの絞り込み条件に追加
・下記のように、変更前後を比べると**CriteriaBuilder#or()**で**論理和**になっていることがわかると思います。

```
predicates.add(
        builder.and(
                builder.equal(root.get(Todo_.OWNER_ID), accountId)));
                ↓
        predicates.add(
                builder.and(
                        builder.or(    // ④
                                builder.equal(root.get(Todo_.OWNER_ID), accountId),
                                root.get(Todo_.GROUPS).in(subquery)    // ⑤
                        )
                )
        );
```

⑤IN条件を指定する
・inはExpressionクラスのメソッドです。引数のSubquery型オブジェクトに対し、SELECT文のIN句と同じ働きをします。

これで「自分のtodo、または自分の所属するグループのtodo」が閲覧できます。たとえば「岡田さん」が「BBQサークル」のToDoを登録すると、「BBQサークル」所属の「井上さん」「稲垣さん」の画面にも表示されます。

【図3-9】自分のToDo+所属グループのToDo

しかしここで井上さんが「定例BBQ大会開催」をクリックしても「操作に何らかの誤りがあるようです。」と表示され、編集できません。これは『応用編』「22.2 プログラムによるアクセス制御」で「所有者(登録者)のみ編集可能」という制限を設けたためです。これを次節で「『登録者』または『ToDo登録者と同じグループに所属する』なら編集可能」へ変更します。

3.7 グループによる編集制限

「編集」はToDo一覧画面(todoList.html)の件名リンクをクリックしたときに呼び出されるTodolistController
#todoById()の処理を変更して対応します。以下のように『応用編』「22.2 プログラムによるアクセス制御」で「所有者(登録者)のみ編集可能」としたところを「**ToDo登録者と同じグループでも可**」とします。

【リスト3-18】com.example.todolist.controller.TodoListController.java

```
// ToDo表示
@GetMapping("/todo/{id}")
public ModelAndView todoById(@PathVariable(name = "id") int id,
ModelAndView mv,
                        :
      // 操作者のToDoか?
      // if (todo.getOwnerId().equals(accountId)) {
      //          ↓
      // 操作者のToDoか? or Todoと同じグループに所属しているか？
      @SuppressWarnings("unchecked")
      List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
      if (todo.getOwnerId().equals(accountId) || isBelong(groupsList,
todo.getGroups())) {
                        :
      }
      :
}
:
// 所属するグループか？
private boolean isBelong(List<Groups> groupsList, Groups groups) {
      return groupsList.stream().anyMatch(g -> g.getId().equals(groups.getId()));
}
```

件名リンクでクリックされたToDoのグループが、セッションに格納している操作者の所属グループに含まれているときもToDo入力画面を表示します。このグループ判定処理がisBelong()です。Javaのストリームを利用していますが、別な書き方(ストリームを使わない)をすると以下のようになります。

■ストリームを使わないisBelong()

```
    private boolean isBelong(List<Groups> groupsList, Groups groups) {
        Integer id = groups.getId();
        for (Groups g :groupsList) {
            if (g.getId().equals(id) ) {
                return true;
            }
        }
        return false;
    }
```

こちらは「なぜfor文でループさせているのか？」その意図を読み解く必要があります。ストリームなら「条件に合致するものの有無を判定したい(anyMatch())」という具合に、何をしたいコードなのか？わかりやすくなります。どちらで書いてもいいのですが、慣れるとストリームを使った方がコンパクト、かつ意図を伝えやすくなります。

これで所属するグループのToDoも一覧からクリックして、編集できるようになります。

以下は井上さんが【図3-9】の状態から、所属グループのToDo「定例BBQ大会開催」をクリックした結果ですが、編集画面へ遷移できていることがわかります。

【図3-10】所属グループのToDo編集

3.8 グループによる削除制限

最後に【仕様５】です。

【仕様５】ToDoの削除は登録者のみ可、タスクは登録者と共有グループ所属のユーザーのみ可とする。

ToDo削除処理には、登録者かどうか判定する処理を追加します。

【リスト3-19】com.example.todolist.controller.TodoListController.java

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

      // ---------- 追加ここから ↓ ↓ ↓ ----------
      // 削除できるのは登録者のみ
      Integer accountId = (Integer) session.getAttribute("accountId");
      if (!todoData.getOwnerId().equals(accountId)) {
         // 削除NGメッセージをセットしてリダイレクト
         String msg = messageSource.getMessage("msg.e.todo_cannot_delete",
null, locale);
         redirectAttributes.addFlashAttribute("msg", new OpMsg("E", msg));
         return "redirect:/todo/" + todoId;
      }
      // ---------- 追加ここまで ↑ ↑ ↑ ----------

      // 添付ファイルを削除
      todoService.deleteAttachedFiles(todoId);
                  :
      }
```

タスクは登録者に加え、同一グループでもOKとします。

【リスト3-20】com.example.todolist.controller.TodoListController.java

```
// Task削除処理
@GetMapping("/task/delete")
public ModelAndView deleteTask(@RequestParam(name = "task_id") int taskId,
                        :
   // 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)) {
      //          ↓
      // 操作者のToDoか? or Todoと同じグループに所属しているか？
      @SuppressWarnings("unchecked")
      List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
      if (todo.getOwnerId().equals(accountId) || isBelong(groupsList,
todo.getGroups())) {
                        :
```

【リスト3-21】src/main/resources/i18n/OperationMessages_en.properties

```
msg.e.todo_cannot_delete=Only registered user can delete todo.
```

【リスト3-22】src/main/resources/i18n/OperationMessages_ja.properties

```
msg.e.todo_cannot_delete=Todoを削除できるのは登録した人だけです。
```

これで削除も完了です。

以下は井上さんが【図3-10】の状態から、[削除]ボタンをクリックした場合ですが、自分が登録者ではないToDoのため、削除できずエラーメッセージが表示されます。

【図3-11】ToDo削除は登録者のみ許される

以上で「共有グループの導入」は一段落です。しかしこの機能を導入したことにより、***さらに厄介な事象が起こります***。それについては第５，６章で取り上げます。

> 本来であれば「一度設定したグループを変更する」といった場合にも何らかのルール、処理が必要となりますが、本書では省略します。

4. Viewを利用したテーブルの検索

プロジェクト名	Todolist19
作成ファイル	com.example.todolist.entity.Todolist.java com.example.todolist.repository.TodolistRepository.java
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.service.TodoService.java src/main/resources/templates/todoList.html
SQLファイル	src/main/resources/sql/40_create_v_todolist.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「いちいちtodoとかtaskとかテーブル結合すんの面倒！」
自分「...(だから、なんでSQLで検索してんだよ！)」
先輩「ビュー作ってよ」
自分「ビュー?」
先輩「誰かが作ればみんな楽できる！」
自分「？」
先輩「よろしくね！」ｽﾀｺﾗｻｯｻ
自分「えっ...(だからビューって何？)」

4.1 View

Criteria APIは、問合せを動的に組み立てられる反面、非常に難解です。書くのも大変ですし、パッと見ただけでは何をしているかわかりません(前章のようにSubqueryを使うとなおさらです)。RDBなのだから、SQLでなんとかできないのか？と思う方もいるでしょう。そこで本章ではRDBの**ビュー(View)**という機能を使い、TodoDaoImpl#findByCriteria()を置き換えます。

> 本章のビューはThymeleafなどで作成した画面などを指すビューとは別なものです。(ただし、どちらもスペルはview)

ビューはSELECT文の実行結果に名前を付ける機能です。
たとえば次のようなToDo一覧を作成したいとします。

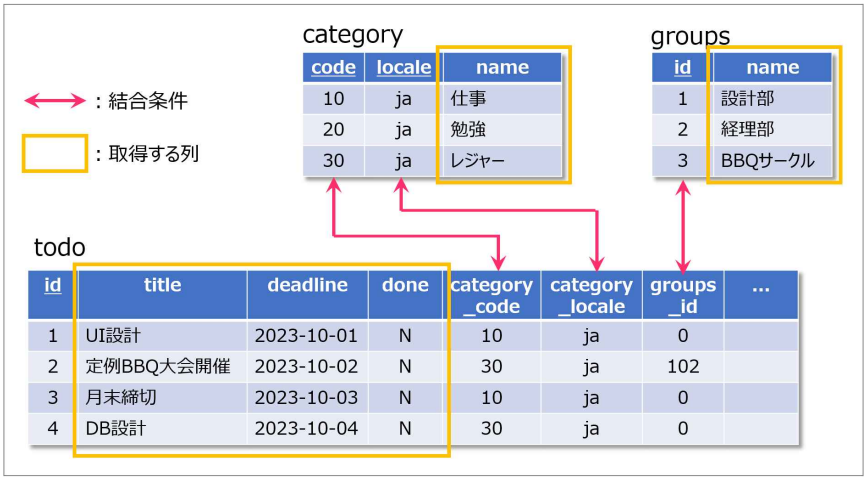
■目的の検索結果

```
 カテゴリ ┆  グループ   ┆      title      ┆  deadline  ┆ done
----------+-------------+-----------------+------------+------
 仕事     ┆             ┆ UI設計          ┆ 2023-10-01 ┆ N
 レジャー ┆ BBQサークル ┆ 定例BBQ大会開催 ┆ 2023-10-02 ┆ N
 仕事     ┆             ┆ 月末締切        ┆ 2023-10-03 ┆ N
 仕事     ┆             ┆ DB設計          ┆ 2023-10-04 ┆ N
(4 行)
```

以下がそのSELECT文です。このようにtodo,category,groupsを結合しています。

```
SELECT  category.name AS カテゴリ, groups.name AS グループ, todo.title,
todo.deadline, todo.done
  FROM  todo
  JOIN  category  ON  todo.category_code = category.code
   AND  todo.category_locale = category.locale
  JOIN  groups  ON  todo.groups_id = groups.id
ORDER  BY todo.id;
```

参考までに図解すると以下のようになります。



code	locale	name
10	ja	仕事
20	ja	勉強
30	ja	レジャー

id	name
1	設計部
2	経理部
3	BBQサークル

id	title	deadline	done	category_code	category_locale	groups_id	...
1	UI設計	2023-10-01	N	10	ja	0	
2	定例BBQ大会開催	2023-10-02	N	30	ja	102	
3	月末締切	2023-10-03	N	10	ja	0	
4	DB設計	2023-10-04	N	30	ja	0	

【図4-1】結合条件と取得列の関係

こういったSQLを都度入力するのは面倒です。もし頻繁に使うならビューを作成し、そちらを使った方が効率的です。

■形式

```
CREATE VIEW　ビュー名　AS　ビューの定義(=SELECT文)
```

```
CREATE VIEW　todolist　AS
  SELECT　category.name AS カテゴリ,　groups.name AS グループ, todo.title, todo.deadline, todo.done
    FROM　todo
    JOIN　category　ON　todo.category_code = category.code
     AND　todo.category_locale = category.locale
    JOIN　groups　ON　todo.groups_id = groups.id
  ORDER　BY todo.id;
```

※１行目のASより後ろは前述のSELECT文と全く同じです。これにtodolistという名前をつけた、という感じです。

これでtodolist(=ビュー)を検索すると、同じ結果が得られます。

■実行結果(1)

```
todolist=> SELECT * FROM todolist;
 カテゴリ ¦  グループ   ¦      title      ¦  deadline  ¦ done
----------+-------------+-----------------+------------+------
 仕事     ¦             ¦ UI設計          ¦ 2023-10-01 ¦ N
 レジャー ¦ BBQサークル ¦ 定例BBQ大会開催 ¦ 2023-10-02 ¦ N
 仕事     ¦             ¦ 月末締切        ¦ 2023-10-03 ¦ N
 仕事     ¦             ¦ DB設計          ¦ 2023-10-04 ¦ N
(4 行)


todolist=>
```

さらにWHERE句で絞り込むこともできますし、他のテーブル、あるいは他のビューと結合させることもできます。

■実行結果(2)

```
todolist=> select * from todolist where title like '%BBQ%';
 カテゴリ ¦  グループ   ¦      title      ¦  deadline  ¦ done
----------+-------------+-----------------+------------+------
 レジャー ¦ BBQサークル ¦ 定例BBQ大会開催 ¦ 2023-10-02 ¦ N
(1 行)


todolist=>
```

別な見方をすると、ビューは「**SELECTしたときに作られる表(仮想表)**」です。しかし、ビューを知らない人には、あたかもそういうテーブルが存在しているように見えるでしょう。

> ビューの元となっているテーブル(この場合はtodo, groups, category)のデータを変更すると、その内容は次回以降ビュー(todolist)をSELECTしたときに反映されます。

> ビューを作成すると、その元となったテーブルを削除(DROP)できなくなります。DROPする場合は、
> ・ビュー →テーブルの順にDROPする　または
> ・「**DROP TABLE** テーブル名 **CASCADE**」のようにCASCADEを指定して、ビューも削除するとしなければなりません。
> なおビューをDROPしても、元になったテーブルのデータは削除されません。

> ビューは更新することも可能です。ただし、いろいろな条件があります。またRDBの種類、バージョンによっても微妙に異なります。興味がある方は調べてみてください。
> ちなみに上記v_todolistは更新できません。なぜできないか？これも興味がある方は調べてみてください。

4.2 Viewのエンティティ化

以下は前章の【リスト3-17】TodoDaoImpl#findByCriteria()相当の検索を行うビューv_todolistを定義するものです(詳しい仕組みは章末の解説を参照)。

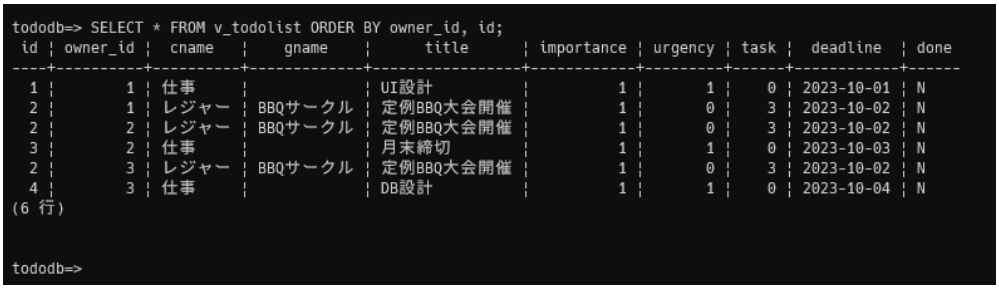
【リスト4-1】src/main/resources/sql/40_create_v_todolist.sql

```
DROP VIEW IF EXISTS v_todolist;
CREATE VIEW v_todolist AS
SELECT td2.id, td2.owner_id, c.NAME AS cname, g.NAME AS gname, td2.title,
        td2.importance, td2.urgency, COALESCE(tk.cnt, 0) AS task, td2.deadline,
        td2.done
  FROM
(
    SELECT id, owner_id, groups_id, category_code, category_locale,
            title, importance, urgency, deadline, done
      FROM todo
     WHERE groups_id = 0
  UNION
    SELECT td.id, ga.account_id AS owner_id, td.groups_id, td.category_code,
td.category_locale,
            td.title, td.importance, td.urgency, td.deadline, td.done
      FROM todo td
      JOIN groups_account ga ON ga.groups_id = td.groups_id
     WHERE td.groups_id != 0
) td2
  JOIN category c ON td2.category_code = c.code AND td2.category_locale
= c.locale
  JOIN groups g ON td2.groups_id = g.id
  LEFT OUTER JOIN ( SELECT todo_id, COUNT(*) AS cnt FROM task GROUP
BY todo_id ) tk
                ON td2.id = tk.todo_id;
```

　やや複雑なビューですが、findByCriteria()より「何をしているか？」を把握しやすいと思います(特にSQLがわかる方)。また思った通りの結果が得られるか、確認しやすいというメリットもあります(psqlにコピー＆ペーストして実行)。

これをSELECTすると以下のようになります。表示項目はToDo一覧の内容をカバーしています。

■実行結果



```
tododb=> SELECT * FROM v_todolist ORDER BY owner_id, id;
 id | owner_id |  cname   |   gname    |      title      | importance | urgency | task |  deadline  | done
----+----------+----------+------------+-----------------+------------+---------+------+------------+------
  1 |        1 | 仕事     |            | UI設計          |          1 |       1 |    0 | 2023-10-01 | N
  2 |        1 | レジャー | BBQサークル | 定例BBQ大会開催 |          1 |       0 |    3 | 2023-10-02 | N
  2 |        2 | レジャー | BBQサークル | 定例BBQ大会開催 |          1 |       0 |    3 | 2023-10-02 | N
  3 |        2 | 仕事     |            | 月末締切        |          1 |       1 |    0 | 2023-10-03 | N
  2 |        3 | レジャー | BBQサークル | 定例BBQ大会開催 |          1 |       0 |    3 | 2023-10-02 | N
  4 |        3 | 仕事     |            | DB設計          |          1 |       1 |    0 | 2023-10-04 | N
(6 行)


tododb=>
```

id=2が3行あるのは、これが「BBQサークル」のToDoであり、このグループに所属する人が 3 人いるからです(owner_id = 1,2,3)。またowner_idを見ると、自分のToDoに加え、所属するグループのToDoも含まれていることがわかります。あとは条件に従い、ここから該当するレコードを抽出すればよいわけです。

例1. account.id=1の人が重要度:高を検索した場合

```
SELECT * FROM v_todolist FROM owner_id = 1 AND importance = 1
```

例2. account.id=2の人が完了:Yを検索した場合

```
SELECT * FROM v_todolist FROM owner_id = 2 AND done = 'Y'
```

ではCriteria APIに代わり、このv_todolistを使ってToDoを検索できるようにします。

まずビューに対応するエンティティを作成します。これはテーブルと同じように@Tableを付与し、そこにビュー名を指定すればOKです。

【リスト4-2】com.example.todolist.entity.Todolist.java

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

import java.sql.Date;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;
```

```java
@Entity
@Table(name = "v_todolist")    // ①
@Data
public class Todolist {
    @Id    // ②
    @Column(name = "id")
    private Integer id;

    @Column(name = "owner_id")
    private Integer ownerId;

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

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

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

   @Column(name = "importance")
    private Integer importance;

    @Column(name = "urgency")
    private Integer urgency;

    @Column(name = "task")
    private Integer numOfTasks;

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

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

```
}
```

①ビュー名を指定する。
②主キーの指定
・エンティティに@Idは必須です。これは**ビューでも同じ**です。
・上記v_todolistのSELECT結果を見ると、id列に同じ値があるため一意でないように見えます。しかし検索するときは、owner_idで絞り込むので検索結果内では一意となります。

リポジトリもテーブルと同様に定義します。

【リスト4-3】com.example.todolist.repository.TodolistRepository.java

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

import java.sql.Date;
import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.todolist.entity.Todolist;

@Repository
public interface TodolistRepository extends JpaRepository<Todolist, Integer> {
  // 検索条件に期限を含む
  Page<Todolist>

findByOwnerIdAndTitleLikeAndImportanceInAndUrgencyInAndDeadlineBetweenAndDoneIn(
        Integer ownerId, String title, List<Integer> importance, List<Integer>
urgency,
        Date from ,Date to, List<String> done,
        Pageable pageable);

  // 検索条件に期限を含まない
  Page<Todolist>
findByOwnerIdAndTitleLikeAndImportanceInAndUrgencyInAndDoneIn(
```

```
        Integer ownerId, String title, List<Integer> importance, List<Integer>
urgency,
        List<String> done,
        Pageable pageable);
}
```

最初のメソッドは、すべての検索条件をつなげた名前になっています。分解すると下表のようになります。

【表4-1】メソッド名の意味

メソッド名称	検索条件	引数
findByOwnerId	owner_idが一致	Integer ownerId
And	かつ	
TitleLike	titleのパターンが一致(LIKE句)	String title
And	かつ	
ImportanceIn	importanceが引数のリストに含まれる (IN句)	List<Integer> importance
And	かつ	
UrgencyIn	urgencyが引数のリストに含まれる (IN句)	List<Integer> urgency
And	かつ	
DeadlineBetween	deadlineが指定の範囲内 (BETWEEN句)	Date from ,Date to
And	かつ	
DoneIn	doneが引数のリストに含まれる (IN句)	List<String> done

こう定義しておけば、たとえば「account.id=1の人が重要度:高、期限：2022-01-01～2022-12-31」で検索した場合、下表のような引数を渡せば良いことになります。

【表4-2】引数指定例(account.id=1の人が重要度:高、期限：2022-01-01～2022-12-31)

引数	値	備考
Integer ownerId	1	入力された条件
String title	"%"	(全件該当する条件)
List<Integer> importance	[1]	入力された条件
List<Integer> urgency	[1, 2]	(全件該当する条件)
Date from ,Date to	"2022-01-01","2022-12-31"	入力された条件
List<String> done	["Y", "N"]	(全件該当する条件)

ポイントは、検索条件が指定されなかった項目には「**全件該当する値**」**を渡している**ところです。こういう形式にしておくと、条件によって動的に組み立てなくてすみます。

なお期限は開始/終了の片方だけ入力された場合、下表のような値で**BETWEEN** from **AND** toを実行します。

【表4-3】期限省略時の設定値

期限：開始	期限：終了	from	to
未入力	'2022-12-31'	'1900-01-01'	'2022-12-31'
'2022-01-01'	未入力	'2022-01-01'	'2999-12-31'

では「開始/終了とも未入力」、つまり「(deadline未設定も含め)全期間のToDo」の場合はどうでしょう？ 上表からfrom='1900-01-01', to='2999-12-31'で良いように思えますが、それでは期限未設定(=NULL)が選択できません。

> NULL値との比較に使えるのはIS NULL演算子だけです。

そこで「開始/終了が未入力」なら「期限を検索条件から外す」こととし、それ用のメソッドを追加します。これが【リスト4-3】２つ目のメソッドです。どちらを使うかは入力された検索条件から判断します。

以下はここまで説明した内容をもとに作成したTodoService#findByCriteria()です。

【リスト4-4】com.example.todolist.service.TodoService.java

```
                    :
import java.util.ArrayList;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import com.example.todolist.entity.Todolist;
import com.example.todolist.repository.TodolistRepository;
                    :
public class TodoService {
                    :
   private final TodolistRepository todolistRepository; // 追加
                    :
   // ------------------------------------------------------------------------
   // Todolist(v_todolist)の検索処理
   // ------------------------------------------------------------------------
   public Page<Todolist> findByCriteria(TodoQuery todoQuery, Integer accountId,

Pageable pageable) {
      // 件名
      String title = "";
      if (todoQuery.getTitle().length() > 0) {
         title = "%" + todoQuery.getTitle() + "%";
      } else {
         title = "%";
      }

      // 重要度
      List<Integer> importance = new ArrayList<>();
      if (todoQuery.getImportance() == -1) {
         // 指定なし
         importance.add(0);
         importance.add(1);
      } else {
         importance.add(todoQuery.getImportance());
```

```
}

// 緊急度
List<Integer> urgency = new ArrayList<>();
if (todoQuery.getUrgency() == -1) {
    // 指定なし
    urgency.add(0);
    urgency.add(1);
} else {
    urgency.add(todoQuery.getUrgency());
}

// 期限：開始～
java.sql.Date from;
if (todoQuery.getDeadlineFrom().equals("")) {
    // 指定なし
    from = Utils.str2date("1900-01-01");   //   String → java.sql.Date変換メソッド
} else {
    from = Utils.str2date(todoQuery.getDeadlineFrom());
}

// ～期限：終了で検索
java.sql.Date to;
if (todoQuery.getDeadlineTo().equals("")) {
    // 指定なし
    to = Utils.str2date("2999-12-31");
} else {
    to = Utils.str2date(todoQuery.getDeadlineTo());
}

// 完了
List<String> done = new ArrayList<>();
if (todoQuery.getDone() != null && todoQuery.getDone().equals("Y")) {
    done.add("Y");
```

```
        } else {
            // 指定なし
            done.add("Y");
            done.add("N");
        }

        // 検索して結果を返す
        if (todoQuery.getDeadlineFrom().equals("") &&
todoQuery.getDeadlineTo().equals("")) {
            return todolistRepository

.findByOwnerIdAndTitleLikeAndImportanceInAndUrgencyInAndDoneIn(
                            accountId, title, importance, urgency, done, pageable);
        } else {
            return todolistRepository

.findByOwnerIdAndTitleLikeAndImportanceInAndUrgencyInAndDeadlineBetweenA
ndDoneIn(
                        accountId, title, importance, urgency, from, to, done, pageable);
        }
    }
}
```

少々長いコードですが、簡単に言えば以下のようなことをやっています。
①todoQueryに検索条件がセットされていれば、それを引数とする。
②検索条件がなければ、全件該当する条件を作成し引数とする。
③期限：開始、終了の有無で実行するメソッドを変える。
→【表4-1】【表4-2】と照らし合わせるとわかりやすいでしょう。

これに合わせCriteria API版TodoDaoImpl#findByCriteria()の呼び出し箇所を変更します(3箇所)。

【リスト4-5】com.example.todolist.controller.TodoListController.java

```
        :
import com.example.todolist.entity.Todolist;
```

```
    :
public class TodoListController {
                    :
    private final AccountRepository accountRepository;

    // ---------- コメントにする or 削除 ここから ↓ ↓ ↓ ----------
    // @PersistenceContext
    // private EntityManager entityManager;
    // TodoDaoImpl todoDaoImpl;

    // @PostConstruct
    // public void init() {
    //       todoDaoImpl = new TodoDaoImpl(entityManager);
    //   }
    // ---------- コメントにする or 削除 ここまで ↑ ↑ ↑ ----------

    // ToDo一覧表示
    @GetMapping("/todo")
    public ModelAndView showTodoList(ModelAndView mv,
                    :
        // Todo検索
        Integer accountId = (Integer)session.getAttribute("accountId");
        //Page<Todo> todoPage = todoDaoImpl.findByCriteria(todoQuery, accountId,
prevPageable);
        //       ↓
        Page<Todolist> todoPage = todoService.findByCriteria(todoQuery, accountId,
prevPageable);
                    :
    }
                    :
    // ToDo検索処理
    @PostMapping("/todo/query")
    public ModelAndView queryTodo(@ModelAttribute TodoQuery todoQuery,
BindingResult result,
```

```
                    :
        //Page<Todo> todoPage = null;
        //       ↓
        Page<Todolist> todoPage = null;

        if (todoService.isValid(todoQuery, result, locale)) {
            // エラーがなければ検索
            Integer accountId = (Integer)session.getAttribute("accountId");
            //todoPage = todoDaoImpl.findByCriteria(todoQuery, accountId, pageable);
            //       ↓
            todoPage = todoService.findByCriteria(todoQuery, accountId, pageable);
                    :
        }
                    :
        // ページリンククリック時
    @GetMapping("/todo/query")
    public ModelAndView queryTodo(
                                    @PageableDefault(page = 0, size = 5, sort = "id")
Pageable pageable,
                    :
        // 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");
        //Page<Todo> todoPage = todoDaoImpl.findByCriteria(todoQuery, accountId,
pageable);
        //       ↓
        Page<Todolist> todoPage = todoService.findByCriteria(todoQuery, accountId,
pageable);
                    :
    }
}
```

またToDo一覧画面(todoList.html)も、Todolistのプロパティ名に合わせて一部変更します。ビューの場合、検索結果は構造を持たない「ただの表」なので、シンプルになります。

【リスト4-6】src/main/resources/templates/todoList.html

```
                :
  <!-- 検索結果エリア -->
  <table border="1">
    <tr>
                :
    </tr>
    <tr th:each="todo:${todoList}">
      <!-- id -->
      <td th:text="${todo.id}"></td>
      <!-- カテゴリ -->
      <td th:text="${todo.cname}"></td> <!-- ${todo.category.name}から変更 --
>
      <!-- グループ -->
      <td th:text="${todo.gname}"></td> <!-- ${todo.groups.name}から変更 -->
      <!-- 件名 -->
                :
      <!-- タスク数 -->
      <!-- ${#lists.size(todo.taskList)}から変更 -->
      <td th:text="${todo.numOfTasks}" style="text-align: center"></td>
                :
```

これで置き換え完了です。ページング操作も含め、これまでと同じ動きをします。

> com.example.todolist.dao.TodoDaoおよびTodoDaoImplはこの後使用しないので、パッケージ(com.example.todolist.dao)ごと削除してもかまいません。

4.3 (N+1)問題

以上のようにCriteria API/ビューのどちらでも同じ結果が得られます。しかし裏側で実行されているSELECT文の実行回数は、大きく違っています。これはapplication.propertiesへ、以下のパラメーターを追加すると確認できます。

【リスト4-7】src/main/resources/application.properties

```
#SQL Log
#Hibernate >= 6
logging.level.org.hibernate=warn
logging.level.org.hibernate.SQL=debug
logging.level.org.hibernate.orm.jdbc.bind=trace
```

詳細は割愛しますが、これでSpring Bootが実行したSELECT/INSERT/UPDATE/DELETE文と、そのパラメーターがSTSのコンソールへ出力されるようになります。

まずビューv_todolistで検索している本章Todolist19の場合です。

> 見やすくするために実行日時、表名による修飾、列別名などを削除し、適宜改行しています。「--①」など「--丸付き数字」は説明用に追加したものです。

■Todolist19の場合

```
--①
SELECT   id, cname, deadline, done, gname, importance, task, owner_id, title,
urgency
  FROM   v_todolist
WHERE owner_id = ?
   AND   title   LIKE   ?   ESCAPE '¥'
   AND   importance   IN (?,?)
   AND   urgency   IN (?,?)
   AND   done IN (?,?)
ORDER BY   id   ASC
OFFSET   ? ROWS   FETCH FIRST   ?   ROWS   ONLY
binding parameter [1] as [INTEGER] - [1]
```

```
binding parameter [2] as [VARCHAR] - [%]
                :
binding parameter [9] as [INTEGER] - [0]
binding parameter [10] as [INTEGER] - [5]
--②
SELECT count(id)
  FROM   v_todolist
WHERE   owner_id = ?
   AND   title   LIKE   ?   ESCAPE '¥'
   AND   importance   IN (?,?)
   AND   urgency   IN (?,?)
   AND   done IN (?,?)
binding parameter [1] as [INTEGER] - [1]
binding parameter [2] as [VARCHAR] - [%]
                :
binding parameter [7] as [VARCHAR] - [Y]
binding parameter [8] as [VARCHAR] - [N]
```

これはaccount.id=1のユーザーが、条件を指定せずToDoを検索した場合です。1 回のSELECT文実行で 1 画面分のToDoを取得しています。

一方Criteria APIを使っている前章Todolist18の場合は、以下のようになります。こちらは同じ結果を得るのにSELECT文を13回実行してます。

■Todolist18場合

```
--①
SELECT   id, category_code, category_locale, deadline, done, groups_id,
importance,
           owner_id, title, urgency
  FROM   todo
WHERE   title   LIKE ? ESCAPE'¥'
   AND   ( owner_id = ?
             OR   groups_id IN ( SELECT groups_id FROM groups_account WHERE
account_id = ? ))
ORDER   BY id   ASC
binding parameter [1] as [VARCHAR] - [%]
```

```
binding parameter [2] as [INTEGER] - [1]
binding parameter [3] as [INTEGER] - [1]
--②
SELECT code,locale,name  FROM  category  WHERE  (code,locale) IN((?,?))
binding parameter [1] as [VARCHAR] - [10]
binding parameter [2] as [VARCHAR] - [ja]
--③
SELECT  id,name  FROM  groups  WHERE  id=?
binding parameter [1] as [INTEGER] - [100]
--④
SELECT  id,name  FROM groups  WHERE id=?
binding parameter [1] as [INTEGER] - [102]
--⑤
SELECT  code,locale,name  FROM category  WHERE (code,locale) IN((?,?))
binding parameter [1] as [VARCHAR] - [20]
binding parameter [2] as [VARCHAR] - [ja]
--⑥
SELECT id,name  FROM groups  WHERE id=?
binding parameter [1] as [INTEGER] - [0]
--⑦
SELECT  code,locale,name  FROM  category  WHERE (code,locale) IN((?,?))
binding parameter [1] as [VARCHAR] - [30]
binding parameter [2] as [VARCHAR] - [ja]
--⑧
SELECT  id, category_code, category_locale, deadline, done, groups_id,
importance,
        owner_id, title, urgency
  FROM  todo
WHERE title  LIKE ? ESCAPE'¥'
  AND  ( owner_id = ?
          OR  groups_id  IN( SELECT groups_id FROM groups_account WHERE
account_id = ? tomcat ))
ORDER  BY  id  ASC
OFFSET ?  ROWS FETCH FIRST ?  ROWS ONLY
```

```
binding parameter [1] as [VARCHAR] - [%]
binding parameter [2] as [INTEGER] - [1]
binding parameter [3] as [INTEGER] - [1]
binding parameter [4] as [INTEGER] - [0]
binding parameter [5] as [INTEGER] - [5]
--⑨
SELECT todo_id,id,deadline,done,title  FROM  task  WHERE todo_id=?  ORDER BY  id  ASC
binding parameter [1] as [INTEGER] - [1]
--⑩
SELECT todo_id,id,deadline,done,title  FROM  task  WHERE todo_id=?  ORDER BY  id  ASC
binding parameter [1] as [INTEGER] - [2]
--⑪
SELECT todo_id,id,deadline,done,title  FROM  task  WHERE todo_id=?  ORDER BY  id  ASC
binding parameter [1] as [INTEGER] - [3]
--⑫
SELECT todo_id,id,deadline,done,title  FROM  task  WHERE todo_id=?  ORDER BY  id  ASC
binding parameter [1] as [INTEGER] - [4]
--⑬
SELECT todo_id,id,deadline,done,title  FROM  task  WHERE todo_id=?  ORDER BY  id  ASC
binding parameter [1] as [INTEGER] - [5]
```

①todo検索
②⑤⑦カテゴリ検索(①に 3 種類のカテゴリが含まれているため)
③④⑥グループ名検索
⑧ 1 ページ分のTodoを検索(OFFSET ? ROWS FETCH FIRST ? ROWS ONLY)
⑨～⑬todoに関連するtask検索(タスク数を算出するため)

以上をまとめると、Todolist18(Criteria API)の場合、SELECT文の実行回数は、以下のように「表示するToDo数に応じて増えそう」です。タスク数にもよりますが、1画面で数十回実行するケースもありそうです。

※ 1 画面分のToDo取得に必要なSELECT文の実行回数
= 2回(①⑧) + カテゴリの種類数(②⑤⑦) + グループの種類数(③④⑥) + 関連するタスク数(⑨〜⑬)

このようにCriteria API、あるいはJPAを使うと、意図せず大量のSELECT文を実行するコードになり得ます。これは「**N+1問題**」などと呼ばれています。@OneToManyにEAGER Fetch(fetch = FetchType.EAGER)を指定することで抑止可能な場合もありますが、万能ではありません。

> 興味がある方は「JPA N+1問題」で検索してみてください。

「SELECT文の実行回数が多い」というのは、WebAPサーバーとDBサーバーがネットワーク的に離れている場合、応答時間悪化のリスクとなります。これはSELECT文を**実行するごとに**WebAPサーバー⇔DBサーバー間で通信を行うためです。環境にもよりますがユーザーが「遅い」と感じてしまうレベルに発展する可能性もあります。またデータ件数によって「急に遅くなる」という印象を持たれるのは得策ではないでしょう。

ビューの場合、実行するSELECT文が複雑でもWebAPサーバー⇔DBサーバー間の通信は **1 往復**なので、条件に該当するものが増えても急激に遅くなることはないでしょう。もし遅くなったら、SELECT文の実行計画(EXPLAIN)を確認し、インデックスを設定するなど、プログラム以外の部分で対策を打てます。

しかし「SQLを書かなくてもアノテーションで関連するデータを取り出せる」というのはJPAの良さでもあります。環境や開発ルールに依存するところでもあるので、一概にどちらが良いとは言えません。いろいろなやり方を習得し、ベストな方法を選べる力が必要となるところです。

(補足) v_todolistの定義内容詳説

各テーブルが以下のよう状態だったとします。

■groupsテーブル

id	name
0	
100	設計部
101	経理部
102	BBQサークル

(4 行)

■groups_accountテーブル

id	groups_id	account_id
1	100	1
2	100	3
3	101	2
4	102	1
5	102	2
6	102	3

(6 行)

■accountテーブル

id	login_id	name	password
1	okada	岡田 是則	s6rizqfk
2	inoue	井上 俊憲	g73phw5n
3	inagaki	稲垣 絵美	s59mrtw3

(3 行)

■categoryテーブル

code	locale	name
10	ja	仕事
20	ja	勉強
30	ja	レジャー
10	en_US	Job
20	en_US	Study
30	en_US	Leisure

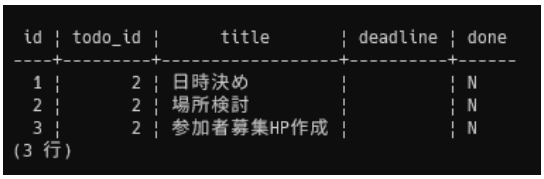
(6 行)

■todoテーブル

id	owner_id	title	importance	urgency	deadline	done	category_code	category_locale	groups_id
1	1	UI設計	1	1	2023-10-01	N	10	ja	0
2	1	定例BBQ大会開催	1	0	2023-10-02	N	30	ja	102
3	2	月末締切	1	1	2023-10-03	N	10	ja	0
4	3	DB設計	1	1	2023-10-04	N	10	ja	0

(4 行)

■taskテーブル



```
 id | todo_id |      title       | deadline | done
----+---------+------------------+----------+------
  1 |       2 | 日時決め         |          | N
  2 |       2 | 場所検討         |          | N
  3 |       2 | 参加者募集HP作成 |          | N
(3 行)
```

以下はv_todolistの定義です。

```
DROP VIEW  IF  EXISTS  v_todolist;
CREATE  VIEW v_todolist  AS
SELECT td2.id, td2.owner_id, c.NAME AS cname, g.NAME AS gname, td2.title,
        td2.importance, td2.urgency, COALESCE(tk.cnt, 0) AS task, td2.deadline,
        td2.done
  FROM
(
    SELECT id, owner_id, groups_id, category_code, category_locale,
            title, importance, urgency, deadline, done
      FROM  todo
     WHERE groups_id = 0
  UNION
    SELECT  td.id, ga.account_id AS owner_id, td.groups_id, td.category_code,
td.category_locale,
            td.title, td.importance, td.urgency, td.deadline, td.done
      FROM todo td
      JOIN groups_account ga ON ga.groups_id = td.groups_id
     WHERE td.groups_id != 0
) td2
  JOIN category c  ON td2.category_code = c.code  AND  td2.category_locale
= c.locale
  JOIN groups g  ON td2.groups_id = g.id
  LEFT OUTER  JOIN ( SELECT todo_id, COUNT(*) AS cnt  FROM task  GROUP
BY todo_id ) tk
                    ON td2.id = tk.todo_id;
```

複雑に見えますが、大きく分けると以下のようになっています。

```
SELECT ③
FROM
(
      SELECT ①
   UNION
      SELECT ②
) td2
   JOIN ①+②
```

①の部分はWHERE groups_id = 0により個人(=グループ共有しない)のToDoを抽出しています。

■todo①

id	owner_id	groups_id	category_code	category_locale	title	importance	urgency	deadline	done
1	1	0	10	ja	UI設計	1	1	2023-10-01	N
3	2	0	10	ja	月末締切	1	1	2023-10-03	N
4	3	0	10	ja	DB設計	1	1	2023-10-04	N

(3 行)

②はWHERE groups_id != 0なのでグループ共有ToDoです。この例で該当するのは次の 1 行です。

■todo②

id	owner_id	groups_id	category_code	category_locale	title	importance	urgency	deadline	done
2	1	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N

(1 行)

これをJOIN groups_account ga ON ga.groups_id = td.groups_idにより、そのグループに属するユーザーへ展開します。ここではgroups_id = 102なので、groups_accountで該当するのは次の 3 行です。

■groups_account

id	groups_id	account_id
4	102	1
5	102	2
6	102	3

(3 行)

よって②の結果は以下のようになります。

■todo②(結合後)

id	owner_id	groups_id	category_code	category_locale	title	importance	urgency	deadline	done
2	1	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N
2	2	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N
2	3	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N

(3 行)

この①と②をUNIONで合わせた結果にtd2という名前を付けます。

(SELECT ① UNION SELECT ②) td2

■td2

id	owner_id	groups_id	category_code	category_locale	title	importance	urgency	deadline	done
2	2	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N
3	2	0	10	ja	月末締切	1	1	2023-10-03	N
2	3	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N
2	1	102	30	ja	定例BBQ大会開催	1	0	2023-10-02	N
4	3	0	10	ja	DB設計	1	1	2023-10-04	N
1	1	0	10	ja	UI設計	1	1	2023-10-01	N

(6 行)

これにcategory, groups, タスク数をJOINして③の結果とします。

なお最後のJOINがLEFT OUTER JOIN(左外結合)になっているのは、todoにタスクがないとtkがNULLとなり、td2のレコードが無くなるからです。それを避けるためLEFT OUTER JOINとし、td2側が残るようにします。

```
    LEFT OUTER    JOIN ( SELECT todo_id, COUNT(*) AS cnt    FROM task    GROUP
BY todo_id ) tk
                          ON    td2.id = tk.todo_id;
```

③は対応するタスク数がない(tk.cntがNULL)の場合、0へ置き換えるよう、COALESCE(tk.cnt, 0) AS taskとしています。

これでSELECT * FROM v_todolist ORDER BY owner_id, id; を実行すると、以下のような結果が得られます。

⇒利用者全員の「自分のToDo」と「所属するグループのToDo」

■SELECT * FROM v_todolist ORDER BY owner_id, idの実行結果

id	owner_id	cname	gname	title	importance	urgency	task	deadline	done
1	1	仕事		UI設計	1	1	0	2023-10-01	N
2	1	レジャー	BBQサークル	定例BBQ大会開催	1	0	3	2023-10-02	N
2	2	レジャー	BBQサークル	定例BBQ大会開催	1	0	3	2023-10-02	N
3	2	仕事		月末締切	1	1	0	2023-10-03	N
2	3	レジャー	BBQサークル	定例BBQ大会開催	1	0	3	2023-10-02	N
4	3	仕事		DB設計	1	1	0	2023-10-04	N

(6 行)

-

5. トランザクション管理

プロジェクト名	Todolist20
作成ファイル	ー(なし)
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Account.java com.example.todolist.entity.Task.java com.example.todolist.entity.Todo.java com.example.todolist.entity.Todolist.java com.example.todolist.form.TaskData.java com.example.todolist.form.TodoData.java com.example.todolist.service.TodoService.java src/main/resources/templates/todoForm.html src/main/resources/templates/todoList.html src/main/resources/i18n/ValidationMessages_en.properties src/main/resources/i18n/ValidationMessages_ja.properties src/main/resources/i18n/FixedDisplayStrings_en.properties src/main/resources/i18n/FixedDisplayStrings_js.properties
SQLファイル	src/main/resources/sql/50_create_v_todolist.sql src/main/resources/sql/51_add_completed_to_todo.sql src/main/resources/sql/52_add_assign_progress_to_task.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」

自分「はい」

先輩「共有したToDoの進み具合がわかんないんだよなぁ...」

自分「？」

先輩「一覧のタスク数は総数でさ、そのうち何個終わってるかは中を見ないとわかんない」

自分「なるほど」

先輩「それとタスクも未完と完了の二択じゃないし、そもそもだれがやってるかわからん！」

自分「つまり、『進捗率』と『担当者』も入力できるようにしろと...」

先輩「おっ、理解が早いね！頼む！」ｽﾀｺﾗｻｯｻ

自分「はぁ...(「グループ共有」やらなきゃよかった...)」

5.1 導出項目

先輩の要件は２つです。
【要件１】完了したタスク数もToDo一覧へ表示する。
【要件２】タスクに担当者、進捗率を追加する。

【要件１】の完了タスク数には、以下のような実現案があるでしょう。
【実現案１】ToDoに「完了タスク数」列を追加→ユーザーの操作に合わせて設定(再計算)する。
【実現案２】完了タスク数が必要になった時点で、集計する(列として持たない)

このうち【実現案２】は、前章のv_todolistの変更で対応可能です(テーブルに列追加不要)。

```
DROP VIEW IF    EXISTS v_todolist;
CREATE VIEW v_todolist    AS
SELECT td2.id, td2.owner_id, c.NAME AS cname, g.NAME AS gname,
td2.title,
            td2.importance, td2.urgency,
            COALESCE(tk2.cnt, 0)    AS    completed,    -- 追加
            COALESCE(tk.cnt, 0)    AS task, td2.deadline, td2.done
FROM
(
      SELECT    id, owner_id, groups_id, category_code,
category_locale,
                  title, importance, urgency, deadline, done
        FROM    todo
       WHERE    groups_id = 0
```

```
  UNION
    SELECT   td.id, ga.account_id AS owner_id, td.groups_id,
td.category_code, td.category_locale,
                td.title, td.importance, td.urgency, td.deadline, td.done
       FROM todo td
       JOIN   groups_account ga ON ga.groups_id = td.groups_id
      WHERE   td.groups_id != 0
) td2
  JOIN   category c ON td2.category_code = c.code AND
td2.category_locale = c.locale
  JOIN   groups g    ON td2.groups_id = g.id
  LEFT OUTER   JOIN (SELECT   todo_id, COUNT(*)   AS cnt   FROM
task   GROUP BY todo_id) tk
                    ON   td2.id = tk.todo_id
  -- 以下を追加
  LEFT   OUTER   JOIN   (SELECT   todo_id, COUNT(done='Y' OR
NULL) AS cnt
                          FROM   task   GROUP BY todo_id) tk2
                    ON   td2.id = tk2.todo_id
```

> COUNT(done='Y' OR NULL)の「OR NULL」は何？必要なの？と思うかもしれません。この集計関数COUNTについて、PostgreSQLの公式ドキュメントには以下のように記載されています。
>
> https://www.postgresql.jp/document/14/html/functions-aggregate.html
> PostgreSQL 14.5文書 > 第9章 関数と演算子 > 9.21. 集約関数
> count(“any”) → bigint

非NULLの入力行数を返します。

つまりCOUNTは「()内がNULLにならないレコードの件数」を返します。単にCOUNT(done='Y')とすると、
・done='Y'のレコードはTRUE→非NULLのためカウントされる
・done='N'のレコードはFALSE→非NULLのためカウントされる
となり、全件カウントされてしまいます。
「OR NULL」を付けると
・done='Y'のレコード　「done='Y' OR NULL」→「TRUE OR NULL」　→TRUE(非NULL→カウントされる)
・done='N'のレコード　「done='Y' OR NULL」→「FALSE OR NULL」→NULL(→カウントされない)
となります。TRUE/FALSEでなくTRUE/NULLになるのが少々不思議ですが、これも公式ドキュメントにあります。
https://www.postgresql.jp/document/14/html/functions-logical.html
PostgreSQL 14.5文書 > 第9章 関数と演算子 > 9.1. 論理演算子

a	b	a AND b	a OR b	備考
TRUE	TRUE	TRUE	TRUE	
TRUE	FALSE	FALSE	TRUE	
TRUE	NULL	NULL	TRUE	done='Y' OR NULL　の場合
FALSE	FALSE	FALSE	FALSE	
FALSE	NULL	FALSE	NULL	done='N' OR NULL　の場合
NULL	NULL	NULL	NULL	

この完了タスク数のように、計算で求められるデータを「**導出項目**」と言います。そして「**導出項目はテーブルに持たない(列として定義しない)**」というのがDB設計の基本です。ただし以下のような場合は、列として持つこともあります。

・業務上重要なデータであり、かつ画面・帳票で頻繁に使う。

・計算コストが大きく(複数のテーブルを結合する必要がある、など)、応答時間に影響を与える。

完了タスク数はどちらにも当てはまらないため、列にする必要はないのですが、本章後半の題材となってもらうため、ここでは**(あえて)**todoテーブルへ追加します。

5.2 完了タスク数の追加

最初にtodoテーブルへ完了タスク数の列completed_tasksを追加します。このとき初期値として、列追加時点での完了タスク数をセットします。

【リスト5-1】src/main/resources/sql/51_add_completed_to_todo.sql

```
ALTER TABLE todo ADD COLUMN completed_tasks
INTEGER;
UPDATE todo SET completed_tasks = 0;
WITH with_completed AS ( -- ①
  SELECT todo_id, COUNT(done='Y' OR NULL) AS completed
    FROM task GROUP BY todo_id
)
UPDATE todo td -- ②
   SET completed_tasks = tmp.completed
  FROM with_completed tmp
WHERE td.id = tmp.todo_id;
ALTER TABLE todo ALTER COLUMN completed_tasks SET NOT
NULL;
```

①(ToDoのid, 完了したタスク数)からなる一時的な表with_completedを作成する。
②①の結果をUPDATE FROM(SELECT結果でUPDATEする)で設定する。

あわせてTodoエンティティに対応するプロパティを追加します。

【リスト5-2】com.example.todolist.entity.Todo.java

```
public class Todo {
                    :
    private Integer ownerId;

    @Column(name = "completed_tasks")   // 追加
    private Integer completedTasks;   // 追加

    @ManyToOne
```

```
                    :
}
```

さらに完了タスク数をToDo一覧(todoList.html)へ表示するためv_todolist、およびエンティティへ追加します。

【リスト5-3】src/main/resources/sql/50_create_v_todolist.sql

```
DROP  VIEW  IF  EXISTS v_todolist;
CREATE  VIEW v_todolist AS
SELECT  td2.id, td2.owner_id, c.NAME AS cname, g.NAME AS
gname, td2.title,
        td2.importance, td2.urgency, COALESCE(tk.cnt, 0) AS task,
        td2.deadline, td2.done,
        td2.completed_tasks   -- 追加
FROM
(
    SELECT id, owner_id, groups_id, category_code, category_locale,
            title, importance, urgency, deadline, done,
            completed_tasks   -- 追加
      FROM todo
     WHERE groups_id = 0
  UNION
    SELECT td.id, ga.account_id AS owner_id, td.groups_id,
td.category_code, td.category_locale,
            td.title, td.importance, td.urgency, td.deadline, td.done,
            td.completed_tasks   -- 追加
      FROM todo td
      JOIN groups_account ga   ON ga.groups_id = td.groups_id
     WHERE  td.groups_id != 0
) td2
```

```
   JOIN category c ON td2.category_code = c.code AND
td2.category_locale = c.locale
   JOIN groups g     ON td2.groups_id = g.id
   LEFT OUTER JOIN (SELECT    todo_id, COUNT(*) AS cnt
                              FROM    task
                              GROUP    BY todo_id) tk
                    ON    td2.id = tk.todo_id;
```

【リスト5-4】com.example.todolist.entity.Todolist.java

```
public class Todolist {
                    :
      private String done;

      @Column(name = "completed_tasks")    // 追加
      private Integer completedTasks;    // 追加

}
```

これを画面へ表示します。

【リスト5-5】src/main/resources/templates/todoList.html

```
                    :
<!-- 検索結果エリア -->
                    :
   <!-- タスク数 --><!-- 変更    -->
   <td style="text-align: center">
      <span th:text="${todo.completedTasks}"></span>/<span
th:text="${todo.numOfTasks}"></span>
    </td>
```

:

一覧にはタスク数が　完了タスク数 / 総タスク数　の形式で表示されます。しかし新規に登録したものは、完了タスク数が0のままです。この処理は本章後半で解説します。

【図5-1】完了タスク数 / 総タスク数

先に担当者と進捗率を追加します。このうち担当者はリストボックスとします。完成イメージは以下の通りです。

【図5-2】担当者と進捗率を追加したタスク

5.3 担当者/進捗率の追加

まずtaskテーブルに担当者assigned_to、進捗率progressを追加し、対応するプロパティをTask(エンティティ)、TaskData(Formデータ)へ追加します。

【リスト5-6】

src/main/resources/sql/52_add_assign_progress_to_task.sql

```
ALTER TABLE task ADD COLUMN assigned_to INTEGER;
ALTER TABLE task ADD COLUMN progress INTEGER;
UPDATE task SET assigned_to=0, progress = 0;
ALTER TABLE task ALTER COLUMN progress SET NOT NULL;
```

【リスト5-7】com.example.todolist.entity.Task.java

```
public class Task {
                    :
    private String done;

    // ---------- 追加ここから ↓ ↓ ↓ ----------
    @Column(name = "assigned_to")
    private Integer assignedTo;

    @Column(name = "progress")
    private Integer progress;
    // ---------- 追加ここまで ↑ ↑ ↑ ----------
}
```

【リスト5-8】com.example.todolist.form.TaskData.java

```
public class TaskData {
                              :
    private String done;

    private Integer assignedTo;    // 追加
    private String progress;    // 追加
}
```

ToDo入力画面(todoForm.html)とやり取りをするTodoDataでassigned_to, progressの値を設定します(前述の完了タスク数もここで設定)。

【リスト5-9】com.example.todolist.form.TodoData.java

```
public class TodoData {
                    :
    private Integer groupsId;
    private Integer completedTasks = 0;    // 追加
                    :
    //    ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
        todo.setGroups(new Groups(groupsId));
        todo.setCompletedTasks(completedTasks);    // 追加
                    :
            task = new Task(
                    :
                taskData.getDone(),
                taskData.getAssignedTo(),    // 追加
                Integer.parseInt(taskData.getProgress()));    // 追加
```

```
                :
    }

    // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
    public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
                :
        this.groupsId = todo.getGroups().getId();
        this.completedTasks = todo.getCompletedTasks();   // 追加
                :
            this.taskList.add(
                new TaskData(
                :
                task.getDone(),
                task.getAssignedTo(),   // 追加
                "" + task.getProgress()));   // 追加
    }

    // ToDo入力画面新規タスク入力行の内容からエンティティを作成
    public Task toTaskEntity() {
                :
        task.setDeadline(Utils.str2date(newTask.getDeadline()));
        task.setAssignedTo(newTask.getAssignedTo());   // 追加
        task.setProgress(Integer.parseInt(newTask.getProgress()));
// 追加
        return task;
    }
}
```

そしてToDo入力画面へ対応するフォーム部品を追加します。

【リスト5-10】src/main/resources/templates/todoForm.html

```
                    :
  <!-- 更新の場合、Task一覧を表示する -->
  <div th:if="${session.mode == 'update'}">
    <hr style="margin-top: 2em; margin-bottom: 1em;">
    ■Task
    <table>
      <tr>
        <th>id</th>
        <th th:text="#{label.title}"></th>
        <th th:text="#{label.assignedTo}"></th> <!-- 追加 -->
        <th th:text="#{label.progress}"></th> <!-- 追加 -->
        <th th:text="#{label.deadline}"></th>
        <th th:text="#{label.check}"></th>
        <th></th>
      </tr>
      <!-- 登録済みTask -->
                    :
        </td>
        <!-- ========== 追加ここから ↓ ↓ ↓
========================================== -->
        <!-- 担当者 -->
        <td>
          <select th:name="${'taskList[' + stat.index +
'].assignedTo'}">
            <option th:each="a : ${session.assignedToList}"
th:value="${a.id}"
```

```
                    th:text="${a.name}" th:selected="${a.id} ==
${task.assignedTo}"></option>
                </select>
            </td>
            <!-- 進捗率 -->
            <td align="center">
                <input type="text" th:name="${'taskList[' + stat.index +
'].progress'}" size="2"
                    th:value="${task.progress}" style="text-align: right">
                <div th:if="${#fields.hasErrors('taskList[' + stat.index +
'].progress')}"
                    th:errors="*{taskList[__${stat.index}__].progress}"
th:errorclass="red"></div>
            </td>
            <!-- ========== 追加ここまで
↑ ↑ ↑ ========================================
-->
            <!-- 期限 -->
                    :
            <!-- 新規タスク入力行 -->
                    :
            </td>
            <!-- ========== 追加ここから ↓ ↓ ↓
========================================== -->
            <!-- 担当者 -->
            <td>
                <select name="newTask.assignedTo">
                    <option th:each="a : ${session.assignedToList}"
th:value="${a.id}"
```

```
                        th:text="${a.name}" th:selected="${a.id} == *
{newTask.assignedTo}">
                    </option>
                </select>
            </td>
            <!-- 進捗率 -->
            <td align="center">
                <input type="text" name="newTask.progress" size="2"
th:value="*{newTask.progress}"
                       placeholder="0-100" style="text-align: right">
                <div th:if="${#fields.hasErrors('newTask.progress')}"
                     th:errors="*{newTask.progress}" th:errorclass="red">
</div>
            </td>
            <!-- ========== 追加ここまで ↑ ↑ ↑
========================================== -->
            <!-- 期限 -->
                    :
```

【リスト5-11】

src/main/resources/i18n/FixedDisplayStrings_en.properties

```
label.assignedTo=Assigned to
label.progress=Progress(%)
```

【リスト5-12】

src/main/resources/i18n/FixedDisplayStrings_ja.properties

```
label.assignedTo=担当者
label.progress=進捗率(%)
```

このうちタスク担当者の選択肢(assignedToList)は、コントローラーで取得します。その仕様は以下のようにします。

1)グループで共有していないToDo(todo.groups_id = 0)は、デフォルトのみとする。

→登録者のToDoなので、担当者は自分しかいない。選択する必要なし。

2)共有しているToDo(todo.groups_id != 0)は、ToDoと同じグループに所属している人全員とする。

→自分以外の人も担当者に設定できるようにする(ここは「自分だけ」とする考え方もあると思います)。

このように選択可能な担当者は(ToDoのグループで決まるため)、ToDoごとに変わります。そのため選択肢は、ToDo入力画面を表示させるtodoById()の中で行います。

【リスト5-13】com.example.todolist.controller.TodoListController.java

```
        :
import com.example.todolist.repository.GroupsRepository;
        :
public class TodoListController {
                        :
    private final GroupsRepository groupsRepository;    // 追加
                        :
    // ToDo表示
    @GetMapping("/todo/{id}")
    public ModelAndView todoById(@PathVariable(name = "id") int
id, ModelAndView mv,
                        :
            // 表示用データ作成
```

```
            mv.addObject("todoData", new TodoData(todo,
attachedFiles));

            // ---------- 追加ここから ↓ ↓ ↓ ----------
            // 担当者の選択肢
            List<Account> assignedToList = new ArrayList<>();
            assignedToList.add(new Account(0, "---------"));    // ①
            if (todo.getGroups().getId() != 0) { // ②
                // グループで共有しているTodoなので、そのグループに属してい
る人を担当者の選択肢とする
                Groups groups =
groupsRepository.findById(todo.getGroups().getId()).get();    // ③
                assignedToList.addAll(groups.getAccountList());    // ④
            }
            session.setAttribute("assignedToList", assignedToList);
            // ---------- 追加ここまで ↑ ↑ ↑ ----------
            session.setAttribute("mode", "update");
                    :
        }
        :
}
```

①デフォルトの選択肢を追加

・以下のコンストラクターをAccountへ追加します。

【リスト5-14】com.example.todolist.entity.Account.java

```
public class Account {
                    :
    // ---------- 追加ここから ↓ ↓ ↓ ----------
    public Account(Integer id, String name) {
```

```
            super();
            this.id = id;
            this.name = name;
        }
        // ---------- 追加ここまで ↑ ↑ ↑ ----------
}
```

②グループ共有のToDoか？

・todo.groups_id列の値は、todo.getGroups().getId()で取得します。これはTodoでグループ情報を持っているのがGroups型プロパティgroupsだからです。この中のidがtodo.groups_id列に対応しています。

③グループメンバーの取得

・②同様にtodo.getGroups().getId()でtodo.groups_id列の値を取得し、それでグループのメンバーを取得します。

④選択肢に検索結果を追加

・List#addAll(List list)は引数のlistをすべて追加するメソッドです。

> このようにJPAでは、定義した関係(@OneToMnay, @ManyToOne, @ManyToMany)を利用してデータを取得していきます。そのため「どことどこがどのようにつながっているか？」を常に把握しながらコーディングしなければなりません。

最後に進捗率チェック処理を追加します。

【リスト5-15】com.example.todolist.service.TodoService.java

```
public class TodoService {
                    :
    public boolean isValid(TodoData todoData, BindingResult result,
boolean isCreate,
                                    Locale locale) {
```

```
                :
            // Taskのチェック
            List<TaskData> taskList = todoData.getTaskList();
            if (taskList != null) {
                :
                //   ---------- 追加ここから ↓ ↓ ↓ ----------
                // 進捗率(追加)
                if (!taskData.getProgress().matches("^[0-9]+$")) {
                    FieldError fieldError
                        = new FieldError(
                                result.getObjectName(),
                                "taskList[" + n + "].progress",
                                messageSource.getMessage(
                                    "Range.taskData.progress", null,
locale));
                    result.addError(fieldError);
                    ans = false;
                } else {
                    int progress =
Integer.parseInt(taskData.getProgress());
                    if (progress < 0 || 100 < progress) {
                        FieldError fieldError
                        = new FieldError(
                                result.getObjectName(),
                                "taskList[" + n + "].progress",
                                 messageSource.getMessage(
                                     "Range.taskData.progress", null,
locale));
                        result.addError(fieldError);
```

```
                ans = false;
            }
        }
        //   ---------- 追加ここまで ↑ ↑ ↑ ----------

        // タスク期限のyyyy-mm-dd形式チェック
            :
    }
            :
    // ----------------------------------------------------------------------
----------
    // 新規Taskのチェック
    // ----------------------------------------------------------------------
----------
    public boolean isValid(TaskData taskData, BindingResult result,
Locale locale) {
            :
        //   ----- 追加ここから ↓ ↓ ↓ -----
        // 進捗率(追加)
        if (!taskData.getProgress().matches("^[0-9]+$")) {
            FieldError fieldError
                = new FieldError(
                        result.getObjectName(),
                        "newTask.progress",
                        messageSource.getMessage(
                            "Range.taskData.progress", null,
locale));
            result.addError(fieldError);
            ans = false;
```

```
            } else {
                int progress = Integer.parseInt(taskData.getProgress());
                if (progress < 0 || 100 < progress) {
                    FieldError fieldError
                        = new FieldError(
                                result.getObjectName(),
                                "newTask.progress",
                                messageSource.getMessage(
                                    "Range.taskData.progress", null,
locale));
                    result.addError(fieldError);
                    ans = false;
                    }
            }
            //   ----- 追加ここまで ↑ ↑ ↑ -----

            // 期限""ならチェックしない
                    :
        }
}
```

①進捗率の数字チェック

・matches()は、引数の「正規表現」と文字列がマッチすればtrueを返します。

・"^[0-9]+$"は、「数字(0-9)だけで構成されている」という意味です。

【リスト5-16】

src/main/resources/i18n/ValidationMessages_en.properties

```
#progress
Range.taskData.progress=Please enter a value between 0 and 100.
```

【リスト5-17】

src/main/resources/i18n/ValidationMessages_ja.properties

```
#進捗率
Range.taskData.progress=0から100の範囲で入力してください
```

これで「担当者」「進捗率」が入力できるようになります。
残りは完了タスク数の計算処理ですが、ここからが本章の主題です。

5.4トランザクション管理

完了タスク数を設定するのは、ToDo入力画面で以下の操作をしたときです。

【処理１】ToDoの[更新]ボタンをクリックしたとき

→ タスク一覧で完了がチェックされているタスクの数をtodo.completed_tasksへ設定する。

【処理２】タスク一覧の[削除]リンクをクリックしたとき

→ チェックされているタスクなら、todo.completed_tasksから1減じる

【処理３】新規タスクの[登録]ボタンをクリックしたとき

→ チェックされていれば、todo.completed_tasksを1増やす

【図5-3】完了タスク数の再計算が必要な処理

以下どのようなコードになるか見ていきます。

【処理１】ToDo更新時にcompleted_tasksも更新する

上述したように完了がチェックされているタスク数を求め、それをtodoへセットするだけです。

【リスト5-18】com.example.todolist.controller.TodoListController.java

```
    // 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();
            //---------- 追加ここから ↓ ↓ ↓ ----------
            int numOfCompletedTasks
                = (int)todoData.getTaskList().stream()
                    .filter(task->
task.getDone().equals("Y")).count();    // ①
            todo.setCompletedTasks(numOfCompletedTasks);    //
②
            //---------- 追加ここまで ↑ ↑ ↑ ----------

            todoRepository.saveAndFlush(todo);    // ③
                :
        }
```

```
    }
```

①完了タスク数算出

・これもストリームを使わなければ、以下のようなコードになるでしょう。

```
int numOfCompletedTasks = 0;
for (TaskData taskData : todoData.getTaskList()) {
    if (taskData.getDone().equals("Y")) {
        numOfCompletedTasks++;
    }
}
```

②完了タスク数設定

③todo/taskテーブル更新

・Todoエンティティが持っているタスクの内容(Todo.taskList)も、同時にTaskテーブルへ反映されます。

これは大丈夫そうです。

【処理２】タスク削除時、completed_tasksを更新する

削除するタスクがチェックされていれば、そのtodoのcompleted_tasksから1減じます。

```
    // Task削除処理
    @GetMapping("/task/delete")
    public ModelAndView deleteTask(@RequestParam(name =
"task_id") int taskId,
                    :
            Integer accountId =
(Integer)session.getAttribute("accountId");
            // 操作者のToDoか? or Todoと同じグループに所属しているか？
            @SuppressWarnings("unchecked")
            List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
```

```
            if (todo.getOwnerId().equals(accountId) ||
isBelong(groupsList, todo.getGroups())) {

                //      ---------- 追加ここから ↓ ↓ ↓ ----------
                Task task = taskRepository.findById(taskId).get(); // ①
                //      ---------- 追加ここまで ↑ ↑ ↑ ----------

                // Taskを削除
                taskRepository.deleteById(taskId);    // ②

                //      ---------- 追加ここから ↓ ↓ ↓ ----------
                // Done="Y"なら completed_tasks--
                if (task.getDone().equals("Y")) {

todo.setCompletedTasks(todo.getCompletedTasks() - 1);    // ③
                    todoRepository.saveAndFlush(todo);    // ④
                }
                //      ---------- 追加ここまで ↑ ↑ ↑ ----------

                // 削除完了メッセージをセットしてリダイレクト
                    :
            }
            :
      }
```

①[削除]リンクをクリックされたTaskエンティティを取得する。
②当該エンティティを削除する。
③タスクがチェックされていたらcompleted_tasksを-1する。
・画面ではなく、テーブルから読み出したレコードの完了チェックで判断する。
・そのためにTaskを①で取得する。

④Todoエンティティを更新する。

「これもOK」に見えますが、②deleteById()→④saveAndFlush()の間で例外が発生すると、taskテーブルとtodoテーブルが「不整合な状態」となります。試しに②と④の間で故意に例外を発生させてみます。

```
                    :
                // Taskを削除
                taskRepository.deleteById(taskId);       // ②

                boolean isTxTest = true;
                if (isTxTest) {
                    throw new IllegalArgumentException("");       //
例外を発生させる
                }

                //      ---------- 追加ここから ↓ ↓ ↓ ----------
                // Done='Y'なら completed_tasks--
                if (task.getDone().equals("Y")) {

todo.setCompletedTasks(todo.getCompletedTasks() - 1);// ③
                    todoRepository.saveAndFlush(todo); // ④
                }
                //      ---------- 追加ここまで ↑ ↑ ↑ ----------
                    :
```

(1)task.id = 2の完了タスク数:3, 総数:3　→　3/3

ログアウト

件名	重要度	緊急度	期限	チェック	
	-	-	yyyy-mm-dd ～ yyyy-mm-dd	完了	検索

新規追加 PDF出力 Excel出力

id	カテゴリ	グループ	件名	重要度	緊急度	タスク	期限	チェック
1	仕事		UI設計	★★★	★★★	0/0	2023-10-01	
2	レジャー	BBQサークル	定例BBQ大会開催	★★★	★	3/3	2023-10-02	

1 / 1 ページを表示中

←前 1 次→

(2)件名リンクをクリック→完了しているタスクを１つ削除

(3)例外発生→再度ログインし直す

(4)再度ログインしてみると完了タスク数:3, 総数:2になっている(不整合な状態)


ログイン ID:okada(岡田 是則)
ログアウト
件名
重要度
緊急度
期限
チェック
検索
yyyy-mm-dd
yyyy-mm-dd
完了
新規追加
PDF出力 Excel出力
id
カテゴリ
グループ
件名
重要度
緊急度
タスク
期限
チェック
1
仕事
UI設計
★★★
★★★
0/0
2023-10-01
2
レジャー
BBQサークル
定例BBQ大会開催
★★★
★
3/2
2023-10-02
1 / 1 ページを表示中
←前 1 次→

この場合②で削除したタスクがチェックされていても、④を実行する前に例外でメソッドから抜けてしまいます。つまり④は実行されないためtodo.completed_tasksは変わらず、完了タスク数と辻褄があわなくなります。これが「**不整合**」です。

【処理３】タスク追加時、completed_tasksを更新する

タスクがチェックされていれば、Task追加後todo.completed_tasksを+1します。

```
    // Task追加処理
    @PostMapping("/task/create")
    public String createTask(@ModelAttribute TodoData todoData,
BindingResult result, Model model,
                    :
        // 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();
            task.setTodo(todo);
            taskRepository.saveAndFlush(task);    // ①

            //   ---------- 追加ここから ↓ ↓ ↓ ----------
            if (task.getDone().equals("Y")) {

todo.setCompletedTasks(todo.getCompletedTasks() + 1);
                todoRepository.saveAndFlush(todo); // ②
            }
```

```
                //   ---------- 追加ここまで ↑ ↑ ↑ ----------

                // 追加完了メッセージをセットしてリダイレクト
                    :
            } else {
                // エラーあり -> エラーメッセージをセット
                    :
            }
        }
```

①Taskはここで追加する。
②チェックされていたら、ここでTodoのcompleted_tasksを更新する。

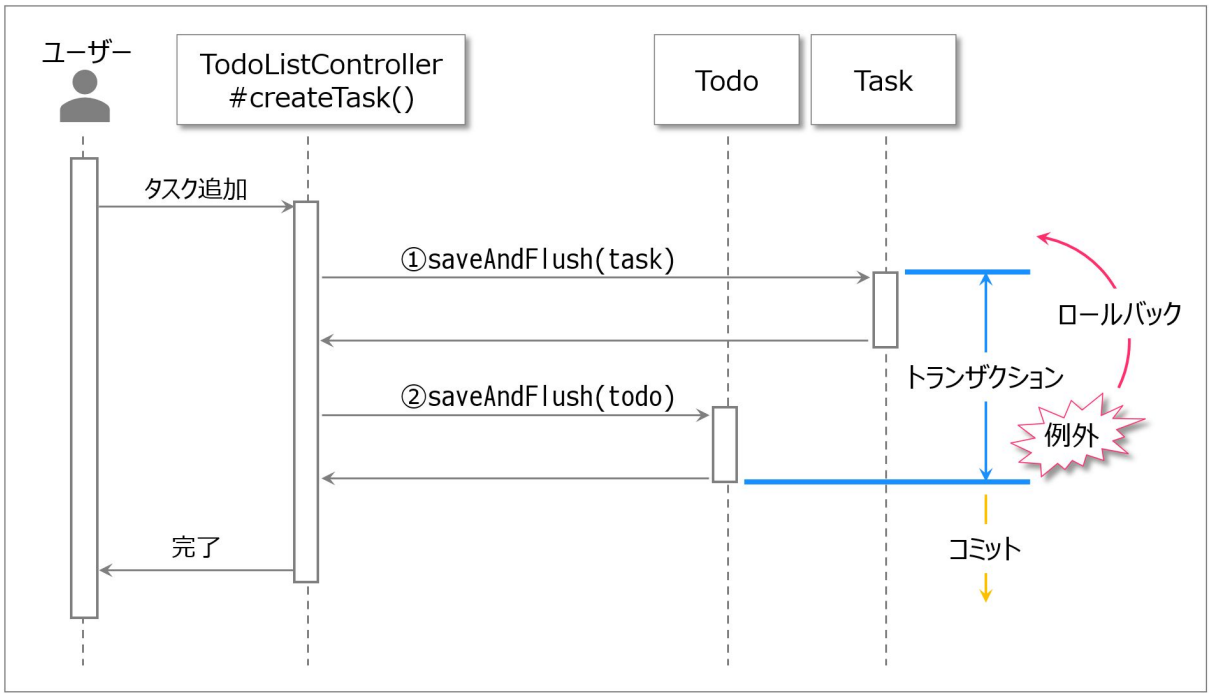
これも①～②間で例外が発生すると「**不整合な状態**」になります。
「そんなこと起こらないだろう」と思いがちですが、実務ではデータのチェック漏れ、コーディング誤り、環境的な問題などにより
・複数テーブルに対する更新系処理(追加、変更、削除)　あるいは
・同一テーブルに対する複数回の更新系処理
が途中で終わってしまった場合、どうリカバーするか考慮し、実装すべきです。これを「**トランザクション管理**」と言います。

「**トランザクション(transaction)**」をわかりやすく言えば「**分けることのできない処理**」になるでしょう。【処理３】の場合、完了タスクを追加するなら①、②とも正常終了しなければなりません。そうでないとテーブルは「不整合な状態」となります。つまり①②はペアであり、①だけ、あるいは②だけ、では成り立ちません。①～②は「分けることのできない処理」であり、これが「**トランザクション**」になります。

またトランザクションには「**原子性**」という考え方があります。これは「**一連の処理(=トランザクション)がすべて完了、または何も行われていない状態にする**」ことで

す。【処理３】なら「①②ともに完了」が前者であり、「どちらも行わない」が後者です。

　では例外が発生し「①は成功、②で失敗」となったら、どうすればよいでしょう？この場合は「データベースのトランザクション管理機能」を使い、①実行前に戻します。つまり一度実行した①を無かったことにします。これを「**ロールバック(rollback)**」と言います。一方「①②とも成功」なら、その結果を確定さます。これを「**コミット(commit)**」と言います。



【図5-4】トランザクションの原子性

> 原子性はトランザクション処理に求められる**ACID(アシッド)特性**の１つです。ACIDは**A**tomicity(原子性)、**C**onsistency(一貫性)、**I**solation(独立性)、**D**urability(耐久性)の頭文字をつなぎ合わせたものです。興味がある方は調べてみてください。

> このうちIsolation(独立性)は**トランザクション分離レベル**という考え方と密接に関係しています。これはトランザクションを同時に実行した場合、相互に及ぼし

> あう影響度合いを定義したもので、これも非常に重要な考えです。興味がある方は調べてみてください。

5.5 Springのトランザクション管理

ここからはSpring Bootのトランザクション管理機能を使い、処理をトランザクション化していきます。
以下はその手順です。

【手順１】
・トランザクションとする処理でpublicメソッドを作成し、**@Transactional**を付与する。
【手順２】
・どういう例外が発生したらロールバックするか？を①の@Transactional内に指定する。
→デフォルトは、非検査例外(RuntimeException、およびそのサブクラス)発生でロールバックする。
→検査例外(Exception、およびそのサブクラスでRuntimeExceptionを継承しない)は、ロールバックされない。
【手順３】
・【手順１】のメソッドを「**別クラス**」から呼び出す。
→例. メソッドをサービスクラスに定義し、それをコントローラークラスから呼び出す。
・メソッド内で指定した例外が発生すると**自動的にロールバック**される。
・例外が発生しなければ、(メソッド終了時)**自動的ににコミット**される。

ではトランザクションを使い「不整合な状態」を防げるようにしていきます。
まず【処理２】タスク削除の場合です。

■【リスト5-19】
com.example.todolist.controller.TodoListController.java

```
        // Task削除処理
```

```
    @GetMapping("/task/delete")
    public ModelAndView deleteTask(@RequestParam(name =
"task_id") int taskId,
                                   :
            // 操作者のToDoか? or Todoと同じグループに所属しているか？
            @SuppressWarnings("unchecked")
            List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
            if (todo.getOwnerId().equals(accountId) ||
isBelong(groupsList, todo.getGroups())) {
                //   ---------- 変更ここから ↓ ↓ ↓ ----------
                try {
                    // この中でTaskを削除し、Todoのcompleted_tasksを再
計算する

todoService.deleteTaskAndRecalcCompletedTasks(taskId, todo);

                    // 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);

                } catch (Exception e) {
                    e.printStackTrace();
                    // Rollbackされている
                    operationError(mv, redirectAttributes, locale);
                }
```

```
            //  ---------- 変更ここまで ↑ ↑ ↑ ----------
        } else {
            // 操作者のものでない
            operationError(mv, redirectAttributes, locale);
        }
          :
    }
```

TodoService#deleteTaskAndRecalcCompletedTasks()がトランザクション化した処理です。といっても、呼び出す側から見ると、ただのメソッドです。定義は以下のようになっています。

【リスト5-20】com.example.todolist.service.TodoService.java

```
                    :
import org.springframework.transaction.annotation.Transactional;
import com.example.todolist.entity.Task;
import com.example.todolist.entity.Todo;
import com.example.todolist.repository.TaskRepository;
import com.example.todolist.repository.TodoRepository;
                    :
public class TodoService {
                    :
    private final TodoRepository todoRepository;
    private final TaskRepository taskRepository;
                    :
    // ----------------------------------------------------------------
--------
    // Task削除 -> todo.completed_tasks再計算
    // ----------------------------------------------------------------
--------
```

```
    @Transactional(rollbackForClassName = { "Exception" })      //
①
    public void deleteTaskAndRecalcCompletedTasks(Integer taskId,
Todo todo) {
        Task task = taskRepository.findById(taskId).get();
        taskRepository.deleteById(task.getId());      // ②

        if (task.getDone().equals("Y")) {
            todo.setCompletedTasks(todo.getCompletedTasks() -
1);
            todoRepository.saveAndFlush(todo);      // ③

        }
        //throw new IllegalArgumentException("");      //④
    }
}
```

①トランザクション宣言(**@Transactional**)

・@Transactionalは、このメソッドをトランザクションとして管理します。

・rollbackForClassNameで、発生したらロールバックする例外クラスを指定します。

→ここで指定した例外クラスのサブクラスも対象となる。

→Exceptionは、すべての例外のスーパークラスなので、何らかの例外が発生したらロールバックする。

②タスク削除

③ToDo更新

・②～③が上述した「分けられない処理」です。

・③完了時(=メソッド終了時)、コミットされます。

> @Transactionalを使うときは、org.springframework.transaction.annotation.Transactionalをimportしてください。jakarta.transaction.TransactionalではrollbackForClassNameが指定できないので注意してください。

> ロールバックする例外をrollbackFor属性で指定する方法もあります。この場合は、次のように書きます。
> @Transactional(rollbackFor=Exception.class)
> この他にも@Transactionalには、上述した分離レベルに関する属性などもあります。興味がある方は調べてみてください。

例外が発生したらテーブルの状態は、このメソッドの開始時点まで戻されます(ロールバック)。これは④の行をアンコメント化(行頭の//を削除)し、例外を発生させると②③とも行われていない(=状態が変化していない)ことで確認できます。

【処理３】タスク追加の場合も同様です。以下のようなメソッドをTodoServiceに作成し、TodoListControllerから呼び出します。

【リスト5-21】com.example.todolist.service.TodoService.java

```
    // ------------------------------------------------------------------------
    // Task追加 -> todo.completed_tasks再計算
    // ------------------------------------------------------------------------
    @Transactional(rollbackForClassName = { "Exception" })
    public void createTaskAndRecalcCompletedTasks(Task task, Integer todoId) {
        taskRepository.saveAndFlush(task);
```

```
            if (task.getDone().equals("Y")) {
                Todo todo = todoRepository.findById(todoId).get();
                todo.setCompletedTasks(todo.getCompletedTasks() +
1);
                todoRepository.saveAndFlush(todo);
            }
        }
```

【リスト5-22】com.example.todolist.controller.TodoListController.java

```
    // Task追加処理
    @PostMapping("/task/create")
    public String createTask(@ModelAttribute TodoData todoData,
BindingResult result, Model model,
                             :
            // 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();
                task.setTodo(todo);
                //   ---------- 変更ここから ↓ ↓ ↓ ----------
                try {
                    // この中でTaskを追加し、Todoのcompleted_tasksを再
計算する

todoService.createTaskAndRecalcCompletedTasks(task,
tododata.getId());
```

```
                    // 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();

                } catch (Exception e) {
                    e.printStackTrace();
                    // Rollbackされている
                    return "redirect:/error";
                }
                //   ---------- 変更ここまで ↑ ↑ ↑ ----------
            } else {
                // エラーあり -> エラーメッセージをセット
                        :
            }
        }
```

6. 楽観ロックによる排他制御(1)

プロジェクト名	Todolist21
作成ファイル	ー(なし)
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Task.java com.example.todolist.form.TasDatak.java com.example.todolist.form.TodoData.java src/main/resources/templates/todoForm.html src/main/resources/i18n/OperationMessages_en.properties src/main/resources/i18n/OperationMessages_ja.properties
SQLファイル	src/main/resources/sql/60_add_version_to_task.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「進捗率おかしい！」
自分「！？」
先輩「このBBQ大会の「参加者募集HP作成」ってタスクなんだけど...」
自分「はい」
先輩「俺はモック作ったから+50%、ヨセミテ支社は掲載する写真用意したから+20%した」
自分「はい」
先輩「それなのに20%っておかしいだろう？おれの50%どこ行った？」
自分「!?(つーか、どこでBBQやるつもりなの？まさかヨセミテ？)」
:

その後の調べで先輩とヨセミテ支社の某さんは、ほぼ同時に進捗率を0→50%, 0→20%としていたことが判明。

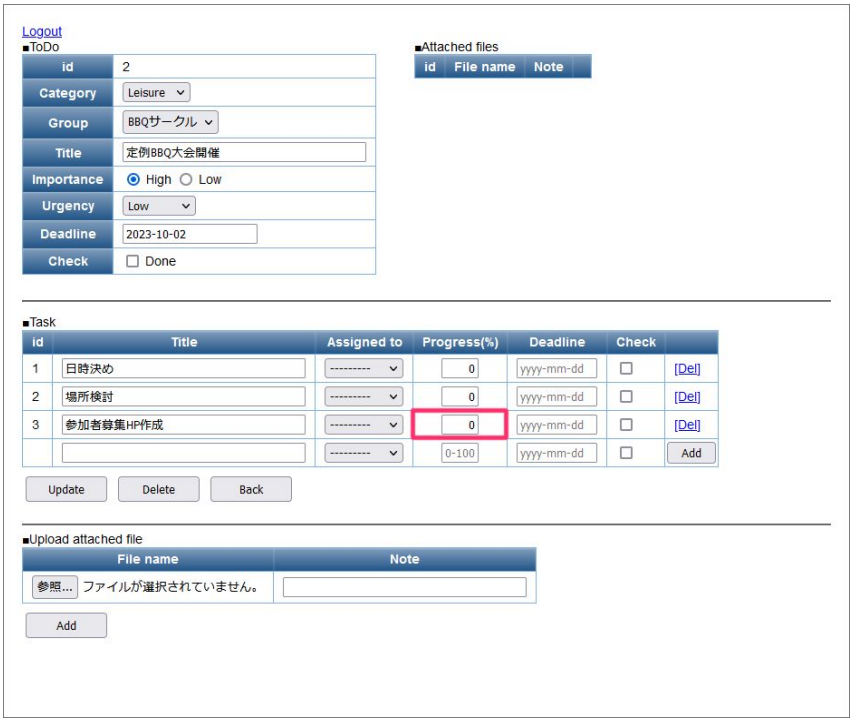
確かに先輩の50%が消えている...。

6.1 同時実行制御

この現象は、以下のように異なるブラウザを使って簡単に再現できます。実際にChromeを日本支社、Firefox(ブラウザのロケール="en_US")をヨセミテ支社にみたて、上記の操作をしてみます。

(1)Chrome(=日本支社)で「定例BBQ大会開催」のToDoを開く。

(2)FireFox(=ヨセミテ支社)でも同じToDoを開く。

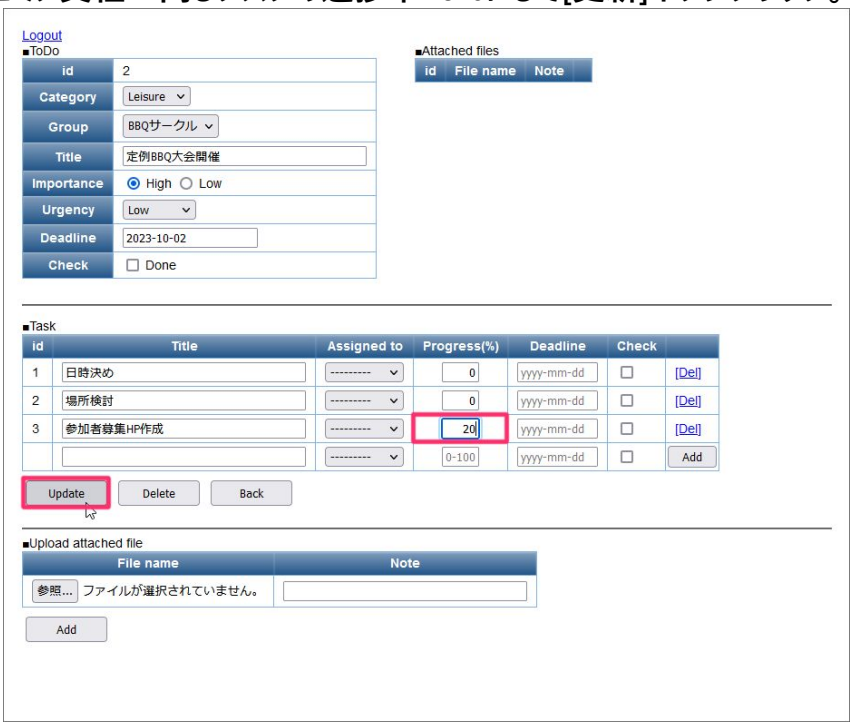


(3)日本支社：タスク「参加者募集HP作成」の進捗率を50%にして[更新]ボタンクリック。

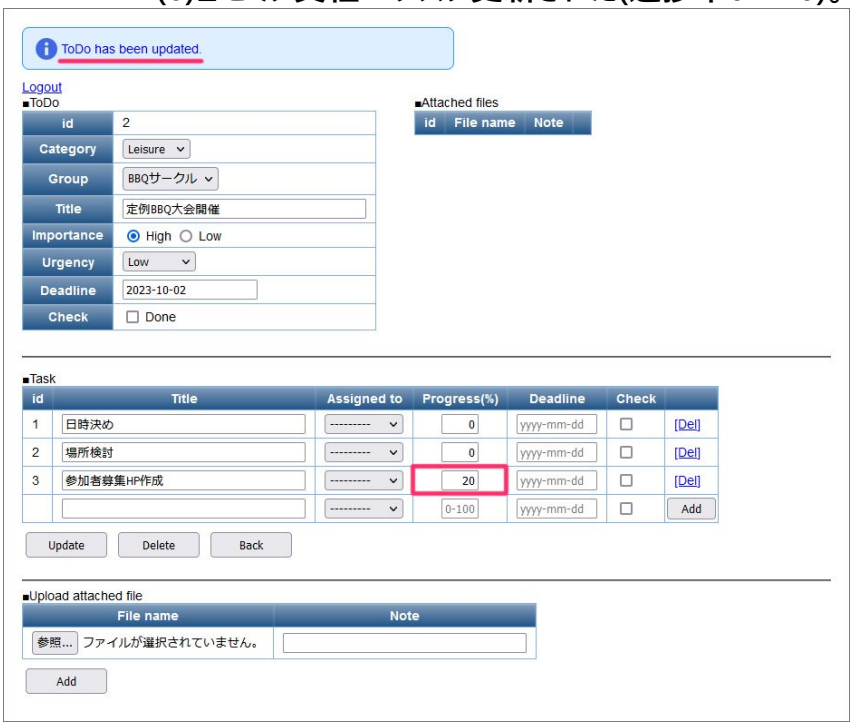
(4)日本支社：タスク更新された(進捗率0→50)。


ToDoを更新しました。
ログアウト
■ToDo
id 2
カテゴリ レジャー
グループ BBQサークル
件名 定例BBQ大会開催
重要度 高い 低い
緊急度 低
期限 2023-10-02
チェック 完了
■添付ファイル
id ファイル名 メモ
■Task
id 件名 担当者 進捗率(%) 期限 チェック
1 日時決め -------- 0 yyyy-mm-dd [削除]
2 場所検討 -------- 0 yyyy-mm-dd [削除]
3 参加者募集HP作成 -------- 50 yyyy-mm-dd [削除]
-------- 0-100 yyyy-mm-dd 登録
更新
削除
戻る
■添付ファイル登録
ファイル名
メモ
ファイルを選択 選択されていません
登録

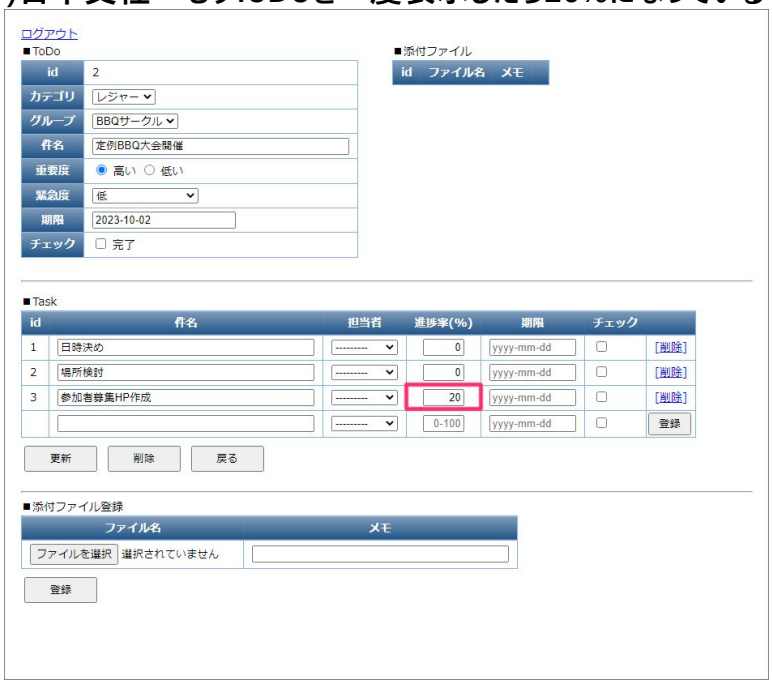
(5)ヨセミテ支社：同じタスクの進捗率20%にして[更新]ボタンクリック。



(6)ヨセミテ支社：タスク更新された(進捗率0→20)。



(7)日本支社：もうToDoを一度表示したら20%になっている→入力した50%が消えた?



これは「後から更新したもの」が最終状態になる、ということです。この場合50+20=70%となるようにしたいなら、「**排他制御**」「**ロック**」という考え方が必要になります。

> 一方「後勝ちでかまわない」という、考え方もあります。その場合、本章のように変更する必要はありません。
> 以下解説する「排他制御」「ロック」は、「**もしデータが変更されていることを知ったら、行動を変える可能性がある**」という場合適用します。上述の場合、ヨセミテ支社は、日本支社が0→50%としたことを知りません。そのため0→20%としました。もし「50%になっている」、ということを知ったら20%ではなく70%とするでしょうか？「する」場合が「行動を変える」ということであり、本章の内容のような処理を実装する必要があります。

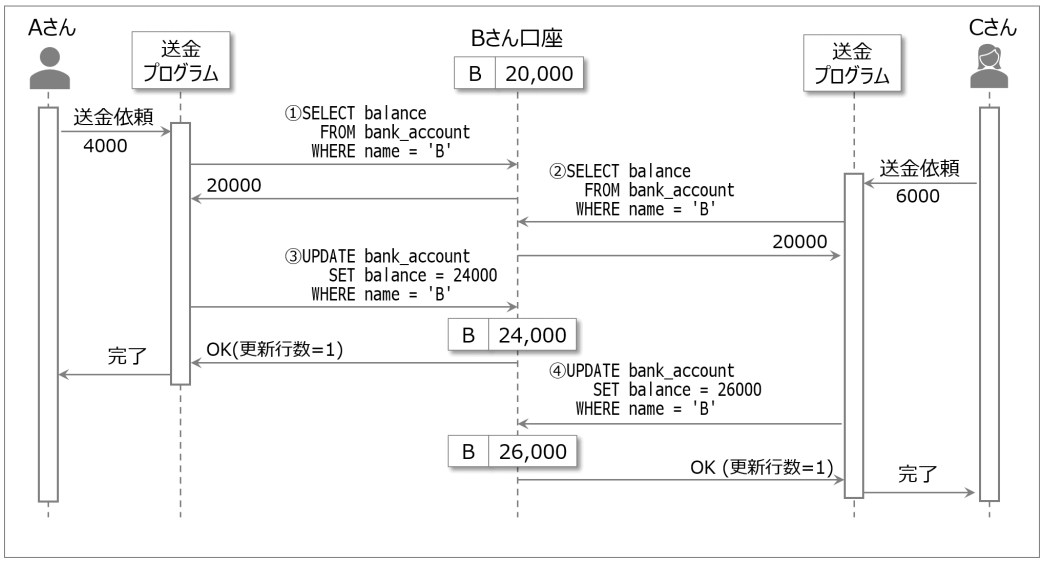
6.2 排他制御

まず「**排他制御**」から行きます。これは「**同時に実行されると不都合が起こる場合、それを回避する仕組み**」のことです。上記の50+20が20になったのは、この「同時実行による不都合」の典型です。それをもうちょっと別な例で説明します。

下図はBさんの口座へAさんが4,000円、Cさんが6,000円それぞれ送金する処理の例です。
ここでテーブルbank_account(銀行口座)は、シンプルにname(氏名;主キー), balance(残高)のみとします。

bank_account

name	balance
B	20000



【図6-1】不整合が発生する処理

①Aさん側がBさんの口座残高(=20,000)を取得。
②Cさん側がBさんの口座残高(=20,000)を取得。
③Aさん側がBさんの口座残高(=24,000=20,000+4,000)を更新。
④Cさん側がBさんの口座残高(=26,000=20,000+6,000)を更新。

この例でも、30,000円になるはずの残高が26,000円となってます。原因は直感的に「**処理が重なったから**」とわかるでしょう。前章の「トランザクション」の考え方で言えば、①③、②④がそれ

ぞれトランザクション(分けることのできない処理)です。これが重ならなければ不都合が起きません。たとえば①→③→②→④、あるいは②→④→①→③なら口座残高は30,000円になります。簡単に言えば、「排他制御」は、こういった**同時実行による不都合を起こさないようにすること**です。

その具体的な仕組みですが、一般には「ロック」という方法が使われます。ロックには「**悲観ロック**」「**楽観ロック**」、さらにはっきりとした名前はないのですが「業務ロジックによるロック」もあります。このうち本書では「楽観ロック」を使います。

> 興味がある方は楽観ロック以外についても調べてみてください。

6.3 楽観ロック

楽観ロックは、以下のようなシステムで採用されることが多いようです。
・同時に実行される確率が低い
・やり直しを許容してもらえる

方法としては以下のような感じになります。
・自分が更新しようとしているデータが、他の人に更新されていないことを確認した上で更新する。
・誰かに更新されていたら、最初からやり直す。

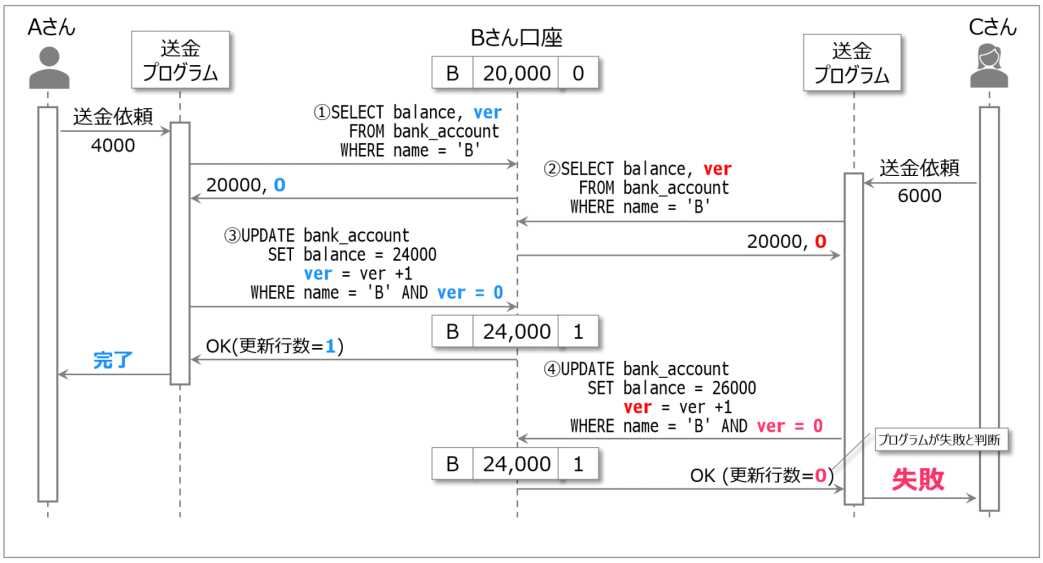
上記の例で言えば、Cさん側が②で検索し、④で更新しようとしたデータは、Aさん側が③で更新しています。これを検知してやり直せれば、不都合は回避できます。

ではどうやって「先に更新されたこと」(競合)を検出するか？というと、それ専用の列をテーブルへ追加します。一般的にはversion/verといった数値項目とすることが多いようです。ここでもそれにならい、bank_accountへverを追加します。

bank_account

name	balance	ver
B	20000	0

そして処理を以下のように変更します
1. 更新対象データを読み込むときverも取得する。
2. 更新時
   1)読み込んだ時のverを更新対象レコードの条件に含める。
   2)verを+1する。
   3)更新できた行数をチェックし、成功/失敗を判断する。



【図6-2】楽観ロックによる不整合の回避

①Aさん側がBさんの口座残高(=20,000)とver(=0)を取得。
②Cさん側がBさんの口座残高(=20,000)とver(=0)を取得。
③Aさん側がBさんの口座残高(=24,000)とver(=1)を更新。
→name = 'B' AND ver = 0に該当するレコードは1件存在するのでそれを更新する。
→UPDATE文の戻り値(=更新行数)は1
→送金成功
④Cさん側がBさんの口座残高(=26,000)とver(=1)を更新。
→name = 'B' AND ver = 0に該当するレコードは、この時点で存在しないので、更新対象レコード・無
→UPDATE文の戻り値(=更新行数)は0 … ただし、エラーにはならない
→送金失敗
→失敗を通知(or 自動的にリトライ)
これでCさん側が②からやり直せば、Bさんの口座残高は30,000になります。

UPDATE文は、「更新した行数」を返します。これはpsqlでUPDATE文を実行したときに表示される数値でもあります。

```
tododb=> UPDATE todo SET urgency=1 WHERE id=1;
UPDATE 1
tododb=> UPDATE todo SET urgency=1 WHERE id=0;
UPDATE 0
tododb=>
```

そしてbank_accountの主キーはnameなので、更新行数は0/1のどちらかです。0のときは、他の人が先に更新した(あるいはレコードが削除された)、つまりトランザクションが重なり(=競合)、不整合な状態になってしまうことがわかります。この場合、処理を最初からやり直せばいいわけです。

このように楽観ロックは簡単に実現できますが、対象テーブル(上例ではbank_account)に対するすべての更新系処理(追加、変更、削除)へ、このルール(verで制御する)を実装する必要があります。漏れがあると「いつのまにか不整合になっていた」、という情報システムとして最悪の事態を招く恐れがあります。そこでSpring Bootのアノテーションを利用します。

6.4 @Versionによる楽観ロック

Spring Bootでは、**@Version**というアノテーションで楽観ロックを実現できます。しかも付与するのがエンティティ(=テーブル)側であり、そのエンティティに対する更新系処理に上記のルールが自動的に適用されます。これをToDoアプリのTask更新へ取り入れていきます。

まずtaskテーブルへ楽観ロック用の列(version)を追加し、エンティティにも対応するプロパティを追加します。

【リスト6-1】src/main/resources/sql/60_add_version_to_task.sql

```
ALTER TABLE task    ADD COLUMN version    INTEGER;
UPDATE task    SET    version=0;
ALTER TABLE task    ALTER COLUMN version    SET NOT NULL;
```

【リスト6-2】com.example.todolist.entity.Task.java

```
                    :
import jakarta.persistence.Version;
                    :
public class Task {
                    :
        private Integer progress;

        //    ---------- 追加ここから ↓ ↓ ↓ ----------
        @Column(name = "version")
        @Version    // ①
        private Integer version;
        //    ---------- 追加ここまで ↑ ↑ ↑ ----------
}
```

①楽観ロック用プロパティ(@Version)

・このプロパティを使って楽観ロックを行います。

【図6-2】のように、入力したレコードのversionは更新時に必要です。入力/更新処理は別リクエストになるため、versionをセッションへ保存するやり方もあります。しかしここでは、versionの値

を確認しやすくするため、画面を経由させることとし、上記プロパティに対応する項目を
TaskData/TodoDataへ追加します。

【リスト6-3】com.example.todolist.form.TasDatak.java

```
public class TaskData {
                    :
    private String progress;

    private Integer version = 0;    // 追加
}
```

【リスト6-4】com.example.todolist.form.TodoData.java

```
public class TodoData {
                    :
    //    ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
        task = new Task(
                taskData.getId(),
                        :
                Integer.parseInt(taskData.getProgress()),
                taskData.getVersion()); //    追加

        //    Todoと関連付け
        todo.addTask(task);
                    :
    // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
    public TodoData(Todo todo, List<AttachedFile> attachedFiles) {
                    :
        // 登録済Task一覧
        this.taskList = new ArrayList<>();
        String dt;
        for (Task task : todo.getTaskList()) {
            dt = Utils.date2str(task.getDeadline());
            this.taskList.add(
```

```
                        new TaskData(
                                task.getId(),
                                   :
                                "" + task.getProgress(),
                                task.getVersion()));    // 追加
            }
                        :
    // ToDo入力画面(todoForm.html)新規タスク入力行の内容からエンティティを作成
    public Task toTaskEntity() {
                        :
        task.setProgress(Integer.parseInt(newTask.getProgress()));
        task.setProgress(Integer.parseInt(newTask.getProgress()));
        task.setVersion(newTask.getVersion());    // 追加

        return task;
    }
}
```

ただしversionは画面表示する必要は無いので、非表示項目(type="hidden")とします。

【リスト6-5】src/main/resources/templates/todoForm.html

```
  <!-- 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="${task.id}" />
      <!-- 競合検出のために必要 -->  <!-- 追加 -->
      <input type="hidden" th:name="${'taskList[' + stat.index + '].version'}"
             th:value="${task.version}" />
    </td>
```

ここまでで既存のタスク行を変更して[更新]ボタンをクリックすると、task.versionが(自動的に)+1されます。

```
tododb=> SELECT* FROM task;
 id | todo_id |      title       | deadline | done | assigned_to | progress | version
----+---------+------------------+----------+------+-------------+----------+---------
  1 |       2 | 日時決め         |          | N    |           0 |        0 |       0
  2 |       2 | 場所検討         |          | N    |           0 |        0 |       0
  3 |       2 | 参加者募集HP作成 |          | N    |           0 |        0 |       0
(3 行)
```

↓

ここで[CTRL]+Uを押下すると、HTMLが表示されversionの値を確認できます。

↓

```
tododb=> SELECT* FROM task;
 id | todo_id |      title       | deadline | done | assigned_to | progress | version
----+---------+------------------+----------+------+-------------+----------+---------
  2 |       2 | 場所検討         |          | N    |           0 |        0 |       0
  3 |       2 | 参加者募集HP作成 |          | N    |           0 |        0 |       0
  1 |       2 | 日時決め         |          | N    |           0 |       10 |       1
(3 行)
```

さらにこの状態から進捗率を10→20へ変更すると、以下のようなUPDATE文が実行されます。

■STSのコンソールに表示されたUPDATE文の情報

```
update task set assigned_to=?, deadline=?, done=?, progress=?, title=?,
todo_id=?, version=?
where id=? and version=?
binding parameter [1] as [INTEGER] - [0]
binding parameter [2] as [DATE] - [null]
binding parameter [3] as [VARCHAR] - [N]
binding parameter [4] as [INTEGER] - [20]
binding parameter [5] as [VARCHAR] - [日時決め]
binding parameter [6] as [INTEGER] - [2]
binding parameter [7] as [INTEGER] - [2]   ←   新しいversion値
binding parameter [8] as [INTEGER] - [1]
binding parameter [9] as [INTEGER] - [1]   ←   更新前のversion値
```

↓

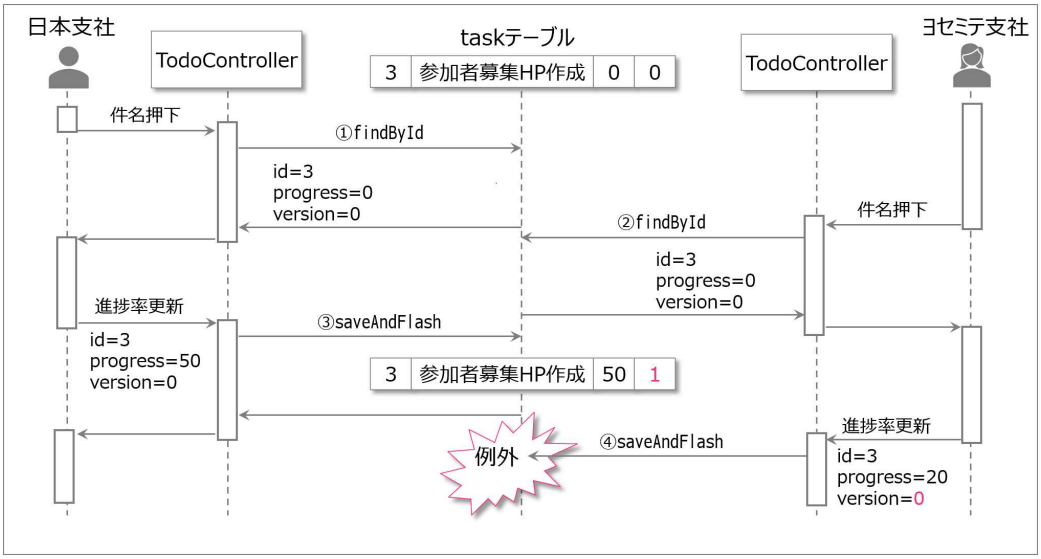
■実際のUPDATE文

```
update   task
   set   assigned_to=0,
         deadline=null,
         done='N',
         progress=20,
         title='日時決め',
         todo_id=2,
         version=2
 where   id=1
   and   version=1
```

このようにversionが使われていることがわかります。このうち「更新レコードのversion値」が、【リスト6-4】で画面を経由させた読み出し時のversionです。

> [更新]ボタンは、すべてのタスク行を更新対象としますが、実際に更新されるのは、画面でデータが変更されたものだけです(変更されていないものは、task.versionが変わらないことからわかります)。これはSpring Data JPAが、変更されたレコードに対してのみUPDATE文を実行するためです。

これで同じタスクを更新すると、後から実行した側には例外が発生します。



【図6-3】@Versionによる楽観ロック

■例外メッセージ例(xはtask.id)

```
org.hibernate.StaleObjectStateException: Row was updated or deleted by another
transaction (or unsaved-value mapping was incorrect) :
[com.example.todolist.entity.Task#x]
    :
```

「(更新しようとした)行が他のトランザクションによって更新または削除されていた」という意味です。この発生個所はTodoListControllerの以下の部分です。

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
BindingResult result,
                    :
        if (!result.hasErrors() && isValid) {
            // エラーなし -> 更新
                    :
            todo.setCompletedTasks(numOfCompletedTasks);
            todoRepository.saveAndFlush(todo);       // ★ここで例外をスローしている
```

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

そこでこの例外が発生したらcatchして「他の人が先に更新した」というメッセージとともに、最新の状態を表示するようにします。

> 先ほどのメッセージではorg.hibernate.StaleObjectStateExceptionが発生したように見えますが、実際にはorg.springframework.orm.ObjectOptimisticLockingFailureExceptionがスローされているので、こちらをcatchします。

```
org.springframework.orm.ObjectOptimisticLockingFailureException: Row was
updated or deleted by
another transaction (or unsaved-value mapping was incorrect) :
[com.example.todolist.entity.
Task#x] at
```

【リスト6-6】com.example.todolist.controller.TodoListController.java

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute @Validated TodoData todoData,
BindingResult result,
                    :
        if (!result.hasErrors() && isValid) {
            // エラーなし -> 更新
            Todo todo = todoData.toEntity();
                :
            todo.setCompletedTasks(numOfCompletedTasks);
```

```
            //   ---------- 変更ここから ↓ ↓ ↓ ----------
            try {
                todoRepository.saveAndFlush(todo);      // ★ここで競合を検出

                // 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));
                return "redirect:/todo/" + todo.getId();

            } catch
(org.springframework.orm.ObjectOptimisticLockingFailureException e) {
                //      競合(=誰かが先に更新 or 削除したとき)
                //       ->      競合メッセージをセットしてリダイレクト
                String msg = messageSource

.getMessage("msg.w.optimistic_locking_failure", null, locale);
                redirectAttributes.addFlashAttribute("msg", new OpMsg("W",
msg));
                return "redirect:/todo/" + todo.getId();
            }
            //   ---------- 変更ここまで ↑ ↑ ↑ ----------
        } else {
            // エラーあり -> エラーメッセージをセット
                :
        }
    }
```

【リスト6-7】src/main/resources/i18n/OperationMessages_en.properties

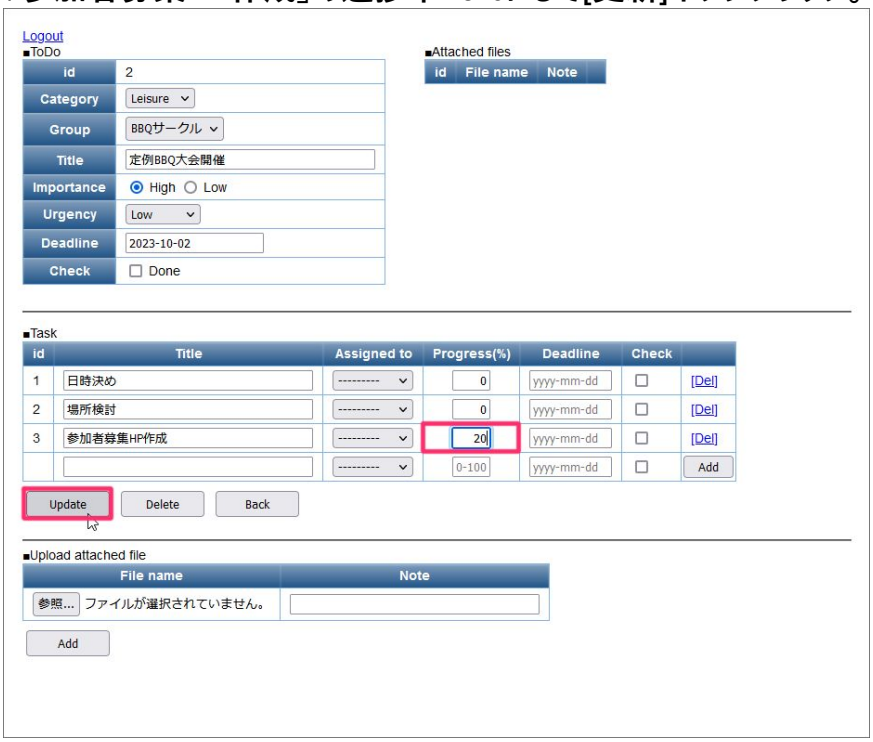
```
msg.w.optimistic_locking_failure=Todo/Task has been updated.
```

【リスト6-8】src/main/resources/i18n/OperationMessages_ja.properties

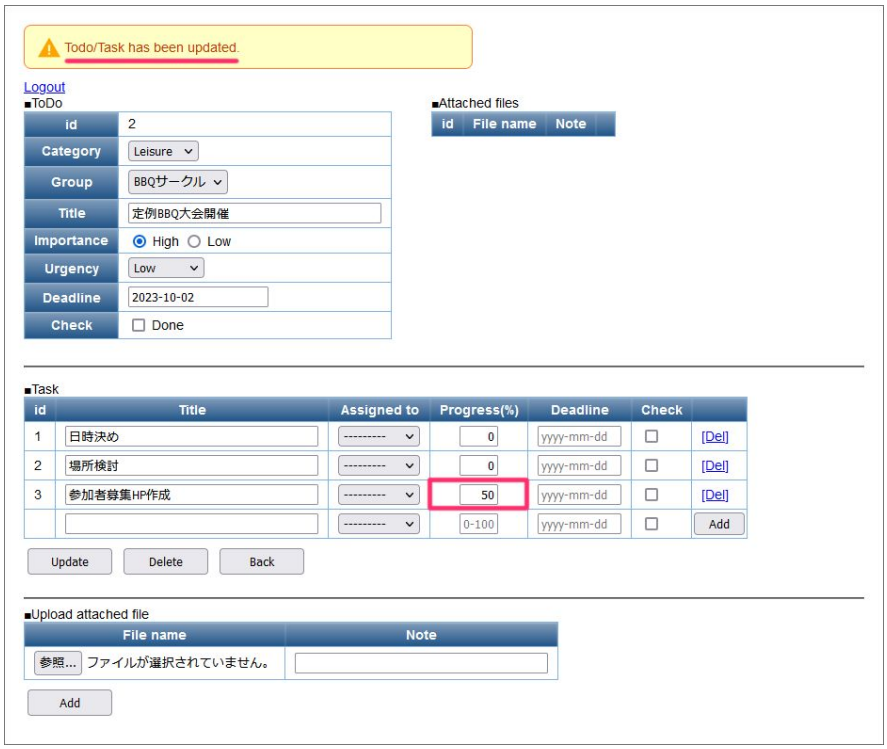
```
msg.w.optimistic_locking_failure=Todo/Taskが更新されていたので再表示します。
```

これで同一タスクへの競合を検出できるようになります。

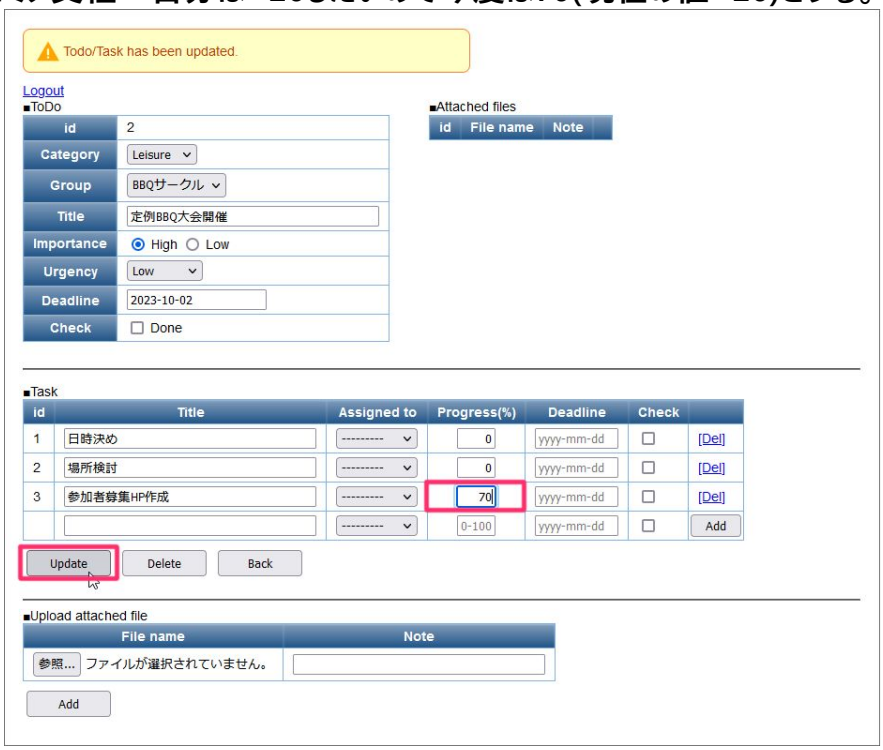
(5)ヨセミテ支社：「参加者募集HP作成」の進捗率20%にして[更新]ボタンクリック。



(6')ヨセミテ支社：他の人が更新した旨のメッセージと現在の値を表示

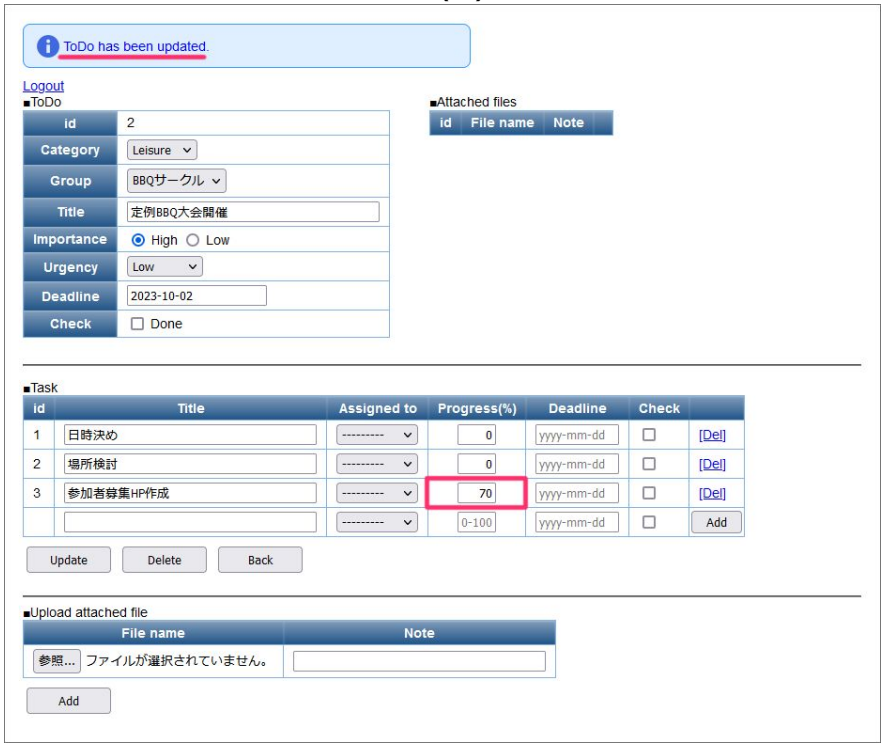


(7')ヨセミテ支社：自分は+20したいので今度は70(現在の値+20)とする。



> 「楽観ロック」が適しているのは、競合が発生しても、こういった再入力が許容される場合です。「ユーザーに再入力させたくない」ならプログラム内部でリトライする、プログラムロジックで「業務ロック」をかける、といった別な方策が必要となるでしょう。

(8')ヨセミテ支社：70へ更新完了



(9')日本支社で見ても70

　これでタスク同士の競合を検出・回避できるようになりました。しかし「Todo削除」×「Task更新」など、まだ対応できていない組み合わせもあります。次章ではToDoにも楽観ロック(@Version)を導入、これらへ対処します。

7. 楽観ロックによる排他制御(2)

プロジェクト名	Todolist22
作成ファイル	ー(なし)
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java src/main/resources/templates/todoForm.html src/main/resources/i18n/OperationMessages_en.properties src/main/resources/i18n/OperationMessages_ja.properties
SQLファイル	src/main/resources/sql/70_add_version_to_task.sql

本章ではTodoにも楽観ロックを導入しますが、その前にTodoの更新・削除とTaskの更新・削除が重なったとき、どうするか？を決めておきます。処理的には色々考えられますが、ここでは以下のようにします。なお追加は、同時に起こってもidが別となるため考慮不要とします。

【表7-1】処理パターン１

#	先Todo	後Todo	遷移先	表示メッセージ
①	U	U	(Todo入力画面のまま)	「更新されていました」

②	U	D	Todo一覧画面	「削除しました」(これまで通り)
③	D	U	Todo一覧画面	「削除されていました」
④	D	D	Todo一覧画面	「削除されていました」

U:更新(UPDATE), D:削除(DELETE)

【表7-2】処理パターン２

#	先 Task	後 Todo	遷移先	表示メッセージ
⑤	U	U	(Todo入力画面のまま)	「更新されていました」
⑥	U	D	Todo一覧画面	「削除しました」(これまで通り)
⑦	D	U	(Todo入力画面のまま)	「削除されていました」
⑧	D	D	Todo一覧画面	「削除しました」(これまで通り)

【表7-3】処理パターン３

#	前 Todo	後 Task	遷移先	表示メッセージ
★1	U	U	(Todo入力画面のまま)	「更新されていました」
★2	U	D	(Todo入力画面のまま)	「削除しました」(これまで通り)
★3	D	U	Todo一覧画面	「削除されていました」
★4	D	D	Todo一覧画面	「削除されていました」

【表7-4】処理パターン４

#	前 Task	後 Task	遷移先	表示メッセージ
★5	U	U	(Todo入力画面のまま)	「更新されていました」
★6	U	D	(Todo入力画面のまま)	「削除しました」(これまで通り)
★7	D	U	(Todo入力画面のまま)	「削除されていました」
★8	D	D	(Todo入力画面のまま)	「削除されていました」

組み合わせ的には

先の操作({Todo, Task} x {更新, 削除} = 4通り)

x 後の操作({Todo, Task} x {更新, 削除} = 4通り) = 16通り

あり得ます。たとえば★１は「更新しようとしたTaskは、誰かが先にその関連するTodoを更新していた」という場合です。

本章では上表の結果となるよう、プログラムを変更していきます。

> 本来であれば上記16パターンについて解説すべきと思いますが、紙面の都合上割愛します。興味がある方は、なぜ上記の結果になるか、コードを調べてみてください。

> ②をOKとする(従来通りとする)、などの考え方もあります。これは「何が正しいか？」ではなく、「ユーザーがどういった結果・操作を望んでいるか？」に依存します。実務では設計段階でユーザーに確認して決める部分です。

> 実務では設計時点で考えられるパターンをすべて列挙し、どのようにすべきか？を事前に洗い出すことが大切です。

7.1 @Version(Todo)

まずTaskエンティティと同様に、楽観ロックに必要な定義を追加します。

【リスト7-1】src/main/resources/sql/70_add_version_to_todo.sql

```
ALTER TABLE todo ADD COLUMN version INTEGER;
UPDATE todo SET version=0;
ALTER TABLE todo ALTER COLUMN version SET NOT NULL;
```

【リスト7-2】com.example.todolist.entity.Todo.java

```
                    :
import jakarta.persistence.Version;
                    :
public class Todo {
                    :
    private Integer completedTasks;

    //  ---------- 追加ここから ↓ ↓ ↓ ----------
    @Column(name = "version")
    @Version
    private Integer version;
    //  ---------- 追加ここまで ↑ ↑ ↑ ----------

    @ManyToOne
                    :
}
```

【リスト7-3】com.example.todolist.form.TodoData.java

```
public class TodoData {
                    :
    private Integer completedTasks = 0;
    private Integer version = 0;    // 追加
                    :
    //    ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
        todo.setCompletedTasks(completedTasks);
        todo.setVersion(version);    // 追加
                    :
    }

    // Todo/AttachedFileエンティティからToDo入力画面へ渡すTodoDataを作成
    public TodoData(Todo todo, List<AttachedFile> attachedFiles)
{
        // Todo部分
                :
        this.completedTasks = todo.getCompletedTasks();
        this.version = todo.getVersion();    // 追加

        // 登録済Task一覧
            :
    }
}
```

【リスト7-4】src/main/resources/templates/todoForm.html

```
                    :
  ■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}">
        <input type="hidden" th:field="*{version}">    <!-- 追加 --
>
                    :
```

【リスト7-5】

src/main/resources/i18n/OperationMessages_en.properties

```
msg.w.todo_already_deleted=This todo has been deleted.
msg.w.task_already_deleted=This task has been deleted.
```

【リスト7-6】

src/main/resources/i18n/OperationMessages_ja.properties

```
msg.w.todo_already_deleted=このTodoは削除されていました。
msg.w.task_already_deleted=このTaskは削除されていました。
```

以上で定義追加は完了です。しかし、この状態でToDoを新規追加しようとすると、次のような例外が発生します。

```
org.springframework.web.method.annotation.MethodArgumentTypeMismatchException: Failed to
convert value of type 'java.lang.String' to required type 'int'; For input string: "null"]
```

原因はTodoListController#createTodo()の最後に実行するreturn文です。

```
        return "redirect:/todo/" + todo.getId();
```

todo.getId()がnullを返すため、これをリダイレクト先のtodoById()がintへ変換しようとして例外を起こしています。

このToDoを新規追加処理を仔細に見てみます。

【リスト7-7】

com.example.todolist.controller.TodoListController.java(変更前)

```
    @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();       // ①

todo.setOwnerId((Integer)session.getAttribute("accountId"));
            todoRepository.saveAndFlush(todo);    // ②

            // 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();       //
③todo.getId()がnullを返す

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

①追加するTodoエンティティを作成する → この時点でtodo.idはnull
②DBに追加する → 採番されたidの値がtodo.idへ設定される
③その値を取得してリダイレクト

しかしTodoに@Versionを追加すると②の挙動が変わり、todo.idはnullのままになるようです。そこでsaveAndFlush()は戻り値として「**保存(テーブルに格納)**

されたエンティティを返す」というのを利用します。

【リスト7-8】

com.example.todolist.controller.TodoListController.java(変更後)

```
    // 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();    // ①

todo.setOwnerId((Integer)session.getAttribute("accountId"));
            //todoRepository.saveAndFlush(todo);
            //        ↓
            Todo _todo = todoRepository.saveAndFlush(todo);
// ②戻り値取得

            // 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();
```

```
            //        ↓
            return "redirect:/todo/" + _todo.getId();       // ③戻り値からid取得

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

①追加するTodoエンティティを作成する → この時点でtodo.idはnull(変わらず)
②DBに追加した結果を取得 → 採番されたidの値が_todo.idへ保存されている
③その値を取得してリダイレクト

これで例外を解消できます。

> ほかにも良い方法があるかもしれません。興味がある方は調べてみてください。

7.2 Ealry Returnでネストを浅くする

ここからは【表7-1】～【表7-4】の結果が得られるようコードを変えていきます。
まず更新処理updateTodo()からです。このメソッドはTodoエンティティとTaskエンティティを同時に変更するためチェック処理が複雑です。こういった場合「**Ealry Return**」という方法を使うとプログラムの見通しをよくできる可能性があります。

たとえば、以下のようなロジックが必要だったとします。
・条件1,2,3,4がすべて成立する場合、「メイン処理」を実行する。
・条件1,2,3,4が不成立の場合、それぞれのエラー処理を実行する。

これを下図左側のように書くと、「条件チェック」と「エラー処理」が離れてしまい、わかりにくくなります。またメイン処理がネストの深いところにあるため「画面をスクロールしないと見えない」ということも起こります。
この処理は下図右側のようにも書けます。エラーなら即returnするので、条件-エラー処理の対応が明確です。またメイン処理のネストも深くならずに済みます。こういう書き方を「**Ealry Return**」といいます。

コード例1	コード例2(Ealry Return)

if(条件1){
if(条件2){
if(条件3){
if(条件4){
// ★メインの処理
} else{
	if(!条件1){
	// 条件1 エラー処理
	return;
	}
	if(!条件2){
	// 条件2 エラー処理
	return;

```
                    // 条件4 エラー処理
                }
            } else{
                //  条件3 エラー処理
            }
        } else{
            //  条件2 エラー処理
        }
    } else{
        //  条件1 エラー処理
}
```

```
    }
    if(!条件3){
        //  条件3 エラー処理
        return;
    }
    if(!条件4){
        //  条件4 エラー処理
        return;
    }
    //  ★メインの処理
```

【図7-1】Ealry Returnの例

以下は、このEalry Returnの考えを取り入れて作成した新しい更新処理updateTodo()です。

【リスト7-9】om.example.todolist.controller.TodoListController.java

```
    :
import com.example.todolist.form.TaskData;
import com.example.todolist.repository.TaskRepository;
    :
public class TodoListController {
    :
    private final TaskRepository taskRepository;
    :
  // ToDo更新処理
  @PostMapping("/todo/update")
```

```
    public String updateTodo(@ModelAttribute 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";
        }

        Todo todo = todoData.toEntity();
        //      完了タスク数
        if (todoData.getTaskList() != null) {
            int numOfCompletedTasks =
                    (int) todoData.getTaskList().stream()
                        .filter(task ->
task.getDone().equals("Y")).count();
            todo.setCompletedTasks(numOfCompletedTasks);
        }

        // ----------------------------------------------
        // - 更新前の整合性確認
```

```
        // ----------------------------------------------
        // 更新対象Todo(とTask)を取得
        Optional<Todo> _targetTodo =
todoRepository.findById(todoData.getId());
        if (!_targetTodo.isPresent()) {
            // 更新対象Todoが存在しない -> 削除された
            String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null,
locale);
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
            return "redirect:/todo";
        }
        // Todoのvesrion確認
        Todo targetTodo = _targetTodo.get();
        if (!targetTodo.getVersion().equals(todo.getVersion())) {
            // version不一致 -> 更新されている
            String msg = messageSource.getMessage(

"msg.w.optimistic_locking_failure", null, locale);
             redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
            return "redirect:/todo/" + todo.getId();
          }

        // Taskの存在/version確認
        boolean isTaskOk = true;
        if (todoData.getTaskList() != null) {
```

```
            for (TaskData taskData : todoData.getTaskList()) {
                Optional<Task> _task =
taskRepository.findById(taskData.getId());
                if (_task.isPresent()) {
                    Task task = _task.get();
                    // version不一致
                    if
(!task.getVersion().equals(taskData.getVersion())) {
                        isTaskOk = false;
                        break;
                    }
                } else {
                    // Taskが存在しない
                    isTaskOk = false;
                    break;
                }
            }
            if (!isTaskOk) {
                // 削除 or 更新されている
                String msg = messageSource.getMessage(

"msg.w.optimistic_locking_failure", null, locale);
                redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
                return "redirect:/todo/" + todo.getId();
            }
        }
```

```
        // ------------------------------------------------
        // - 更新処理
        // ------------------------------------------------
        try {
            todo = todoRepository.saveAndFlush(todo);

            // 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));
            return "redirect:/todo/" + todo.getId();

        } catch
(org.springframework.orm.ObjectOptimisticLockingFailureException
e) {

            // taskの更新が競合した(=誰かが先に更新 or 削除したとき)
            String msg = messageSource.getMessage(

"msg.w.optimistic_locking_failure", null, locale);
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
            return "redirect:/todo/" + todo.getId();
        }
    }
    :
}
```

このようにTodoを更新するときは、以下のようなチェックをしています。
1)対象となるTodoが存在する
2)対象となるTodoが更新されていない(Versionが一致する)
3)関連するTaskが存在する
4)関連するTaskが更新されていない(Versionが一致する)

Todo削除は、対象となるTodoが存在することを確認します(Versionの内容は問わない)。

【リスト7-10】

com.example.todolist.controller.TodoListController.java

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

        // 削除できるのは所有者のみ
        Integer accountId = (Integer) session.getAttribute("accountId");
        if (!todoData.getOwnerId().equals(accountId)) {
            // 削除NGメッセージをセットしてリダイレクト
            String msg = messageSource.getMessage("msg.e.todo_cannot_delete", null, locale);
```

```
                redirectAttributes.addFlashAttribute("msg", new
OpMsg("E", msg));
                return "redirect:/todo/" + todoId;
            }
            //      ---------- 追加ここから ↓ ↓ ↓ ----------
            //      存在チェック
            Optional<Todo> _targetTodo =
todoRepository.findById(todoId);
            if (!_targetTodo.isPresent()) {
                // 更新対象Todoが存在しない -> 削除された
                String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null,
locale);
                redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
                return "redirect:/todo";
            }
            //   ---------- 追加ここまで ↑ ↑ ↑ ----------

            // 添付ファイルを削除
            todoService.deleteAttachedFiles(todoId);

                    :
}
```

次にTask削除の処理ですが、これもEarly Returnで書き換えています。

【リスト7-11】

com.example.todolist.controller.TodoListController.java

```
        // 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,
                    RedirectAttributes redirectAttributes, Locale locale) {

                // ToDo取得 -> なければ削除済
                Optional<Todo> _todo =
todoRepository.findById(todoId);
                if (!_todo.isPresent()) {
                    String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null,
locale);
                    redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
                    mv.setViewName("redirect:/todo");
                    return mv;
                }

                // 操作者のToDoか? or Todoと同じグループに所属しているか？
                Todo todo = _todo.get();
                Integer accountId = (Integer)
session.getAttribute("accountId");
                @SuppressWarnings("unchecked")
                List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
```

```
            if (!todo.getOwnerId().equals(accountId) &&
!isBelong(groupsList, todo.getGroups())) {
                operationError(mv, redirectAttributes, locale);
                return mv;
            }

            // ------------------------------------------------
            // - 削除
            // ------------------------------------------------
            Optional<Task> _task = taskRepository.findById(taskId);
            if (_task.isPresent()) {
                try {
                    // この中でTaskを削除し、Todoのcompleted_tasksを
再計算する

todoService.deleteTaskAndRecalcCompletedTasks(taskId, todo);

                    // 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);

                } catch (Exception e) {
                    e.printStackTrace();
                    // Rollbackされている
                    operationError(mv, redirectAttributes, locale);
```

```
                }

            } else {
                // Taskが存在しない -> Task削除済
                String msg =
messageSource.getMessage("msg.w.task_already_deleted", null,
locale);
                redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
                mv.setViewName("redirect:/todo/" + todoId);
            }

            return mv;
        }
```

ここでは以下をチェックしてから削除します。
1)関連するTodoの存在チェック(Versionはチェックしない)
2)該当するTaskの存在チェック

さらに添付ファイルをアップロード/ダウンロードするときも、該当Todoがあることをチェックをします。

【リスト7-12】

com.example.todolist.controller.TodoListController.java

```
        // 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,
                                         :
```

```
        //      ---------- 追加ここから↓ ↓ ↓ ----------
        // ToDo取得
        Optional<Todo> someTodo =
todoRepository.findById(todoId);
        if (!someTodo.isPresent()) {
            // Todoが存在しない -> Todo削除済
            String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null,
locale);
            redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
            return "redirect:/todo";
        }
        //      ---------- 追加ここまで↑ ↑ ↑ ----------

        // ファイルが空？
        if (fileContents.isEmpty()) {
            :
```

【リスト7-13】

com.example.todolist.controller.TodoListController.java

```
    // 添付ファイルを削除する
    @GetMapping("/todo/af/delete")
    public ModelAndView
deleteAttachedFile(@RequestParam(name = "af_id") int afId,
            :
        // ToDo取得
```

```
            Optional<Todo> someTodo =
todoRepository.findById(todoId);
            someTodo.ifPresentOrElse(todo -> {

                :
            }, () -> {
                // todoが存在しない
                //    operationError(mv, redirectAttributes, locale);
                //    ↓
                // Todoが存在しない -> Todo削除済
                String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null,
locale);
                redirectAttributes.addFlashAttribute("msg", new
OpMsg("W", msg));
                mv.setViewName("redirect:/todo");
            });

            return mv;
        }
```

これでTodoとTaskの競合も検出・回避できるようになります。

8. 論理削除

プロジェクト名	Todolist23
作成ファイル	−(なし)
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Task.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java com.example.todolist.form.TodoData.java com.example.todolist.service.TodoService.java
SQLファイル	src/main/resources/sql/80_create_v_todolist.sql src/main/resources/sql/81_add_deleted_to_todo.sql src/main/resources/sql/82_add_deleted_to_ask.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「[削除]ボタン押したらどうなる？」
自分「テーブルから消えますけど...」
先輩「おかしい！」
自分「！？」
先輩「『そういうToDoがあった』というのも情報なんだから残すべき！」
自分「！」
先輩「改善よろしく！」ｽﾀｺﾗｻｯｻ
自分「...(どうすればいいの?)」

8.1 物理削除と論理削除

実務では、DELETE文でテーブルからデータを削除することは、そう多くありません。先輩が言うように、「ある時点でそういうデータが存在した」ということ自体一種の情報(事実)だからです。またこのToDoアプリのように、複数のテーブルを関連付けている場合、途中のテーブルのレコードが無くなると、関連付けが失われるためデータを表示できなくなる恐れもあります。

そのために実務では、DELETEする代わりに「**論理削除**」という手法をよく使います。これはテーブルに「削除されている/されていない」を表す列(フラグ)を設け、それで扱いを変える、というものです。これに対しDELETEでテーブルからレコードを本当に消してしまうことを「**物理削除**」と言います。

> この他、金額、数量などの数値訂正には「**赤黒処理**」という方法も使われます(特に会計処理の分野)。興味がある方は調べてみてください。

例としてaccountテーブルにis_deletedという列を追加し、これが'N'なら「削除されていないデータ」、'Y'なら「削除済」として扱ってみます。

■accountテーブル

id	login_id	name	password	is_deleted	備考
1	okada	岡田 是則	s6rizqfk	N	
2	inoue	井上 俊憲	g73phw5n	N	
3	inagaki	稲垣 絵美	s59mrtw3	N	
4	dummy	ダミー	....	Y	削除済

この場合有効なユーザー(=削除されてないユーザー)を取得するには、以下のようにします。

```
SELECT *  FROM  account  WHERE is_deleted = 'N';
```

「is_deleted='Y'」のレコードは取得されないので、「削除された」とみなせます。

非常に単純な仕組みですが、accountテーブルをアクセスする処理は、すべてこのルール(is_deleted != 'N'は削除済み)を守る必要があります。

またJPAの場合、is_deletedを含むメソッド宣言が必要となるでしょう。

・findAll()　→　findByIsdeleted(String isDeleted)　//　(削除されていない)全レコード検索

・findById()　→　findByIdIsdeleted(String isDeleted)　//　(削除されていない)主キーによる検索

：　：

Spring Bootには、このような条件を自動的にSELECT文へ追加するアノテーション(**@Where**)があります。本章ではこれを使い、TodoとTaskを論理削除できるようにします。

8.2 @Whereによる論理削除

まずtodoテーブルに論理削除用の列(フラグ)を追加し、Todoエンティティにも対応するプロパティを追加します。

【リスト8-1】src/main/resources/sql/81_add_deleted_to_todo.sql

```
ALTER TABLE todo ADD COLUMN is_deleted TEXT;
UPDATE todo SET is_deleted='N';
ALTER TABLE todo ALTER COLUMN is_deleted SET NOT NULL;
```

【リスト8-2】com.example.todolist.entity.Todo.java

```
                    :
import org.hibernate.annotations.Where;
                    :
@Entity
@Table(name = "todo")
@Where(clause = "is_deleted = 'N'")   // 追加
@Data
@ToString(exclude = "taskList")
public class Todo {
                    :
    private Integer version;

    @Column(name = "is_deleted")   // 追加
    private String isDeleted;   // 追加

    @ManyToOne
                    :
```

```
}
```

①SELECT文に付加する条件指定(**@Where**)

・clause属性に指定した内容は、JPAが生成した**SELECT文を実行するとき、自動的に付加されます。**

これだけで論理削除が機能するので、確認してみます。

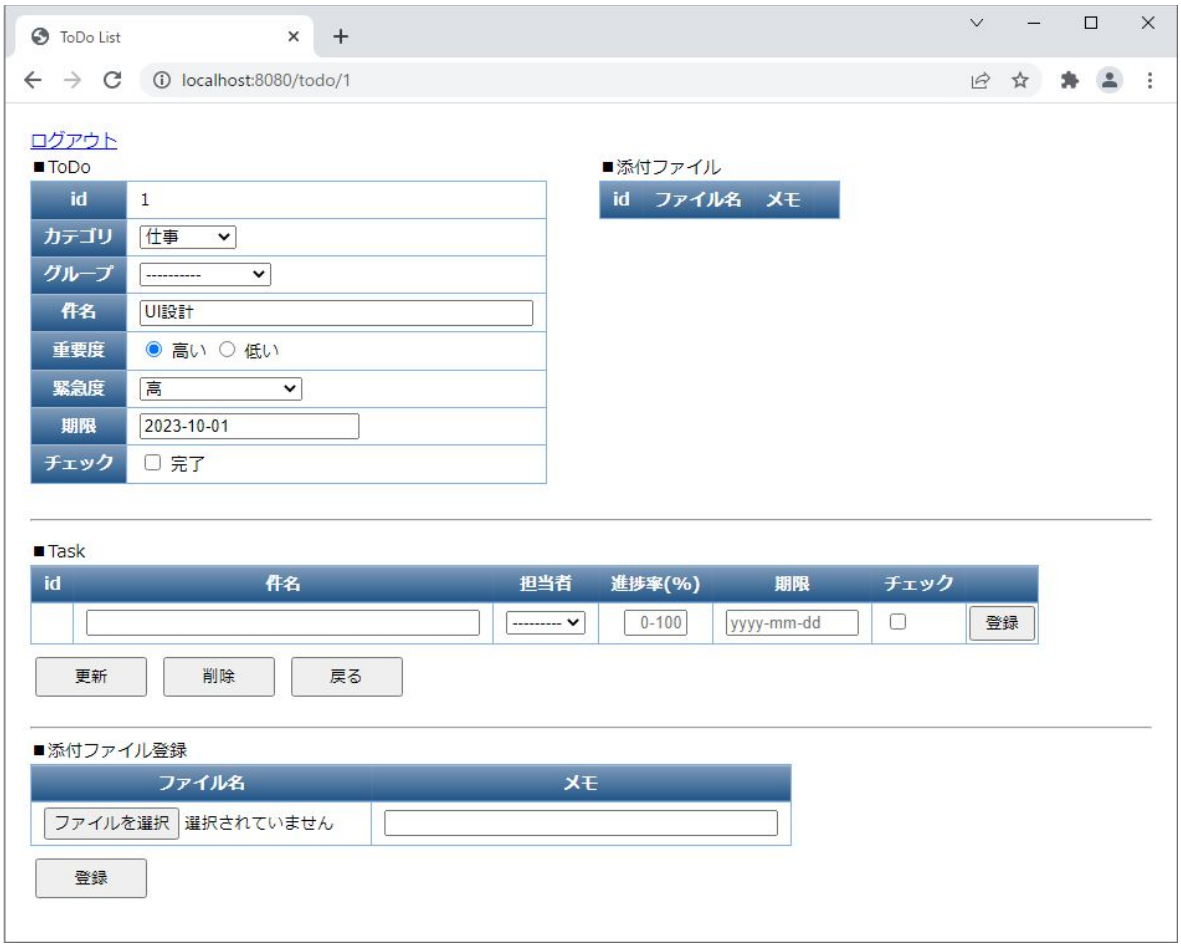
(1)Loginする

(2)Login完了

(3)Login者が作成したToDoをURL欄から直接アクセスする。

例. id=1のToDoを直接表示

(4)ToDo入力画面(todoForm.html)が表示される。



(5)(3)のToDoの論理削除フラグ(is_deleted)を'Y'にする。

例. id=1の場合

```
UPDATE todo SET is_deleted='Y' WHERE id = 1;
```

```
tododb=> UPDATE todo SET is_deleted = 'Y' WHERE id = 1;
UPDATE 1
tododb=>
```

(6)再度(3)のURLをアクセスする。

(7)今度は「操作に何らかの誤りがあるようです。」が表示される(/error)→削除データ扱いされている

デバッガーを使うとTodoListController#todoById()で、findById()を実行したときTodoを取得できないため、このメッセージが表示されることがわかります。

```
        // ToDo表示
        @GetMapping("/todo/{id}")
        public ModelAndView todoById(@PathVariable(name = "id") int
id, ModelAndView mv,
                                     RedirectAttributes
redirectAttributes, Locale locale) {
            // ToDo取得
            Optional<Todo> someTodo = todoRepository.findById(id);
```

```
            someTodo.ifPresentOrElse(todo -> {
                    :
            }, () -> {
                // todoが存在しない
                operationError(mv, redirectAttributes, locale);    // ★ここを実行している
            });
        }
```

4.3で解説した方法で実行したSELECT文を確認すると、次のようになっています。

```
SELECT    id, code, locale, NAME, completed_tasks, deadline, done, id, NAME, importance, ...
    FROM    todo
    LEFT    JOIN category    ON    code = category_code
     AND    locale = category_locale
    LEFT    JOIN GROUPS    ON    id = groups_id
WHERE    id = ?
    AND    (is_deleted = 'N') -- 追加されている
```

このように@Whereアノテーションの条件が追加されているため、is_deleted='Y'のToDoは検索できません(「該当・無」になる)。このように「**テーブルには存在するけど、無いことにする**」というのが「**論理削除**」です。

しかしToDo一覧(todoList.html)には、論理削除したはずのToDoが表示され続けています。リンクをクリックすると「操作に何らかの誤りがあるようです。」(/error)となってしまいます。

【図8-1】ビューには@Whereが適用されない

これは一覧の元になっているビューには、@Whereが適用されないためです。そしてv_todolistも、まだ論理削除のルール(is_deleted='N'を検索する)を盛り込んでいません。そこで以下のように条件を追加し、v_todolistを再作成すると、今度は一覧に表示されないようになります。

【リスト8-3】src/main/resources/sql/80_create_v_todolist.sql

```
DROP VIEW IF EXISTS v_todolist;
CREATE   VIEW v_todolist AS
SELECT   td2.id,
                    :
FROM
(
     SELECT id,
                    :
       FROM   todo
      WHERE groups_id = 0
```

```
        AND    is_deleted = 'N'    -- 追加
  UNION
    SELECT td.id,
                    :
      FROM todo td
      JOIN    groups_account ga ON ga.groups_id = td.groups_id
     WHERE    td.groups_id != 0
       AND    is_deleted = 'N'    -- 追加
) td2
                    :
```

8.3 論理削除処理

次は[削除]ボタンをクリックしたときの処理を論理削除へ変更します。これは「DELETEで削除する代わりにis_deletedへ'Y'を設定する」ということです。

まずToDo削除処理TodoListController#deleteTodo()を変更します

【リスト8-4】com.example.todolist.controller.TodoListController.java

```
    // ToDo削除処理
    @PostMapping("/todo/delete")
    public String deleteTodo(@ModelAttribute TodoData todoData,
                             :
            //      ---------- 削除ここから ↓ ↓ ↓ ----------①
            //      // 添付ファイルを削除
            //      todoService.deleteAttachedFiles(todoId);
            //
            //      // attached_fileテーブルから削除
            //      List<AttachedFile> attachedFiles =
attachedFileRepository
            //
.findByTodoIdOrderById(todoId);
            //      attachedFileRepository.deleteAllInBatch(attachedFiles);
            //      ---------- 削除ここまで ↓ ↓ ↓ ----------

            // todoを削除
            //todoRepository.deleteById(todoData.getId());
            //    ↓
            Todo targetTodo = _targetTodo.get();    // ②
            targetTodo.setIsDeleted("Y");    // ③
```

```
            todoRepository.saveAndFlush(targetTodo);   // ④

            // 削除完了メッセージをセットしてリダイレクト
                        :
    }
```

①添付ファイルは削除しない

・手動(SQL)でis_deletedに'N'をセットして、論理削除を取り消す(復活)場合があるかもしれないので、添付ファイルの情報は残しておきます。

②Todoを取得

・_targetTodoはOptional<Todo>型のためget()でTodoオブジェクトを取得します

③論理削除フラグセット

④テーブル更新

またTodoを登録、変更するときはisDeletedへ"N"を設定します。これはTodoData#toEntity()で入力画面の内容からTodoを作成するときに行います。

【リスト8-5】com.example.todolist.form.TodoData.java

```
public class TodoData {
                        :
    //    ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                        :
        todo.setVersion(version);
        todo.setIsDeleted("N");    // 追加
                        :
    }
    :
}
```

これで画面からの入力(登録、変更)は常に"N"となるので、ToDo入力画面にisDeletedに対応するフォーム部品を配置する必要はありません。

Taskも同様の手順で論理削除へ変更します。

【リスト8-6】src/main/resources/sql/82_add_deleted_to_task.sql

```
ALTER TABLE task ADD COLUMN is_deleted TEXT;
UPDATE task SET is_deleted='N';
ALTER TABLE task ALTER COLUMN is_deleted SET NOT NULL;
```

【リスト8-7】com.example.todolist.entity.Task.java

```
                    :
import org.hibernate.annotations.Where;
                    :
@Entity
@Table(name = "task")
@Where(clause = "is_deleted = 'N'")   // 追加
@Data
                    :
public class Task {
                    :
    private Integer version;

    @Column(name = "is_deleted")   // 追加
    private String isDeleted;   // 追加
}
```

【リスト8-8】com.example.todolist.form.TodoData.java

```
public class TodoData {
                    :
    //   ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
    public Todo toEntity() {
                    :
                      task = new Task(
                    :
                            taskData.getVersion(),
                            "N");    // 追加
                    :
    }
    :
    // ToDo入力画面(todoForm.html)新規タスク入力行の内容からエンティティを作成
    public Task toTaskEntity() {
                    :
        task.setVersion(newTask.getVersion());
        task.setIsDeleted("N");    // 追加
                    :
    }
```

タスクもToDoと同じく、画面から受け取るデータはis_deleted="N"です。よってTaskData経由で画面へ持たせる必要は無く、ここでリテラルとしてセットします。

【リスト8-9】com.example.todolist.service.TodoService.java

```
    @Transactional(rollbackForClassName = { "Exception" })
    public void deleteTaskAndRecalcCompletedTasks(Integer taskId, Todo todo) {
        Task task = taskRepository.findById(taskId).get();
```

```
        //taskRepository.deleteById(task.getId());
        //        ↓
        task.setIsDeleted("Y");
        taskRepository.saveAndFlush(task);
                :
    }
```

これでタスクも論理削除となりますが、ToDo一覧のタスク数にはそれが反映されていません。ビューv_todolisに以下の条件を追加し、再作成します。

【リスト8-10】src/main/resources/sql/80_create_v_todolist.sql

```
LEFT OUTER JOIN (SELECT    todo_id, COUNT(*) AS cnt
                           FROM    task
                           GROUP    BY    todo_id) tk
                ON td2.id = tk.todo_id;
                           ↓
LEFT OUTER JOIN (SELECT todo_id, COUNT(*) AS cnt
                           FROM    task
                           WHERE    is_deleted = 'N'    -- 追加
                           GROUP BY    todo_id) tk
                ON td2.id = tk.todo_id;
```

8.4 @DynamicUpdate

以下はtodo.id=1の件名を「UI設計」→「UI設計(Mock作成含む)」と変更したときに、Spring Bootが実行するUPDATE文です。件名だけ変更したのに、全項目SETしていることがわかります。

```
UPDATE    todo
   SET    category_code=?,
          category_locale=?,
          completed_tasks=?,
          deadline=?,
          done=?,
          groups_id=?,
          importance=?,
          is_deleted=?,
          owner_id=?,
          title=?,
          urgency=?,
          version=?
WHERE    id=?
      AND    version=?
binding parameter [1] as [VARCHAR] - [10]
binding parameter [2] as [VARCHAR] - [ja]
binding parameter [3] as [INTEGER] - [0]
binding parameter [4] as [DATE] - [2023-10-01]
binding parameter [5] as [VARCHAR] - [N]
binding parameter [6] as [INTEGER] - [0]
binding parameter [7] as [INTEGER] - [1]
binding parameter [8] as [VARCHAR] - [N]
```

```
binding parameter [9] as [INTEGER] - [1]
binding parameter [10] as [VARCHAR] - [UI設計(Mock作成含む)]
binding parameter [11] as [INTEGER] - [1]
binding parameter [12] as [INTEGER] - [1]
binding parameter [13] as [INTEGER] - [1]
binding parameter [14] as [INTEGER] - [0]
```

エンティティに**@DynamicUpdate**を付与すると、値を変える必要のある列だけをUPDATE文の対象とします。

【リスト8-11】com.example.todolist.entity.Todo.java

```
                    :
import org.hibernate.annotations.DynamicUpdate;
                    :
@Entity
@Table(name = "todo")
@Where(clause = "is_deleted = 'N'")
@DynamicUpdate      // 追加
                    :
public class Todo extends AuditInfo {
                    :
}
```

今度は「UI設計(Mock作成含む)」→「UI設計」としてみます。更新対象が大幅に減っていることがわかると思います。

```
UPDATE    todo
    SET    title=?,
           version=?
WHERE    id=?
```

```
    AND    version=?
binding parameter [1] as [VARCHAR] - [UI設計]
binding parameter [2] as [INTEGER] - [2]
binding parameter [3] as [INTEGER] - [1]
binding parameter [4] as [INTEGER] - [1]
```

Spring Boot(正確にはHibernate)は、一度実行したSQLをキャッシュして再利用します。しかし**@DynamicUpdate**を指定すると毎回生成するため、パフォーマンス悪化の原因になる可能性もあります。一般的には列数が多いテーブルのエンティティに適用すべきでしょう。

9. 監査情報の出力

プロジェクト名	Todolist24
作成ファイル	com.example.todolist.entity.AuditInfo.java com.example.todolist.config.LocalDateTimeConverter.java com.example.todolist.config.TodolistAuditorAware.java
変更ファイル	com.example.todolist.controller.TodoListController.java com.example.todolist.entity.Task.java com.example.todolist.entity.Todo.java com.example.todolist.form.TodoData.java com.example.todolist.service.TodoService.java src/main/resources/templates/todoForm.html src/main/resources/i18n/FixedDisplayStrings_en.properties src/main/resources/i18n/FixedDisplayStrings_ja.properties
SQLファイル	src/main/resources/sql/90_add_audit_to_todo.sql src/main/resources/sql/91_add_audit_to_task.sql

ある日オフィスの片隅で...

先輩「ToDoアプリなんだけどさ...」
自分「はい」
先輩「今度のシステム監査の対象になったから」
自分「はぁ？」
先輩「だって世界中の支社があれ使ってマネジメントしてんだから当然だろ？」
自分「...(マジかよ?)」
先輩「監査証跡(かんさしょうせき)とってるよな？」
自分「なんですかそれ？」
先輩「ということはとってない？」
自分「(コックリ...つーか聞いたことないよ)」
先輩「じゃ、遅ればせながら出せるようにしといてねぇ～！」ｽﾀｺﾗｻｯｻ
自分「...(何をどうすればいいの?)」

9.1 システム監査

ITの世界には「**システム監査**」という制度があります。これは企業などが使用している情報システムを、第三者(システム監査人)が客観的にチェック・評価するものです。このなかでは「企業経営に役立っているか」「障害発生につながるリスクはないか」「不正アクセスを防ぐことができるか」といった視点で課題を抽出し、システムの改善へつなげていきます。

とくに昨今は情報漏洩、サイバー攻撃など情報システムのリスク対策には、万全な対応が求められています。『応用編』「21. ログイン認証」で取り入れた「認証と認可」もその方策の 1 つですが、他にも「監査ログ」の出力など多様な手段があります。

中でも「いつ、誰がこのデータを変更した」あるいは「削除した」といった情報を記録する**監査証跡(監査情報)**は、多くのシステムで取り入れられています。そしてSpring Bootにも、サポートするアノテーションがあります。本章ではこれらを使い「データの作成者、作成時刻、変更者、変更時刻」を記録できるようにします。さらに次章では「変更前・後の値」を別テーブルへ出力できるようにするなど、機能強化を図っていきます。

> システム監査については、以下の資料が参考になります。冒頭にシステム監査の定義、目的が書かれています。
>
> システム監査基準(平成30年4月20日改訂) (PDF) (経済産業省)
> https://www.meti.go.jp/policy/netsecurity/downloadfiles/system_kansa_h30.pdf
> システム監査とは、専門性と客観性を備えたシステム監査人が、一定の基準に基づいて情報システムを総合的に点検・評価・検証をして、監査報告の利用者に情報システムのガバナンス、マネジメント、コントロールの適切性等に対する保証を与える、又は改善のための助言を行う監査の一類型である。
>
> システム監査は、情報システムにまつわるリスクに適切に対処しているかどうかを、独立かつ専門的な立場のシステム監査人が点検・評価・検証することを通じて、組織体の経営活動と業務活動の効果的かつ効率的な遂行、さらにはそれらの変革を支援し、組織体の目標達成に寄与すること、又は利害関係者に対する説明責任を果たすことを目的とする。

> 監査証跡として何が必要とされるかは、システム・組織・制度などにより異なります。本書で扱う項目以外にも、多くのものがあります。興味がある方は調べてみてください。

9.2 監査情報出力アノテーション

Spring Bootには、以下のような**監査情報を自動的に記録するアノテーション**があります。これらは後述するように、エンティティのプロパティへ付与します。そしてそのエンティティ(レコード)を追加・変更すると、監査情報がそれらのプロパティに記録されます。

同じ仕組みはアノテーションを使わなくても実現できますが、「漏れなく」行うのは大変です。アノテーションを使った方が、効率的に実現できるでしょう。

【表9-1】監査情報用アノテーション

#	アノテーション	監査(出力)情報	備考
①	@CreatedBy	レコードの新規作成者	
②	@CreatedDate	レコードの新規作成日時	
③	@LastModifiedBy	レコードの最終変更者	複数回変更したときは、その**最後**のもの(※)
④	@LastModifiedDate	レコードの最終変更日時	同上

※複数回変更した場合、「最後の変更情報しか残らない」ことに注意してください。途中の情報も残したいなら、次章の方法と併用します。

以下Todoへ適用する手順を見ていきます。

(1)監査情報用アノテーション/プロパティの追加

まずTodoエンティティに監査情報用プロパティとアノテーションを追加します。

【リスト9-1】com.example.todolist.entity.Todo.java

```
                    :
import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedBy;
import org.springframework.data.annotation.LastModifiedDate;
```

```
import org.springframework.data.jpa.domain.support.AuditingEntityListener;
import jakarta.persistence.EntityListeners;
                    :
@Entity
@Table(name = "todo")
@Where(clause = "is_deleted = 'N'")
@DynamicUpdate
@EntityListeners(AuditingEntityListener.class)    // ⑥
                    :
public class Todo {
                    :
    private String isDeleted;

    // ---------- 追加ここから ↓ ↓ ↓ ----------
    @CreatedBy
    @Column(name = "created_by", updatable = false)    // ③
    private String createdBy; // ①

    @CreatedDate
    @Column(name = "created_on", updatable = false)    // ③
    private java.util.Date createdOn; // ②

    @LastModifiedBy
    @Column(name = "lastmodified_by")
    private String lastModifiedBy;    // ④

    @LastModifiedDate
    @Column(name = "lastmodified_on")
    private java.util.Date lastModifiedOn; // ⑤
    //---------- 追加ここまで ↑ ↑ ↑ ----------

    @ManyToOne
                    :
}
```

①④新規作成者、最終変更者を表すプロパティ
・任意の型を指定できますが、ここではString型とします。
②⑤新規作成日時、最終変更日時を表すプロパティ
・使用できるデータ型については以下のような説明があります。
Spring Data JPA - Reference Documentation > 5.1.9. Auditing > Basics
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#auditing.basics

> As you can see, the annotations can be applied selectively, depending on which information you want to capture. The annotations, indicating to capture when changes are made, can be used on properties of type JDK8 date and time types, long, Long, and legacy Java Date and Calendar.

> DeepLによる翻訳
> ご覧のように、アノテーションはどの情報をキャプチャしたいかに応じて選択的に適用することができます。変更が加えられたときにキャプチャすることを示すアノテーションは、JDK8の日付と時刻のタイプ、long、Long、およびレガシーのJava DateとCalendarのプロパティで使用できます。

ここから以下のデータ型が標準で使用可能と考えられます。
・java.util.Date, Calendar
・JDK8のDate, Time(JDK8 date and time types)
・long
・java.lang.Long

このうち「JDK8のDate,Time」は何を指しているかはっきりしません。そこで本章では、最初java.util.Dateを使用し、後半「コンバーター」クラスを作成して明示的にjava.time.LocalDateTimeも使えるようにします。

> 著者が試した範囲では、後述のコンバータークラス無しでもjava.time.LocalDateTimeが使えるようです。これが上記ドキュメントの「JDK8 date and time types」のことかどうかは、不明です。

③「新規作成者」「新規作成日時」の変更不可指定

・updatable=falseを追加し、対応するテーブルの列がUPDATE文で変更されないようにします。
→この指定がないとtodo変更時「新規作成者」「新規作成日時」がNULLになる。

⑥コールバックリスナークラスの指定(**@EntityListeners**)
・このエンティティで追加・変更・削除、といったイベントが発生したとき呼び出すクラス(コールバックリスナークラス)を指定します。監査情報を記録させるときはAuditingEntityListener.classを指定します。

(2) テーブルに監査項目追加

(1)で追加したプロパティに対応する列をtodoテーブルへ追加します。

> 監査情報なので「仮の値を設定するのは不適切」と判断し、NULLのままとします。よってNOT NULL制約を付与しません。しかし、これから作成・使用するテーブルであれば、NOT NULL制約の適用を検討すべきでしょう。

【リスト9-2】src/main/resources/sql/90_add_audit_to_todo.sql

```
ALTER TABLE todo ADD COLUMN created_by TEXT; -- ①
ALTER TABLE todo ADD COLUMN created_on TIMESTAMP; -- ②
ALTER TABLE todo ADD COLUMN lastmodified_by TEXT; -- ①
ALTER TABLE todo ADD COLUMN lastmodified_on TIMESTAMP; -- ②
```

①@CreatedBy/@LastModifiedByのプロパティがString型なのでTEXT型
②@CreatedDate/@LastModifiedDateのプロパティがjava.util.DateなのでTIMESTAMP型
→Dateは年月日＋時分秒＋ミリ秒なので、PostgreSQLで対応するのは(DATEではなく)TIMESTAMP型

(3) イベントリスナーの作成

監査項目のうち「新規作成日時」「最終変更日時」はシステム時刻なので、Spring Bootで取得できます。しかし「新規作成者」「最終変更者」は、Spring Bootへ提供しなければなりませ

ん(Spring Bootだけでは、何を操作者情報として記録すればよいかわからないため)。
　これはAuditorAware<T>インターフェースを実装したクラスで行います。具体的には、このインターフェースが実装を要求するOptional<T> getCurrentAuditor()の戻り値が「新規作成者」「最終変更者」に使われます。

　以下が実装例です。

【リスト9-3】com.example.todolist.config.TodolistAuditorAware.java

```
package com.example.todolist.config;

import java.util.Optional;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.domain.AuditorAware;
import org.springframework.data.jpa.repository.config.EnableJpaAuditing;
import jakarta.servlet.http.HttpSession;
import lombok.RequiredArgsConstructor;

@Configuration    // ①
@EnableJpaAuditing    // ②
@RequiredArgsConstructor
public class TodolistAuditorAware implements AuditorAware<String> { // ③
	private final HttpSession session;

	@Override
	public Optional<String> getCurrentAuditor() {    // ④

		Integer accountId = (Integer) session.getAttribute("accountId");
		if (accountId == null) {
			return null;

		} else {
			return Optional.ofNullable("" + accountId); //    ⑤
		}
	}
}
```

①アプリ起動時、このクラスをSpring Bootに取り込んでもらう(**@Configuration**)
②監査情報記録を有効にする(**@EnableJpaAuditing**)
③AuditorAware<T>を実装する。
　総称型で「新規作成者」「最終変更者」を表すデータ型を指定します。
④「新規作成者」「最終変更者」を返すメソッド
　総称型で戻り値の型を指定します。
③セッションに保存されているaccountId(=todo.id)を返す
　→これが「**新規作成者**」「**最終変更者**」とし使われる。
　戻り値の型Optional<String>に合わせます。

以上で監査情報の設定は完了です。

ToDoアプリからToDoを登録後、以下のSQLを実行すると下図のようにcreated_by～lastmodified_onに監査情報が設定されていることを確認できます。

```
SELECT   id, owner_id AS oid, title,
         created_by AS c_by, created_on AS c_on,
         lastmodified_by AS m_by, lastmodified_on AS m_on
  FROM   todo ORDER BY id;
```

■登録時

```
 id ¦ oid ¦      title      ¦ c_by ¦           c_on          ¦ m_by ¦           m_on
----+-----+-----------------+------+-------------------------+------+-------------------------
  1 ¦   1 ¦ UI設計           ¦      ¦                         ¦      ¦
  2 ¦   1 ¦ 定例BBQ大会開催  ¦      ¦                         ¦      ¦
  3 ¦   2 ¦ 月末締切         ¦      ¦                         ¦      ¦
  4 ¦   3 ¦ DB設計           ¦      ¦                         ¦      ¦
  5 ¦   1 ¦ 新人歓迎会       ¦ 1    ¦ 2022-03-08 14:14:15.856 ¦ 1    ¦ 2022-03-08 14:14:15.856
(5 行)
```

このToDoを変更するとlastmodified_by, lastmodified_onだけが更新されます。

■変更時

```
 id ¦ oid ¦      title      ¦ c_by ¦           c_on          ¦ m_by ¦           m_on
----+-----+-----------------+------+-------------------------+------+-------------------------
  1 ¦   1 ¦ UI設計           ¦      ¦                         ¦      ¦
  2 ¦   1 ¦ 定例BBQ大会開催  ¦      ¦                         ¦      ¦
  3 ¦   2 ¦ 月末締切         ¦      ¦                         ¦      ¦
  4 ¦   3 ¦ DB設計           ¦      ¦                         ¦      ¦
  5 ¦   1 ¦ 新人歓迎会(仮)   ¦ 1    ¦ 2022-03-08 14:14:15.856 ¦ 3    ¦ 2022-03-08 14:15:04.348
(5 行)
```

なお監査情報の出力内容は、②@EnableJpaAuditingのパラメーターで一部変えることができます。たとえばmodifyOnCreateでは、以下のように新規作成時に、最終変更情報を記録する/しないを決められます。

【表9-2】@EnableJpaAuditing(modifyOnCreate = true) ... defaultの場合

	@CreatedBy	@CreatedDate	@LastModifiedBy	@LastModifiedDate
登録時	記録される	←	←	←
変更時	***変更なし***	←	記録される	←

【表9-3】@EnableJpaAuditing(modifyOnCreate = false)の場合

	@CreatedBy	@CreatedDate	@LastModifiedBy	@LastModifiedDate
登録時	記録される	←	**NULL**	←
変更時	***変更なし***	←	記録される	←

> @EnableJpaAuditingには、他にもsetDatesなどのパラメーターがあります。興味がある方は調べて見てください。

> 本章の監査情報出力は、ToDoアプリ経由でレコードを追加、変更したときだけ有効です。ToDoアプリ以外から操作したとき(たとえばpsqlなどからSQL文で直接操作した、など)は、機能しません。そういった場合も記録を残すのであれば、RDBが提供する監査機能を利用する、など他の手段を検討すべきでしょう。

9.3.Converterの追加

ここまで新規作成日時/最終変更日時はjava.util.Date型としましたが、昨今はJava SE 8で導入されたDate-Timeパッケージ(java.time)を使うのが一般的です。本章でもjava.time.LocalDateTimeへ変更します。

そのために必要なのがコンバータークラスです。これは上述したように、@CreatedDate/@LastModifiedDateがLocalDateTime型をサポートしておらず、Spring Bootがどう扱えばよいかわからないためです。そこでLocalDateTime型とテーブルのTIMESTAMP型に対応するjava.sql.timestamp型を相互に変換するコンバーターを作成し、Spring Bootへ組み込みます。

【図9-1】本章のコンバーターの概要

まずTodoエンティティのプロパティをLocalDateTime型へ変更します。

【リスト9-1】com.example.todolist.entity.Todo.java

```
public class Todo
                    :
    @CreatedDate
    @Column(name = "created_on", updatable = false)
    //private java.util.Date createdOn;    // ②
    //    ↓    変更
    private java.time.LocalDateTime createdOn;
                    :
    @LastModifiedDate
    @Column(name = "lastmodified_on")
    //private java.util.Date lastModifiedOn;    // ⑤
    //    ↓    変更
    private java.time.LocalDateTime lastModifiedOn;
                    :
}
```

次にコンバーターはAttributeConverterインターフェースを実装して作成します。

```
public interface AttributeConverter<X,Y> {   //   ①
        public Y convertToDatabaseColumn (X attribute); //   ②
        public X convertToEntityAttribute (Y dbData);    //   ③
}
```

①総称型でデータ型を表す
　エンティティのプロパティ型をX, DBに対応するデータ型をYに指定します。
②X型引数をY型へ変換するメソッド(エンティティ→テーブル用)
③Y型引数をX型へ変換するメソッド(テーブル→エンティティ用)

以下が実装したコンバータークラスです

【リスト9-2】com.example.todolist.config.LocalDateTimeConverter.java

```
package com.example.todolist.config;

import java.sql.Timestamp;
import java.time.LocalDateTime;
import jakarta.persistence.AttributeConverter;
import jakarta.persistence.Converter;

@Converter(autoApply = true)   // ④
public class LocalDateTimeConverter
                        implements AttributeConverter<LocalDateTime, Timestamp>
{   // ①

        @Override
        public java.sql.Timestamp convertToDatabaseColumn(LocalDateTime
localDateTime) {   // ②
                //       Entity LocalDateTime -> Table Timestamp
                return (localDateTime == null ? null :
java.sql.Timestamp.valueOf(localDateTime));
        }

        @Override
```

```
    public LocalDateTime convertToEntityAttribute(Timestamp timestamp) {   //
③
        //      Table Timestamp -> Entity LocalDateTime
        return (timestamp == null ? null : timestamp.toLocalDateTime());
    }
}
```

①プロパティ型とDBのデータ型を総称型で指定

・ここではjava.time.LocalDateTime⇔java.sql.Timestampの相互変換をします。

②LocalDateTime → Timestamp変換メソッド

③Timestamp → LocalDateTime変換メソッド

④コンバーターの適用方法

・autoApply = trueとすると、すべてのエンティティのLocalDateTime型プロパティ(総称型のX)に対し、このコンバーターが適用されます。

@Converter(autoApply = true)とせず、プロパティごとにコンバーターを指定することもできます。

```
    @CreatedDate
    @Column(name = "created_date")
  @Convert(converter = LocalDateTimeConverter.class)   //   追加
    private LocalDateTime   createdOn;
    :
    @LastModifiedDate
    @Column(name = "lastmodified_date")
    @Convert(converter = LocalDateTimeConverter.class)   //   追加
    private java.time.LocalDateTime lastModifiedOn;
```

これでToDoを登録、変更するとマイクロ秒まで記録されるようになります。

→【図9-1】のようにLocalDateTimeはns(ナノ秒)の精度だが、TIMESTAMPはμs(マイクロ秒)まで。

例. id=6がLocalDateTimeConverter適用後のデータ

```
 id ¦ oid ¦      title      ¦ c_by ¦            c_on            ¦ m_by ¦            m_on
----+-----+-----------------+------+----------------------------+------+----------------------------
  1 ¦   1 ¦ UI設計          ¦      ¦                            ¦      ¦
  2 ¦   1 ¦ 定例BBQ大会開催 ¦      ¦                            ¦      ¦
  3 ¦   2 ¦ 月末締切        ¦      ¦                            ¦      ¦
  4 ¦   3 ¦ DB設計          ¦      ¦                            ¦      ¦
  5 ¦   1 ¦ 新人歓迎会(仮)  ¦ 1    ¦ 2022-03-08 14:14:15.856    ¦ 3    ¦ 2022-03-08 14:15:04.348
  6 ¦   1 ¦ 夏季合宿(仮)    ¦ 1    ¦ 2022-03-08 15:02:09.482316 ¦ 3    ¦ 2022-03-08 15:06:25.619692
(6 行)
```

9.4 エンティティの継承

さらにTaskにも監査項目を追加します。まずTodoと同じようにテーブルへ監査情報の列を追加します。

【リスト9-3】src/main/resources/sql/91_add_audit_to_task.sql

```
ALTER TABLE task ADD COLUMN created_by TEXT;
ALTER TABLE task ADD COLUMN created_on TIMESTAMP;
ALTER TABLE task ADD COLUMN lastmodified_by TEXT;
ALTER TABLE task ADD COLUMN lastmodified_on TIMESTAMP;
```

次に対応するプロパティをTaskエンティティへ追加するわけですが、これはTodoと同じです。こういった場合、共通のスーパークラスに監査項目を定義し、それをTodo/Taskへ派生させることができます。

まずスーパークラスを定義します。

【リスト9-4】com.example.todolist.entity.AuditInfo .java

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

import java.time.LocalDateTime;
import org.springframework.data.annotation.CreatedBy;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedBy;
import org.springframework.data.annotation.LastModifiedDate;
import jakarta.persistence.Column;
import jakarta.persistence.MappedSuperclass;
import lombok.Getter;
import lombok.Setter;

@MappedSuperclass    //  ①
@Setter
@Getter
public class AuditInfo {
```

```
    @CreatedBy
    @Column(name = "created_by", updatable = false)
    protected String createdBy;

    @CreatedDate
    @Column(name = "created_on", updatable = false)
    protected LocalDateTime createdOn;

    @LastModifiedBy
    @Column(name = "lastmodified_by")
    protected String lastModifiedBy;

    @LastModifiedDate
    @Column(name = "lastmodified_on")
    protected LocalDateTime lastModifiedOn;
}
```

①親クラスであること表す(**@MappedSuperClass**)

・親クラスには@Tableを指定しません(=対応するテーブルを作らない)。

・本クラスで定義したプロパティに対応する列は、継承したクラスの@Tableで指定したテーブルに持ちます。

これをTodoとTaskで継承します。

【リスト9-5】com.example.todolist.entity.Todo.java

```
                    :
import lombok.EqualsAndHashCode;
                    :
@Entity
@Table(name = "todo")
@Where(clause = "is_deleted = 'N'")
@DynamicUpdate
@EntityListeners(AuditingEntityListener.class)
@Data
@EqualsAndHashCode(callSuper=true)   //   ①追加
@ToString(exclude = "taskList")
```

```
public class Todo extends AuditInfo {
//                        ^^^^^^^^^^^^^^^^^^^ 追加
                    :
          // ----- コメント化 or 削除ここから ↓ ↓ ↓ -----
//      @CreatedBy
//      @Column(name = "created_by", updatable = false)
//      private String createdBy;
//
//      @CreatedDate
//      @Column(name = "created_on", updatable = false)
//      //private java.util.Date createdOn;
//      //    ↓    変更
//      private java.time.LocalDateTime createdOn;
//
//      @LastModifiedBy
//      @Column(name = "lastmodified_by")
//      private String lastModifiedBy;
//
//      @LastModifiedDate
//      @Column(name = "lastmodified_on")
//      //private java.util.Date lastModifiedOn;
//      //    ↓    変更
//      private java.time.LocalDateTime lastModifiedOn;
      //----- コメント化 or 削除ここまで ↑ ↑ ↑ -----
                    :
}
```

①@EqualsAndHashCode

・これが無いと@Dataに次のような警告が出ます。

Generating equals/hashCode implementation but without a call to superclass, even though this class does not extend java.lang.Object. If this is intentional, add '@EqualsAndHashCode(callSuper=false)' to your type.

> DeepLによる翻訳
>
> equals/hashCodeの実装を生成するが、このクラスがjava.lang.Objectを継承していないにもかかわらず、superclassを呼び出さない。もしこれが意図的なものであれば、型

に'@EqualsAndHashCode(callSuper=false)'を追加してください。
→継承したプロパティをequals()/hashCode()へ含めない場合は、メッセージ通りとする。含めるときはcallSuper=trueとします。
・ここで指摘されているAuditInfoクラスは監査情報のみ持っており、equals()/hashCode()に含める必要は無い、と判断しここではfalseとします。

Taskエンティティも同様です。

【リスト9-6】com.example.todolist.entity.Task.java

```
                    :
import org.springframework.data.jpa.domain.support.AuditingEntityListener;
import jakarta.persistence.EntityListeners;
import lombok.EqualsAndHashCode;
                    :
@Entity
@Table(name = "task")
@Where(clause = "is_deleted = 'N'")
@EntityListeners(AuditingEntityListener.class) // 追加
@Data
@EqualsAndHashCode(callSuper = true) // 追加
@AllArgsConstructor
@NoArgsConstructor
public class Task    extends AuditInfo {
//                        ^^^^^^^^^^^^^^^^^^^ 追加
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;

                    :
}
```

これでtaskにも監査情報が設定されます。

```
SELECT    id,    todo_id AS tid, title,
          created_by AS c_by,    created_on AS c_on,
```

```
        lastmodified_by AS m_by,    lastmodified_on AS m_on
  FROM    task
ORDER    BY id;
```

```
 id ¦ tid ¦      title       ¦ c_by ¦            c_on            ¦ m_by ¦            m_on
----+-----+------------------+------+----------------------------+------+----------------------------
  1 ¦   2 ¦ 日時決め         ¦      ¦                            ¦      ¦
  2 ¦   2 ¦ 場所検討         ¦      ¦                            ¦      ¦
  3 ¦   2 ¦ 参加者募集HP作成 ¦      ¦                            ¦      ¦
  4 ¦   6 ¦ 予算申請(仮)     ¦ 1    ¦ 2022-03-08 15:16:18.719969 ¦ 3    ¦ 2022-03-08 15:26:41.927464
(4 行)
```

しかし以下のような不都合(問題)があります。
・ToDoを変更すると、関連する全タスクの最終変更者/日時も変更される。
・タスクを変更すると、関連するToDoの最終変更者/日時も変更される。
(todo.completed_tasksを変更しない場合も)

> 上記のような挙動で問題なし、と判断する考え方もあります。その場合、以下のような変更は不要です。

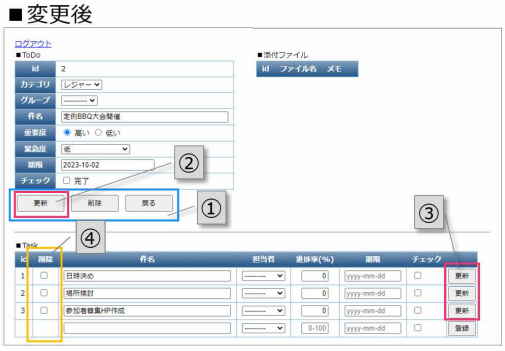
これはTodoとTaskを関連付けており(@OneToMnay/@ManyToOne)、それをTodoController#updateTodo()のTodoRepository#saveAndFlush(todo)で一括更新しているためです。そこでTodo/Taskの更新処理を分割し、監査情報が個別に設定されるようにします。

> ここから先は、特に目新しい技術的な話題はないため、お急ぎの方は次章へお進みください。

まずToDo入力画面(todoForm.html)のレイアウトを変更します。
①タスク一覧下にあった[更新][削除][戻る]ボタンをToDoの直下へ移動する。
②(①に含まれる)ToDoとタスクで共有していた[更新]はToDo専用とする。
③タスクの各行に[更新]ボタンを配置し、タスク個別に変更できるようにする。
④タスクを削除するときは[削除]チェック ON → 当該タスクの[更新]ボタンクリック、で行う。

■変更前

■変更後



【図9-2】ToDo入力画面(todoForm.html)変更前後比較

【リスト9-7】src/main/resources/templates/todoForm.html

```
                    :
<div>
  ■ToDo
  <!-- ToDo入力エリア -->
  <table>
    <!-- id -->
                    :
  </table>
<!-- ②タスク一覧下から移動 ここから
↓ ↓ ↓==========================================================
=====-->
  <!-- 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>
<!-- ②タスク一覧下から移動 ここまで
↑ ↑ ↑==========================================================-->
</div>
<!-- 更新の場合、添付ファイル一覧を表示する -->
    <div th:if="${session.mode == 'update'}">
                        :
    </div>
</div>
<!-- 更新の場合、Task一覧を表示する -->
<div th:if="${session.mode == 'update'}">
    <hr style="margin-top: 2em; margin-bottom: 1em;">
    ■Task
    <table>
        <tr>
            <th>id</th>
            <th th:text="#{label.delete}"></th><!-- ④削除チェックボックスの見出し追加 -->
                        :
            <th></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}" />
```

```
            <!-- 競合検出のために必要 -->
            <input type="hidden" th:name="${'taskList[' + stat.index + '].version'}"
                    th:value="${task.version}" />
        </td>
<!-- ④削除チェックボックス追加 ここから ↓ ↓ ↓ -->
        <!-- 削除 -->
        <td>
            <input type="checkbox" th:name="delTaskId" th:value="${task.id}">
        </td>
<!-- ④削除チェックボックス追加 ここまで ↑ ↑ ↑ -->
        <!-- 件名 -->
                    :
        <!-- チェック -->
        <td>
            <input type="checkbox" th:name="${'taskList[' + stat.index + '].done'}"
value="Y"
                    th:checked="*{taskList[__${stat.index}__].done=='Y'}" />
            <input type="hidden" th:name="${'!taskList[__${stat.index}__].done'}"
value="N" />
        </td>
        <!-- 削除リンク -->
<!-- ⑤削除リンク削除
================================================================
================================
        <td>
            <a th:href="@{/task/delete(task_id=${task.id},todo_id=*{id})}"
                th:text="#{link.delete}"></a>
        </td>
================================================================
=====================================-->
<!-- ⑥変更ボタン追加　ここから ↓ ↓ ↓ -->
        <td style="padding: 0px;">
            <button type="submit" th:formaction="@{'/task/update/' + ${task.id}}"
```

```
                    th:text="#{button.update}" style="margin: 2px; padding: 2px;
width: 4em;">
            </button>
        </td>
<!-- ⑥変更ボタン追加　ここまで ↑ ↑ ↑ -->
    </tr>
    <!-- 新規タスク入力行 -->
    <tr>
      <!-- id -->
      <td></td>
      <td></td><!-- ⑦削除チェックボックス列 追加-->
      <!-- 件名 -->
                :
    </tr>
  </table>
</div>
<!-- 更新時の操作ボタン -->
<!-- ②タスク一覧へ移動
================================================================
===============
<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 -->
<!-- ②タスク一覧へ移動
================================================================
===============
```

```
<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>
   <!-- 更新の場合、添付ファイル登録エリアを表示する -->

      :
```

【リスト9-8】src/main/resources/i18n/FixedDisplayStrings_en.propertie

```
label.delete=Delete
```

【リスト9-9】src/main/resources/i18n/FixedDisplayStrings_ja.propertie

```
label.delete=削除
```

次にロジックですが
　画面入力内容からTodo/Task作成 → saveAndFlush()
としていたものを
　Todo/Taskをfind → find結果に画面データ上書き → saveAndFlush()
へ変えます。そしてToDo更新処理からタスク更新処理を独立させます(updateTask())。

【リスト9-10】src.main.java.com.example.todolist.form.TodoData.java

```
public class TodoData {
                        :
    //   ToDo入力画面(todoForm.html)に入力された内容からTodoエンティティを作成
```

```
        public Todo toEntity() {

                        :
            // ①Taskの更新データはTaskの[更新]クリック時に、個別に取得するので削除
            // Task部分
//              Date date;
//              Task task;
//              if (taskList != null) {
//                    for (TaskData taskData : taskList) {
//                          date = Utils.str2dateOrNull(taskData.getDeadline());
//                          task = new Task(
//                                taskData.getId(),
//                                 :
//                                "N");
//
//                          //    Todoと関連付け
//                          todo.addTask(task);
//                    }
//              }

            return todo;
        }
}
```

【リスト9-11】com.example.todolist.controller.TodoListController.java①

```
    // ToDo更新処理
    @PostMapping("/todo/update")
    public String updateTodo(@ModelAttribute TodoData todoData, BindingResult
result, Model model,
                        :
          //    ----- Todoの変更ではcompleted_tasksの更新不要 → タスク更新時に移す--
---
          //       完了タスク数
          //if (todoData.getTaskList() != null) {
```

```
        //      int numOfCompletedTasks =
        //              (int) todoData.getTaskList().stream()
        //                                      .filter(task ->
task.getDone().equals("Y")).count();
        //      todo.setCompletedTasks(numOfCompletedTasks);
        //}
        // ------------------------------------------------------------------
                    :
        // ------------------------------------------------
        // - 更新前の整合性確認
        // ------------------------------------------------
                    :
        //   ----- Todoの変更ではTaskのversionチェック不要 → タスク更新時に移す-----
        // Taskの存在/version確認
        //boolean isTaskOk = true;
        //if (todoData.getTaskList() != null) {
                    :
        //}
        //   ------------------------------------------------------

        // ------------------------------------------------
        // - 更新処理
        // ------------------------------------------------
        try {
            //todo = todoRepository.saveAndFlush(todo);
            //    ↓
            //findしたTodoに画面データを上書き -> saveAndFlush
            targetTodo.setGroups(todo.getGroups());
            targetTodo.setCategory(todo.getCategory());
            targetTodo.setTitle(todo.getTitle());
            targetTodo.setImportance(todo.getImportance());
            targetTodo.setUrgency(todo.getUrgency());
            targetTodo.setDone(todo.getDone());
            targetTodo.setDeadline(todo.getDeadline());
```

```
                todoRepository.saveAndFlush(targetTodo);
                        :
    }
        :
}
```

【リスト9-12】com.example.todolist.controller.TodoListController.java(updateTask追加)

```
                    :
import org.springframework.data.repository.query.Param;
                    :
        //      Task更新処理
        @PostMapping("/task/update/{id}")
        public ModelAndView updateTask(
                            @PathVariable(name = "id") int taskId,
                            @Param("delTaskId") String delTaskId,
                            @ModelAttribute TodoData todoData, BindingResult
result,
                            ModelAndView mv, RedirectAttributes redirectAttributes,
Locale locale) {

                //      削除の場合
                if (delTaskId != null && !delTaskId.equals("")) {
                    int delId = Integer.parseInt(delTaskId);
                    if (taskId == delId) {
                        return deleteTask(taskId, todoData.getId(), mv,
redirectAttributes, locale);
                    }
                }

                // 操作者のTodoでない AND 所属するグループのTodoでない -> Something
wrong
                Todo todo = todoRepository.findById(todoData.getId()).get();
                Integer accountId = (Integer) session.getAttribute("accountId");
```

```
            @SuppressWarnings("unchecked")
            List<Groups> groupsList = (List<Groups>)
session.getAttribute("groupsList");
            if (!todoData.getOwnerId().equals(accountId) &&
                !isBelong(groupsList, todo.getGroups())) {
                mv.setViewName("redirect:/error");
                return mv;
            }

            // エラーチェック
            if (!todoService.isValid(todoData.getTaskList(), result, taskId, 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);
                mv.addObject("msg", new OpMsg("E", msg));
                mv.setViewName("todoForm");
                return mv;
            }

            // ------------------------------------------------
            // - 更新前の整合性確認
            // ------------------------------------------------
            Optional<Todo> _targetTodo =
todoRepository.findById(todoData.getId());
            if (!_targetTodo.isPresent()) {
                // 更新対象Todoが存在しない -> 削除された
                String msg =
messageSource.getMessage("msg.w.todo_already_deleted", null, locale);
                redirectAttributes.addFlashAttribute("msg", new OpMsg("W",
msg));
                mv.setViewName("redirect:/todo");
                return mv;
            }
```

```
            // ---------------------------------------------
            // - 更新処理
            // ---------------------------------------------
            try {
                // この中でTaskを更新し、Todoのcompleted_tasksを再計算する
                todoService.updateTaskAndRecalcCompletedTasks(todoData,
taskId);

                // 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));
                mv.setViewName("redirect:/todo/" + todoData.getId());
                return mv;

            } catch
(org.springframework.orm.ObjectOptimisticLockingFailureException e) {
                // taskの更新が競合した(=誰かが先に更新 or 削除したとき)
                String msg = messageSource.getMessage(
                                    "msg.w.optimistic_locking_failure", null, locale);
                redirectAttributes.addFlashAttribute("msg", new OpMsg("W",
msg));
                mv.setViewName("redirect:/todo/" + todoData.getId());
                return mv;
            }
        }
```

【リスト9-13】src/main/java/com/example/todolist/service/TodoService.java

```
                    :
import java.util.Optional;
import org.springframework.orm.ObjectOptimisticLockingFailureException;
                    :
    // ---------------------------------------------------------------------
```

```
    // Todo + Taskのチェック
    // ----------------------------------------------------------------------
    public boolean isValid(TodoData todoData, BindingResult result, boolean
isCreate,
                                        Locale locale) {
        boolean ans = true;
                :
        //   ----- Todo入力時、Taskはチェックしないので削除 ここから ↓ ↓ ↓ -----
        // ----------------------------------------------------------------------
        // Taskのチェック
        // ----------------------------------------------------------------------
        //List<TaskData> taskList = todoData.getTaskList();
        //if (taskList != null) {
            :
        //}
        //   ----- Todo入力時、Taskはチェックしないので削除 ここまで ↑ ↑ ↑ -----

        return ans;
    }
    :
    // ----------------------------------------------------------------------
    // 既存Taskのチェック(追加)
    // ----------------------------------------------------------------------
    public boolean isValid(List<TaskData> taskList, BindingResult result, int
taskId,
                                        Locale locale) {

        boolean ans = true;

        if (taskList != null) {
            // [更新]ボタンがクリックされたタスクだけチェックする
            //「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);
                    if (taskData.getId() != taskId) {
                        continue;
                    }

                    // 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(), "taskList[" + n + "].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(), "taskList[" + n + "].title",
                                messageSource.getMessage(
                                    "DoubleSpace.taskData.title", null, locale));
                            result.addError(fieldError);
                            ans = false;
                        }
                    }
                    // 進捗率
                    if (!taskData.getProgress().matches("^[0-9]+$")) {
                        FieldError fieldError = new FieldError(
                            result.getObjectName(), "taskList[" + n + "].progress",

messageSource.getMessage("Range.taskData.progress", null, locale));
                        result.addError(fieldError);
                        ans = false;
                    } else {
```

```
                        int progress = Integer.parseInt(taskData.getProgress());
                        if (progress < 0 || 100 < progress) {
                            FieldError fieldError = new FieldError(
                                    result.getObjectName(), "taskList[" + n +
"].progress",

messageSource.getMessage("Range.taskData.progress", 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",
                            messageSource.getMessage(
                                "InvalidFormat.todoData.deadline", null, locale));
                        result.addError(fieldError);
                        ans = false;
                    }

                    break;
                }
            }
            return ans;
        }
        :
        // ------------------------------------------------------------------------
        // Task更新 / Todoの完了タスク数再計算(追加)
        // ------------------------------------------------------------------------
        @Transactional(rollbackForClassName = { "Exception" })
```

```
    public void updateTaskAndRecalcCompletedTasks(TodoData todoData,
Integer taskId) {
        // 該当するTaskデータを取得
        TaskData taskData
            = todoData.getTaskList().stream()
                .filter(task -> task.getId() == taskId).findFirst().get();

        // Taskの存在/version確認
        Optional<Task> _task = taskRepository.findById(taskId);
        if (_task.isPresent()) {
            Task task = _task.get();
            // version不一致
            if (!task.getVersion().equals(taskData.getVersion())) {
                throw new ObjectOptimisticLockingFailureException(Task.class,
taskId);
            }
        } else {
            // Taskが存在しない
            throw new ObjectOptimisticLockingFailureException(Task.class,
taskId);
        }

        //      Task更新
        Task task =   _task.get();
        task.setTitle(taskData.getTitle());
        task.setAssignedTo(taskData.getAssignedTo());
        task.setProgress(Integer.parseInt(taskData.getProgress()));
        task.setDeadline(Utils.str2date(taskData.getDeadline()));
        task.setDone(taskData.getDone());
        taskRepository.saveAndFlush(task);

        //      TodoのCompletedTasks更新
        int numOfCompletedTasks
            = (int) todoData.getTaskList().stream()
```

```
                .filter(t -> t.getDone().equals("Y")).count();
        Todo todo = todoRepository.findById(todoData.getId()).get();
        todo.setCompletedTasks(numOfCompletedTasks);
        todoRepository.saveAndFlush(todo);
    }
```

以上の変更で、
・ToDoを変更しても、タスクの最終変更者/日時は変わらない
・タスクを変更しても、ToDoの最終変更者/日時は変わらない
ようになります。

> またToDoの更新とTaskの更新を分けたため、【表7-1】～【表7-4】とは挙動が一部変わります。さらに完了タスク数(todo.completed_tasks)を変更したかどうかでも変わります。興味がある方は調べてみてください。

10. 監査テーブルの作成

プロジェクト名	Todolist25
作成ファイル	com.example.todolist.config.TodoRevInfoListener.java com.example.todolist.entity.TodoRevInfo.java
変更ファイル	com.example.todolist.entity.Task.java com.example.todolist.entity.Todo.java src/main/resources/application.properties pom.xml
SQLファイル	ー(なし)

本章ではさらにレコードが追加・変更・削除されたら(いわゆる「更新系の処理」が実行されたら)、その内容を変更履歴として、別テーブルへ出力されるようにします。この出力先テーブルを「**監査テーブル**」と言います。この情報があれば「いつ、だれが、このタスクの進捗率を10から50へ変更したのか？」といったことを、後から調査できます。

以下、本章では監査テーブルへ監査情報などを出力することを「監査する」と表現します。

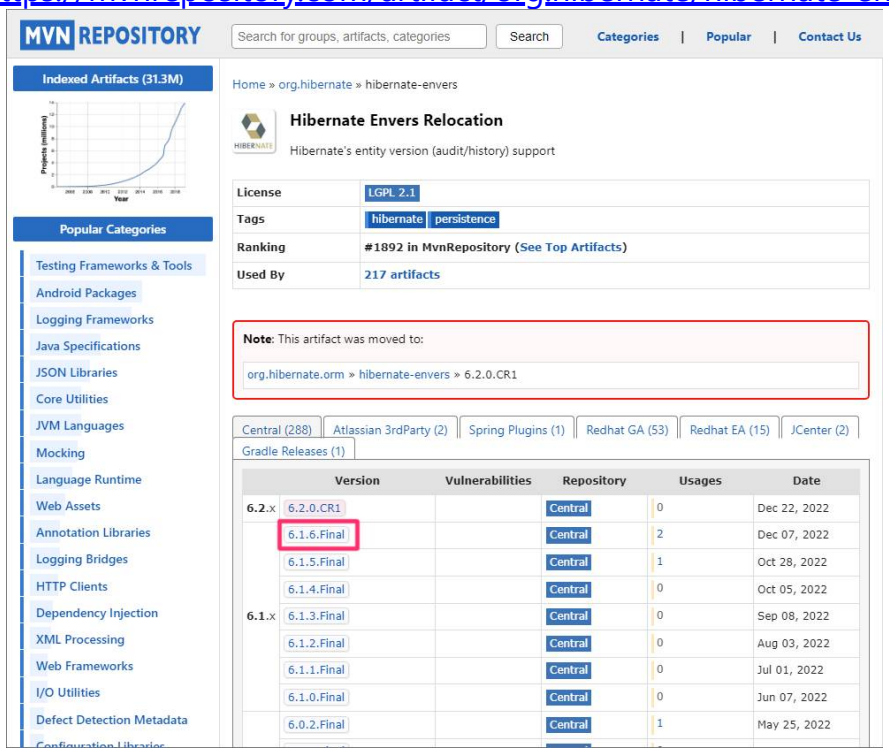
これもやろうとしていることは単純ですが、すべての更新系処理に監査テーブルへの出力機能をつけるのは面倒ですし、漏れがあっては監査データとしての信頼性を失いかねません。

ここまでこういった時はSpring Bootの便利機能を使ってきましたが、これに相当するものはありません。代わりに「**Hibernate Envers**」というものを利用します(これもアノテーション中心です)。Hibernateは、Spring Bootを構成するコンポーネントの１つですが、標準ではEnversが含まれていません。そこで以下のように、ToDoアプリへ組み込むところから始めます。

10.1 Hibernate Envers

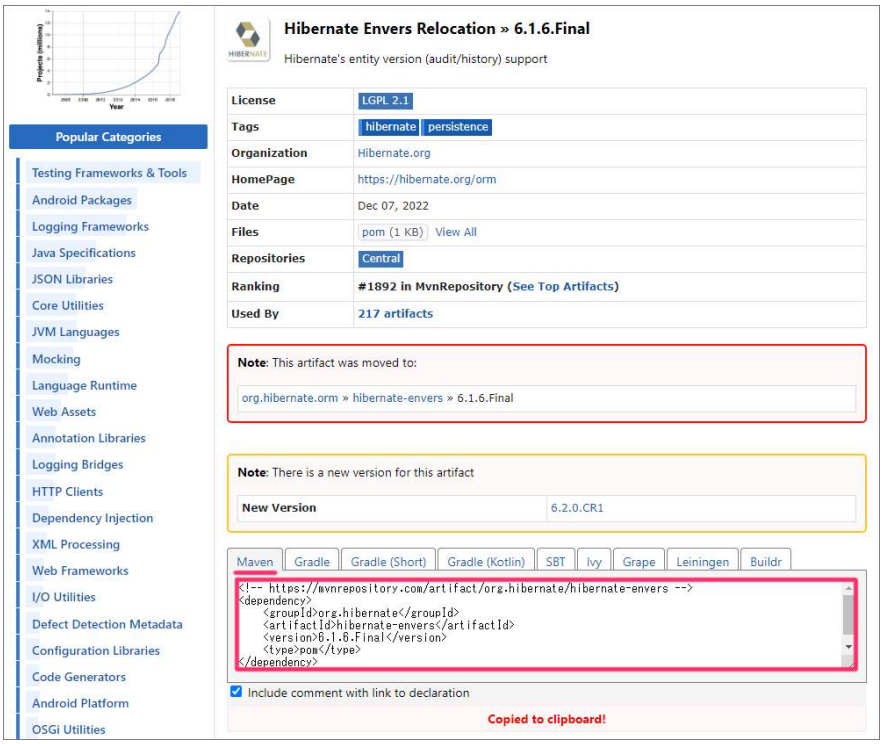
(1) Hibernate Enversの導入

1)Hibernate Enversは以下のページから入手できます。
https://mvnrepository.com/artifact/org.hibernate/hibernate-envers

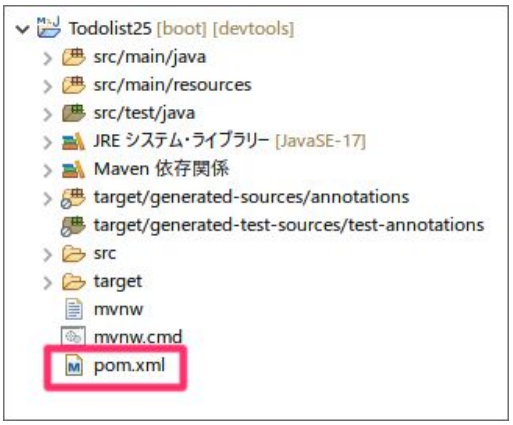


→執筆時点の最新版は6.1.6.Final(6.2.0.CR1はリリース候補版(rerease candidate))
→6.1.6.Finalのリンクをクリック

2)Mavenのタグが選択されていることを確認し、textarea部分をクリックする。
→内容がクリップボードにコピーされる



3)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。



4)</dependencies>直前に、クリップボードの内容を貼り付ける。
5)貼り付けた中にある**<type>pom</type>の行は削除**する。

【リスト10-1】pom.xml

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-envers -->
<dependency>
    <groupId>org.hibernate</groupId>
```

```
        <artifactId>hibernate-envers</artifactId>
        <version>6.1.6.Final</version>
</dependency>
```

これでpom.xmlを保存すると、自動的にHibernate Enversがダウンロードされ、ToDoアプリへ組み込まれます。

(2) Todoエンティティの変更

次にどのテーブルを監査対象とするか？を決めます。ToDoアプリの全テーブルを対象としてもいいのですが、ここではtodo/taskのみとします。これは**@Audited**というアノテーションで指定します。

【リスト10-2】com.example.todolist.entity.Todo.java

```
                    :
import org.hibernate.envers.Audited;
import org.hibernate.envers.RelationTargetAuditMode;
                    :
@Entity
@Audited    // ①追加
@Table(name = "todo")
                    :
public class Todo extends AuditInfo {
      :
      @ManyToOne
      @JoinColumns ({
```

```
            @JoinColumn(name="category_code", referencedColumnName =
"code"),
            @JoinColumn(name="category_locale", referencedColumnName =
"locale"),
        })
        @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)// ②
追加
        private Category category;
                        :
        @ManyToOne
        @JoinColumn(name = "groups_id")
        @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)// ③
追加
        private Groups groups;
}
```

①このエンティティ全体を監査対象とする(**@Audited**)

・@Auditedは、エンティティ/プロパティのどちらにも付与できますが、ここではTodoエンティティとします。これでTodoの**全プロパティが監査対象**となります(Todoのプロパティが変更されたら、監査テーブルへ情報が出力される)。

②リレーションシップ先のエンティティは監査しない

(RelationTargetAuditMode.***NOT_AUDITED***)

・Categoryには@ManyToOneが付与されているので1:nのn側です。ここでは「categoryの値は対象とするが、Categoryエンティティ(=categoryテーブル)の変更は**監査しない**」ことを指定します。これは上述のようにTodo/Taskのみ監査対象とするためです。

→この指定をしないなら、Categoryにも@Auditedを付与して監査対象としなければなりません。

③②同様対象外

(3) Taskエンティティの変更

Taskエンティティにも@Auditedを付与します。

【リスト10-3】com.example.todolist.entity.Task.java

```
                    :
import org.hibernate.envers.Audited;
                    :
@Entity
@Audited    // ①追加
@Table(name = "task")
                    :
public class Task extends AuditInfo {
                    :
}
```

①Taskもエンティティ全体を監査対象とする

Todo/Taskの監査準備は以上です。

(4) 監査テーブルを自動生成する

次に監査結果を格納するテーブル(監査テーブル)を作成します。方法はいくつかありますが、ここではSpring Boot起動時、自動生成してもらうこととし、application.propertiesに次の 1 行を追加します。

【リスト10-4】src/main/resources/application.properties

```
#①
spring.jpa.hibernate.ddl-auto=update
```

①アプリケーション起動時テーブルがなければ作成

・spring.jpa.hibernate.ddl-autoは、エンティティクラス(=@Entityが付与されているクラス)からテーブルを自動生成する機能です。これで監査テーブルを作成します。

・なおこのプロパティに指定できる値には、以下のようなものがあります。

【表10-1】spring.jpa.hibernate.ddl-autoに設定可能な値

設定値	機能
none	(何もしない)
validate	エンティティとテーブルの定義が合っているか検証する(DBに対しては何もしない)

create	アプリケーション起動時、エンティティに対応するテーブルがなければ作成する。 ※もしあればテーブルを**削除(DROP)後**、作成するので注意
create-drop	アプリケーション起動時、エンティティに対応するテーブルがなければ作成する。 ※アプリケーション**終了時**、テーブルを**削除**するので注意
update	アプリケーション起動時、エンティティに対応するテーブルがなければ作成する。 ※もしあれば**何もしない** ※アプリケーション終了時、テーブルを**削除しない**

> 【注意】
> ここでspring.jpa.hibernate.ddl-auto=updateを追加してToDoアプリを起動すると、エンティティ内で使用されている@OneToMany/@ManyToOneに対応した外部キー制約(FOREIGN KEY)がテーブルへ付与されます。ToDoアプリの実行には問題無いのですが、これ以降テーブルを手動で削除するときは、以下のようにします。
> ・参照先(子テーブル)→参照元(親テーブル)の順に削除する　または
> ・**DROP TABLE IF EXISTS** テーブル名 **CASCADE;**　のようにCASCADEを指定する

　これでToDoアプリ(Todolist25)を起動すると、監査テーブルなどが自動生成されます。これはpsqlの¥dコマンドで確認できます。

```
tododb=> \d
                リレーション一覧
 スキーマ |        名前          |   タイプ   |  所有者
----------+----------------------+------------+----------
 public   | account              | テーブル   | todouser
 public   | account_id_seq       | シーケンス | todouser
 public   | attached_file        | テーブル   | todouser
 public   | attached_file_id_seq | シーケンス | todouser
 public   | category             | テーブル   | todouser
 public   | groups               | テーブル   | todouser
 public   | groups_account       | テーブル   | todouser
 public   | groups_account_id_seq | シーケンス | todouser
 public   | groups_id_seq        | シーケンス | todouser
 public   | revinfo              | テーブル   | todouser   //  (3)追加
 public   | revinfo_seq          | シーケンス | todouser   //  (4)追加
 public   | task                 | テーブル   | todouser
 public   | task_aud             | テーブル   | todouser   //  (1)追加
 public   | task_id_seq          | シーケンス | todouser
 public   | todo                 | テーブル   | todouser
 public   | todo_aud             | テーブル   | todouser   //  (2)追加
 public   | todo_id_seq          | シーケンス | todouser
 public   | v_todolist           | ビュー     | todouser
(18 行)
```

【図10-1】起動後のリレーション一覧

(1)のtask_audはtaskの監査テーブルです(付加された"_aud"はaudit(監査)の略)。
定義は以下のようになっています。

■task_audの定義

```
tododb=> \d task_aud
                    テーブル"public.task_aud"
     列      |         タイプ         | 照合順序 | Null 値を許容 | デフォルト
-------------+------------------------+----------+---------------+------------
 id          | integer                |          | not null      |
 rev         | integer                |          | not null      |
 revtype     | smallint               |          |               |
 assigned_to | integer                |          |               |
 deadline    | date                   |          |               |
 done        | character varying(255) |          |               |
 is_deleted  | character varying(255) |          |               |
 progress    | integer                |          |               |
 title       | character varying(255) |          |               |
 todo_id     | integer                |          |               |
インデックス:
    "task_aud_pkey" PRIMARY KEY, btree (rev, id)
外部キー制約:
    "fkaerb34sjraiw4vjh4oh46rb71" FOREIGN KEY (rev) REFERENCES revinfo(rev)
```

※外部キー制約名は環境で異なります。

もとのtaskテーブルと比較すると下表のようになります。

【表10-2】taskとtask_audの比較

#	列名	task	task_aud	備考
1	id	○	○	
2	todo_id	○	○	
3	title	○	○	String → character varying(255)
4	deadline	○	○	
5	done	○	○	String → character varying(255)
6	assigned_to	○	○	
7	progress	○	○	
8	version	○	×	
9	is_deleted	○	○	String → character varying(255)
10	created_by	○	×	
11	created_on	○	×	
12	lastmodified_by	○	×	
13	lastmodified_on	○	×	
14	rev	×	○	integer型
15	revtype	×	○	smallint型(-32768～+32767)

○：あり、×：なし

このようにtaskテーブルと以下の点が異なります。
1. rev,revtypeという列が追加されている
2. 監査情報、楽観ロック用の列が無い
3. String型プロパティは上限255文字にされる(character varying(255))

(2)todo_audも同じようなルールで作成されています。

■todo_audの定義

```
tododb=> \d todo_aud
                        テーブル"public.todo_aud"
       列        |         タイプ         | 照合順序 | Null 値を許容 | デフォルト
-----------------+------------------------+----------+---------------+------------
 id              | integer                |          | not null      |
 rev             | integer                |          | not null      |
 revtype         | smallint               |          |               |
 completed_tasks | integer                |          |               |
 deadline        | date                   |          |               |
 done            | character varying(255) |          |               |
 importance      | integer                |          |               |
 is_deleted      | character varying(255) |          |               |
 owner_id        | integer                |          |               |
 title           | character varying(255) |          |               |
 urgency         | integer                |          |               |
 category_code   | character varying(255) |          |               |
 category_locale | character varying(255) |          |               |
 groups_id       | integer                |          |               |
インデックス:
    "todo_aud_pkey" PRIMARY KEY, btree (rev, id)
外部キー制約:
    "fkou5iipws0uvcno30bt4qx0qbs" FOREIGN KEY (rev) REFERENCES revinfo(rev)
```

ではこの監査テーブルにどのような情報が格納されるか、試してみます。ここでは次のような操作を行ったものとします。

■シナリオ

1. 岡田さん(account.id=1)がToDoを登録(カテゴリ：仕事、タイトル：Envers, 重要度：高、緊急度：高)

2. 岡田さんが上記ToDoを変更(カテゴリ：勉強、グループ：設計部)

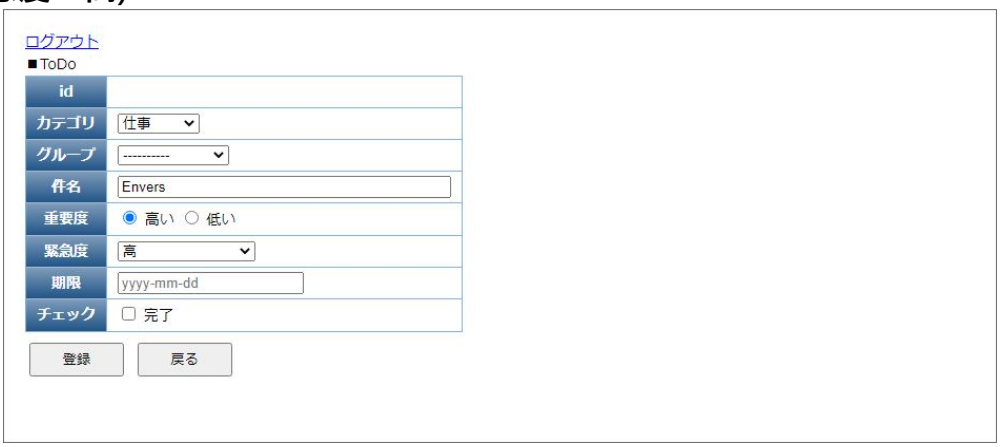
3. 岡田さんがタスクを登録(タイトル：task-1、進捗度:0)

4. 稲垣さん(account.id=3)が上記タスクを変更(担当者：稲垣)

5. 稲垣さんが上記タスクを削除

6. 岡田さんが上記ToDoを削除

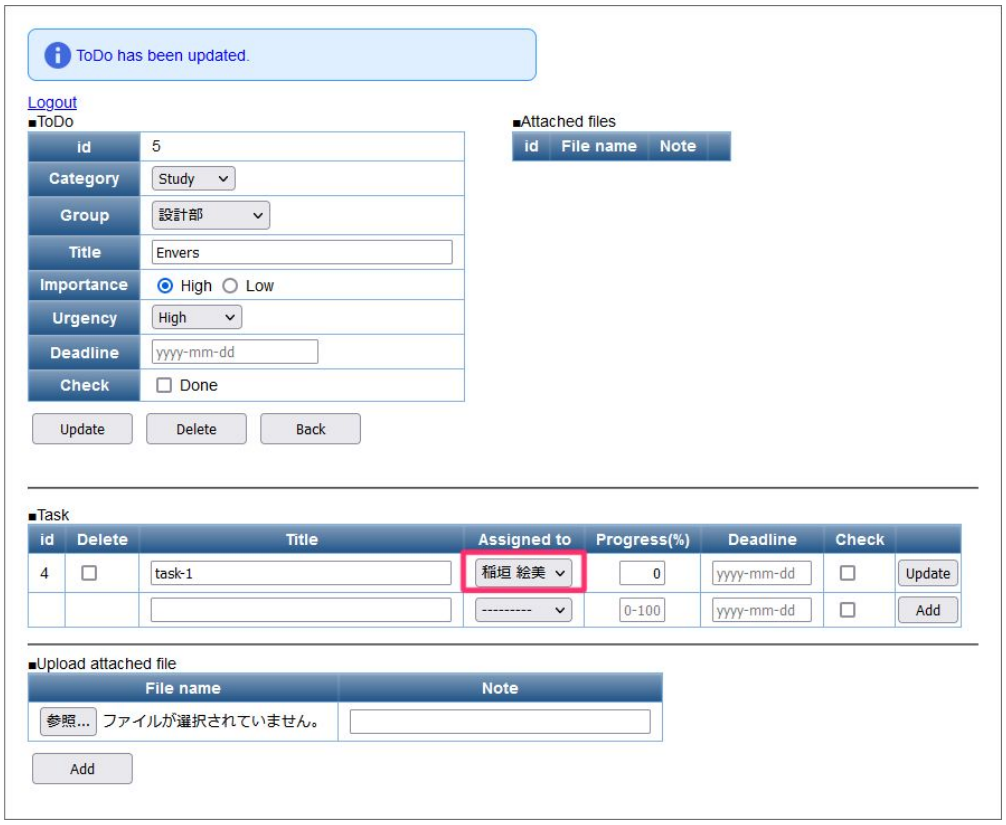
1. 岡田さん(account.id=1)がToDoを登録(カテゴリ：仕事、件名：Envers, 重要度：高、緊急度：高)



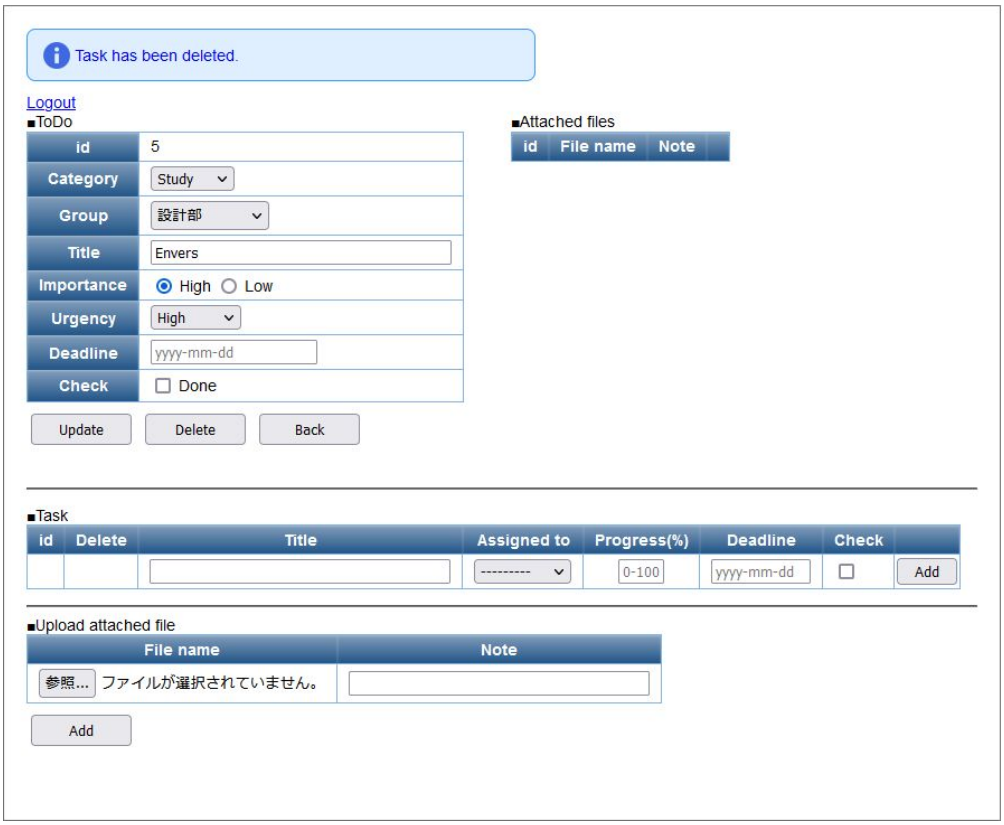
2. 岡田さんが上記ToDoを変更(カテゴリ：勉強、グループ：設計部)

3. 岡田さんがタスクを登録(タイトル：task-1、進捗度:0)

4. 稲垣さん(account.id=3)が上記タスクを変更(担当者：稲垣)



5. 稲垣さんが上記タスクを削除



6. 岡田さんが上記ToDoを削除

ここで以下のようなSELECT文を実行するとtodo_aud/task_audの内容を確認できます。

■シナリオ実行後の監査テーブル

```
SELECT   id, rev, revtype, owner_id,   title, importance, urgency,done,
category_code,
            groups_id, is_deleted
```

```
    FROM    todo_aud;

SELECT    id, rev, revtype, todo_id, title, deadline, done, assigned_to, progress,
is_deleted
    FROM    task_aud;
```

■実行結果

```
tododb=> select id, rev, revtype, owner_id,title, importance, urgency, done, category_code, groups_id, is_deleted from todo_aud;
  id ¦ rev ¦ revtype ¦ owner_id ¦ title  ¦ importance ¦ urgency ¦ done ¦ category_code ¦ groups_id ¦ is_deleted
----+-----+---------+----------+--------+------------+---------++------+--------------+-----------+-----------
   5 ¦   1 ¦       0 ¦        1 ¦ Envers ¦          1 ¦       1 ¦ N    ¦ 10           ¦         0 ¦ N
   5 ¦   2 ¦       1 ¦        1 ¦ Envers ¦          1 ¦       1 ¦ N    ¦ 20           ¦       100 ¦ N
   5 ¦   3 ¦       1 ¦        1 ¦ Envers ¦          1 ¦       1 ¦ N    ¦ 20           ¦       100 ¦ N
   5 ¦   6 ¦       1 ¦        1 ¦ Envers ¦          1 ¦       1 ¦ N    ¦ 20           ¦       100 ¦ Y
(4 行)

tododb=> select id, rev, revtype, todo_id, title, deadline, done, assigned_to, progress, is_deleted from task_aud;
 id ¦ rev ¦ revtype ¦ todo_id ¦ title  ¦ deadline ¦ done ¦ assigned_to ¦ progress ¦ is_deleted
----+-----+---------+---------+--------+----------+------+-------------+----------+-----------
  4 ¦   3 ¦       0 ¦       5 ¦ task-1 ¦          ¦ N    ¦           0 ¦        0 ¦ N
  4 ¦   4 ¦       1 ¦       5 ¦ task-1 ¦          ¦ N    ¦           3 ¦        0 ¦ N
  4 ¦   5 ¦       1 ¦       5 ¦ task-1 ¦          ¦ N    ¦           3 ¦        0 ¦ Y
(3 行)
```

格納されている内容は、シナリオと照らし合わせると大体想像がつくと思います。ポイントはrevtypeの値です。

【表10-3】監査テーブルの内容

#	操作	revtype	監査テーブルの内容
1	追加(INSERT)	0	追加した内容
2	変更(UPDATE)	1	変更後の内容
3	削除(DELETE)	2	削除時点した内容

> todo/taskは論理削除のため、削除してもrevtype=1です。

このうちrevは、【図10-1】(3)revinfoテーブルの外部キーになっています(上記todo_aud/task_audの定義参照)。そのrevinfoには、revの操作を行った時間のタイムスタンプ(1970/1/1 00:00:00からの経過ミリ秒)がrevtstmpとして記録されています。

■シナリオ実行後のrevinfo

```
tododb=> SELECT * FROM revinfo;
 rev |   revtstmp
-----+---------------
   1 | 1646456580425
   2 | 1646456618203
   3 | 1646456679861
   4 | 1646456726923
   5 | 1646456742698
   6 | 1646456764803
(6 行)
```

PostgreSQLではrevtstmpを以下のようなSQLで'YYYY-MM-DD HH24:MI:SS.sss'形式へ変換できます(元のタイムスタンプがUTCのため、この例のように必要に応じてJSTへ変更する必要あり)

```
SELECT    rev,
          timezone('JST',
               to_timestamp('1970-01-01 09:00:00', 'YYYY-MM-DD HH24:MI:SS.sss')
               +    revtstmp    *    interval    '1 millisecond')
  FROM    revinfo;
```

■シナリオ実行後のrevinfo(タイムスタンプ編集版)

```
 rev |        timezone
-----+-------------------------
   1 | 2022-03-05 14:03:00.425
   2 | 2022-03-05 14:03:38.203
   3 | 2022-03-05 14:04:39.861
   4 | 2022-03-05 14:05:26.923
   5 | 2022-03-05 14:05:42.698
   6 | 2022-03-05 14:06:04.803
(6 行)
```

これがrevの示す操作が行われた日時です。
さらにrevがいくつまで使われたか？を管理するのが【図10-1】(4)hibernate_sequenceです。

■シナリオ実行後のhibernate_sequence

```
tododb=> select * from revinfo_seq;
 last_value | log_cnt | is_called
------------+---------+-----------
          6 |      29 | t
(1 行)
```

　これでtodo/taskに対して「いつ、どのような更新系操作をしたのか？」を記録できます。しかし「誰か」は不完全です。新規作成者/最終更新者はtodo/taskのcreated_by/lastmodified_byにありますが、途中の変更者は不明です(記録されていない)。そこでEnversの設定を変更し、その記録が残せるようにします。またそれに合わせ、監査テーブル名などをわかりやすいものに変更します。

10.2 Enversの拡張

(1) 監査テーブルの名称変更

まず監査テーブルの名称を変更します。

【リスト10-5】src/main/resources/application.properties

```
#以下の行を追加する
#①
spring.jpa.properties.org.hibernate.envers.audit_table_suffix=_AUDIT
#②
spring.jpa.properties.org.hibernate.envers.revision_type_field_name=OP_TYPE
```

①監査テーブルのサフィックス(接尾辞、末尾に付け加える英単語)

・上述のように何も指定しないと、監査テーブル名は「**元のテーブル名**」_**AUD**となります。しかし少々わかりにくいので「**元のテーブル名**」_**AUDITへ変更**します。

・このプロパティ名は本来org.hibernate.envers.audit_table_suffixですが、Enversを Spring Boot経由で実行する場合は、"spring.jpa.properties."を付加して application.propertiesに記述します。これは②も同様です。

②監査テーブルのREVTYPE列名変更

これも内容に即して"Operational Type"を略した**"OP_TYPE"へ変更**します。

> 他にも指定可能なプロパティがあります。たとえばorg.hibernate.envers.audit_table_prefix は監査テーブルのプレフィックス(接頭辞、前に付け加える英単語)を指定できます。
> 興味がある方は「ENVERS properties」などでインターネット検索をしてみるとよいでしょう。

次に操作者の情報を追加した監査テーブルを定義します。

【リスト10-6】com.example.todolist.entity.TodoRevInfo.java

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

import org.hibernate.envers.DefaultRevisionEntity;
import org.hibernate.envers.RevisionEntity;
```

```
import com.example.todolist.config.TodoRevInfoListener;
import jakarta.persistence.Entity;
import lombok.Data;
import lombok.EqualsAndHashCode;

@Entity // ①
@RevisionEntity(TodoRevInfoListener.class)   // ②
@Data
@EqualsAndHashCode(callSuper = false)   // ③
public class TodoRevInfo extends DefaultRevisionEntity {   // ④

    private static final long serialVersionUID = 1L;   // ⑤
    private String opId;   // ⑦

}
```

①エンティティであることを宣言

・これでアプリケーション起動時、spring.jpa.hibernate.ddl-auto=updateにより、このエンティティに対応するテーブルが(無ければ)自動生成されます。

②操作者情報を供給するクラスの指定(**@RevisionEntity**)

・操作者の情報は、後述のTodoRevInfoListenerクラスから供給します。そのクラスをここに指定します。

③スーパークラスのプロパティをequals()/hashCode()に含めない

このTodoRevInfoをインスタンス化して操作することは想定していないので不要とします。

④監査テーブル対応エンティティの宣言

・このエンティティに対応する監査テーブル名は、"TODO_REV_INFO"となります。

→クラス名(キャメルケース)をスネークケースに変換した名前が監査テーブル名となる。

・またDefaultRevisionEntityを継承したクラスとします。この場合自動生成される監査テーブルの列名が、以下のように変わります。

rev → id

revtstmp → timestamp

⑤Serializable対応

継承したDefaultRevisionEntityがSerializableの実装を要求しているため追加。

⑥追加するプロパティ(列)

・文字型の"OP_ID"列(operator idの略)が監査テーブルへ追加されるようにします。

→これもスネークケースへ変換したものが列名となります。
・ここに後述のTodoRevInfoListenerが操作者の情報をセットします。

> ■キャメルケース(camel case)
> ・複数の単語をつなげて書く場合、2語目以降は最初の文字を大文字とする記法
> camel case → CamelCase または camelCase
> ・大文字が「ラクダ(camel)のこぶ」に見えることからそう呼ばれているらしい
> ・大文字で始まるものはアッパーキャメルケース(upper camel case)、またはパスカルケース(pascal case)
> ・小文字で始まるのをローワーキャメルケース(lower camel case)と細分化することもある
> ・Javaの識別子(クラス名、フィールド名、ローカル変数名、...)は、このキャメルケースが一般的
> ■スネークケース(snake case)
> ・単語をアンダーバーでつなぐ記法
> snake case → snake_case
> ・アンダーバーが地を這う蛇(snake)に見えることからそう呼ばれているらしい
> ・Javaの定数(単語は大文字)、SQLでよく使われる(SQLは大/小文字を区別しないのでキャメルケースNG)
> ■ケバブケース(kebab case)
> ・単語をハイフンでつなぐ記法
> kebab case → kebab-case
> ・単語がハイフンで刺されているように見え、肉に串を刺して焼く料理ケバブ(kebab)を連想させるためらしい
> ・HTML/CSSでは一般的な記法
> ・JavaやSQLでは - を減算またはマイナス符号と解釈するためこの記法は使えない(文法エラーとなる)

(2) RevisionListenerの実装

Hibernate Enversへ操作者の情報を提供するTodoRevInfoListenerクラスは以下のようにします。

【リスト10-7】com.example.todolist.config.TodoRevInfoListener.java

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

import org.hibernate.envers.RevisionListener;
import org.springframework.context.annotation.Configuration;
import com.example.todolist.entity.TodoRevInfo;
import jakarta.servlet.http.HttpSession;
import lombok.RequiredArgsConstructor;

@Configuration // ①
@RequiredArgsConstructor
public class TodoRevInfoListener implements RevisionListener {    // ②
    private final HttpSession session;

    @Override
    public void newRevision(Object revisionEntity) {    // ③
        TodoRevInfo todoRevInfo = (TodoRevInfo) revisionEntity;    // ④

        Integer accountId = (Integer) session.getAttribute("accountId");    // ⑤
        if (accountId == null) {
            todoRevInfo.setOpId(null);

        } else {
            todoRevInfo.setOpId("" + accountId);    // ⑥
        }
    }
}
```

①アプリ起動時Spring Bootに取り込んでもらう(**@Configuration**)

②RevisionListenerを実装する

・@RevisionEntityで指定したクラスは、RevisionListenerインターフェースを実装する必要があります。

③監査情報作成時の処理を記述

・Enverは監査テーブルへレコードを出力直前に、このnewRevision()を呼び出します。

→newRevision()はRevisionListenerインターフェースが実装を要求するメソッド

・追加項目である操作者の情報をここでセットすると、それが監査テーブルへ出力されます。

④監査テーブルに対応するエンティティ取得

・引数revisionEntityは、監査テーブルに対応するエンティティを表しますがObject型です。これを実際の型であるTodoRevInfoへキャストします。

⑤操作者情報取得

・セッションからaccountIdを取得します。これを操作者情報とします。

⑥監査エンティティのプロパティへ設定

これで完了です。ToDoアプリを起動後、¥dコマンでテーブルの一覧を表示させると以下の４テーブルが追加されています。

```
tododb=> \d
                リレーション一覧
 スキーマ |         名前          |   タイプ   |  所有者
----------+-----------------------+------------+----------
 public   | account               | テーブル   | todouser
 public   | account_id_seq        | シーケンス | todouser
 public   | attached_file         | テーブル   | todouser
 public   | attached_file_id_seq  | シーケンス | todouser
 public   | category              | テーブル   | todouser
 public   | groups                | テーブル   | todouser
 public   | groups_account        | テーブル   | todouser
 public   | groups_account_id_seq | シーケンス | todouser
 public   | groups_id_seq         | シーケンス | todouser
 public   | revinfo               | テーブル   | todouser
 public   | revinfo_seq           | シーケンス | todouser
 public   | task                  | テーブル   | todouser
 public   | task_aud              | テーブル   | todouser
 public   | task_audit            | テーブル   | todouser   //  (1)追加
 public   | task_id_seq           | シーケンス | todouser
 public   | todo                  | テーブル   | todouser
 public   | todo_aud              | テーブル   | todouser
 public   | todo_audit            | テーブル   | todouser   //  (2)追加
 public   | todo_id_seq           | シーケンス | todouser
 public   | todo_rev_info         | テーブル   | todouser   //  (3)追加
 public   | todo_rev_info_seq     | シーケンス | todouser   //  (4)追加
 public   | v_todolist            | ビュー     | todouser
(22 行)
```

【図10-2】起動後のリレーション一覧(2)

(1)task用監査テーブル(【リスト10-5】①のよりテーブル名＋"_audit"となった)

(2)todo用監査テーブル(【リスト10-5】①のよりテーブル名＋"_audit"となった)

(3)監査テーブル(【リスト10-6】から自動生成)

(4)監査テーブルのシーケンス(テーブル名_seq)

またtodo_audit/task_auditは、revtypeだったところがop_typeになっています(【リスト10-5】②)。

```
tododb=> \d todo_audit
                        テーブル"public.todo_audit"
       列        |         タイプ          | 照合順序 | Null 値を許容 | デフォルト
-----------------+-------------------------+----------+---------------+------------
 id              | integer                 |          | not null      |
 rev             | integer                 |          | not null      |
 op_type         | smallint                |          |               |
 completed_tasks | integer                 |          |               |
 deadline        | date                    |          |               |
 done            | character varying(255)  |          |               |
 importance      | integer                 |          |               |
 is_deleted      | character varying(255)  |          |               |
 owner_id        | integer                 |          |               |
 title           | character varying(255)  |          |               |
 urgency         | integer                 |          |               |
 category_code   | character varying(255)  |          |               |
 category_locale | character varying(255)  |          |               |
 groups_id       | integer                 |          |               |
インデックス:
    "todo_audit_pkey" PRIMARY KEY, btree (rev, id)
外部キー制約:
    "fkht5yn9jlfw1ro4kp7wfcjkxkt" FOREIGN KEY (rev) REFERENCES todo_rev_info(id)


tododb=>
```

この状態でもう一度シナリオ通り操作し、以下のSELECT文で監査情報を確認してみます。

```
SELECT    id, rev, op_type, owner_id, title, importance, urgency,done,
category_code,
            groups_id, is_deleted
   FROM    todo_audit;

SELECT    id, rev, op_type, todo_id, title, deadline, done, assigned_to, progress,
is_deleted
   FROM    task_audit;
```

■実行結果

```
tododb=> select id, rev, op_type, owner_id,title, importance, urgency, done, category_code, groups_id, is_deleted from todo_audit;
 id | rev | op_type | owner_id | title  | importance | urgency | done | category_code | groups_id | is_deleted
----+-----+---------+----------+--------+------------+---------+------+---------------+-----------+------------
  5 |   1 |       0 |        1 | Envers |          1 |       1 | N    | 10            |         0 | N
  5 |   2 |       1 |        1 | Envers |          1 |       1 | N    | 20            |       100 | N
  5 |   3 |       1 |        1 | Envers |          1 |       1 | N    | 20            |       100 | N
  5 |   6 |       1 |        1 | Envers |          1 |       1 | N    | 20            |       100 | Y
(4 行)


tododb=> select id, rev, op_type, todo_id, title, deadline, done, assigned_to, progress, is_deleted from task_audit;
 id | rev | op_type | todo_id | title  | deadline | done | assigned_to | progress | is_deleted
----+-----+---------+---------+--------+----------+------+-------------+----------+------------
  4 |   3 |       0 |       5 | task-1 |          | N    |           0 |        0 | N
  4 |   4 |       1 |       5 | task-1 |          | N    |           3 |        0 | N
  4 |   5 |       1 |       5 | task-1 |          | N    |           3 |        0 | Y
(3 行)
```

このように途中の変更者も把握できるようになったことがわかると思います。

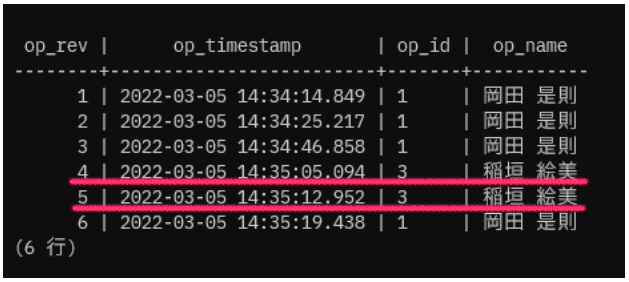
たとえばtask_auditを見るとtask.id=4の監査情報が出力されていますが、前の行と比較すると

・2行目は担当者(assigned_to)を変更している(0→3)

・3行目は論理削除(is_deleted)している(N→Y)

ことがわかります。そして2,3行目のrevは4,5です。これをtodo_rev_infoと照らし合わせてみます。

```
SELECT   tri.id   AS   op_rev,
         timezone('JST',
           to_timestamp('1970-01-01 09:00:00', 'YYYY-MM-DD HH24:MI:SS.sss')
                        + timestamp * interval '1 millisecond')
AS   op_timestamp,
         op_id,
         a.name   AS   op_name
  FROM todo_rev_info tri
  JOIN account a   ON cast(tri.op_id AS integer ) = a.id
ORDER   BY   tri.id;
```



```
 op_rev |      op_timestamp       | op_id |  op_name
--------+-------------------------+-------+-----------
      1 | 2022-03-05 14:34:14.849 | 1     | 岡田 是則
      2 | 2022-03-05 14:34:25.217 | 1     | 岡田 是則
      3 | 2022-03-05 14:34:46.858 | 1     | 岡田 是則
      4 | 2022-03-05 14:35:05.094 | 3     | 稲垣 絵美
      5 | 2022-03-05 14:35:12.952 | 3     | 稲垣 絵美
      6 | 2022-03-05 14:35:19.438 | 1     | 岡田 是則
(6 行)
```

どちらもop_id=3です。よってaccount.id=3のユーザーによって行われた操作であり、いつ行ったか？もop_timestampに記録されています。

10.3 注意事項

■注意 1

本章では監査テーブルを作成するのにspring.jpa.hibernate.ddl-autoを利用しましたが、本番環境での使用は推奨されていません。開発・テスト環境で生成したものを移行するか、あるいはスクリプトを作成→本番環境で実行、などほかの手段を検討してください。

■注意 2

監査情報を出力していても、それが改ざんされては何の意味もありません。特定のユーザー(システム管理者)しか操作できないようアクセス権限を設定する、別のスキーマへ出力する、などの対策が必要です。

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]

著者:原田 けいと, 竹田 甘地, Robert Segawa / 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]

著者: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)

著者: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 Boot3で始めるWebアプリケーション開発入門(応用編)」(Amazon 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(発展編)

2023年1月7日　初版発行

著者　　菊田英明
発行者　菊田英明