

Java EE 7 入門

NetBeans で始める Java EE 7 First Tutorial

日本 Java ユーザ・グループ[°]

2013/08/24

目次

第 1 章 はじめに	5
[1-1] このチュートリアルで学ぶこと	5
[1-2] このチュートリアルの対象読者	5
[1-3] 検証環境	6
[1-4] アーキテクチャ概要	6
第 2 章 作成するアプリケーションの説明	7
[2-1] アプリケーションの概要	7
[2-2] アプリケーションの業務要件	8
[2-3] アプリケーションの画面遷移	8
[2-4] アプリケーションの画面要件	9
■ Todo 全件表示	9
■ Todo 新規作成	9
■ Todo 完了	9
■ Todo 削除	9
[2-5] エラーメッセージ一覧	10
第 3 章 環境構築	10
[3-1] NetBeans のインストール	10
■ Mac 版	12
■ Windows 版	17
[3-2] データベース作成	26

[3-3] プロジェクトの作成	32
[3-4] 動作確認	38
[3-5] プロジェクト構成	40
第 4 章 Todo アプリケーションの作成.....	41
[4-1] JPA の Entity 作成	41
■ リバースエンジニアリングで DB からエンティティ生成	42
■ 生成されたソースの修正	49
[4-2] EJB で業務処理を実装	59
■ SessionBean の作成	59
■ JUnit の作成	62
■ 業務処理の実装	71
[4-3] JSF で画面を作成	84
■ ManagedBean の作成	84
■ Facelets の作成	87
■ 日本語の文字化けに対応する	89
■ Todo 全件表示	91
■ Todo 新規作成	100
■ Todo 完了	119
■ Todo 削除	129
[4-4] JAX-RS で REST API を作成	136
■ Dev HTTP Client のインストール	136
■ リソースクラスの作成	139
■ GET Todos	145
■ GET Todo	147
■ POST Todos	154
■ PUT Todo	159
■ DELETE Todo	163
■ 入力チェックの追加	166

第 5 章 WebSocket でリアルタイムイベントモニタリング機能追加	179
[5-1] WebSocket エンドポイントの作成	179
■ エンドポイントクラスを作成	179
■ WebSocket の JavaScript API でモニタ画面作成.....	183
[5-2] CDI によるイベント操作.....	191
■ TodoEvent 用修飾子の作成	191
■ Event 用モデルを作成	195
■ イベントの発火	197
■ イベントの購読	200
第 6 章 おわりに	203

第1章 はじめに

[1-1] このチュートリアルで学ぶこと

- Java EE7による基本的なアプリケーションの開発方法およびNetBeansプロジェクトの構築方法
- JPA, EJB, JSF, JAX-RS, WebSocketによる簡単なアプリケーション作成方法

[1-2] このチュートリアルの対象読者

- Java で Web アプリケーション(Servlet/JSP または Struts などの Web フレームワーク)開発経験がある
- SQL が分かる
- Java EE 6/7 の名前を聞いたことがあるが使ったことがなく、まずは試してみたい

尚、本チュートリアルは JavaEE の個別技術(JPA,EJB,JSF など)の詳細説明は行わないため、本チュートリアル中に不明な用語がある場合、適宜別のドキュメントを参照してください。ただし、基本的には記述内容に従い操作を行うことでアプリケーションの作成は可能です。本チュートリアルは、まずは手を動かしてアプリケーションを作成して Java EE の開発に触れていただく事を目的としています。必要に応じて個別に深掘りしていくことを推奨します。

[1-3] 検証環境

このチュートリアルは以下の環境で動作確認しています。

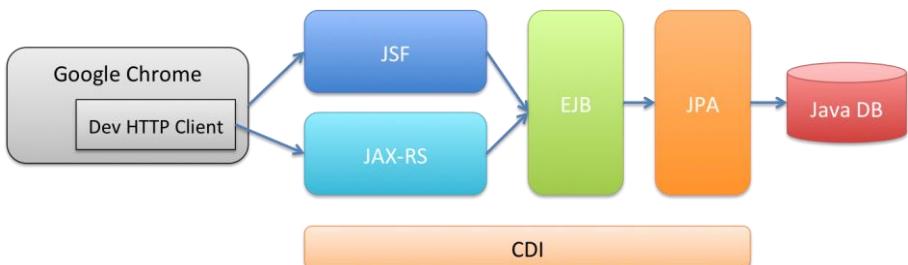
種別	名前
OS	OS X 10.8.4
JVM	Java 1.7
IDE	NetBeans 7.3.1
AP サーバー	GlassFish Server Open Source Edition 4.0
Web ブラウザ	Google Chrome 27.0.1453.94 m
RESTクライアント	Dev HTTP Client 0.6.9.7

必ずしも上記環境に合わせる必要はありませんが、Dev HTTP Client は Chrome の拡張機能であるため、ブラウザを Chrome にする必要があります。

[1-4] アーキテクチャ概要

本チュートリアルは、下記のアーキテクチャでアプリケーションを開発します。

EJB が業務処理を行い、JPA が永続化処理を行います。この EJB に、JSF(画面)と JAX-RS(REST-API)でアクセスします。これらは疎結合に実装され、CDI で繋ぎあわせます。



第5章ではWebSocketによる機能追加も行います。

第2章 作成するアプリケーションの説明

[2-1] アプリケーションの概要

TODOを管理するアプリケーションを作成します。本アプリケーションはTODOの一覧表示、登録、完了、削除機能を持っています。

以下に画面のイメージを示します。

Create Todo

Title:

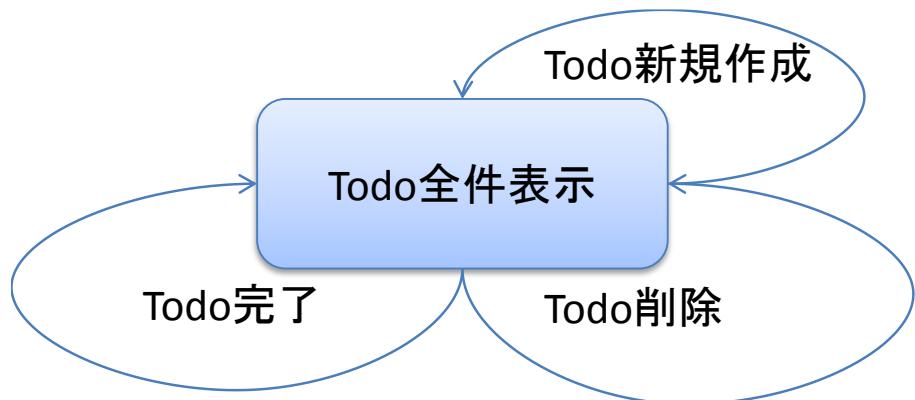
Todo List

Title	Created At	Actions
clean desktop	Sun Jun 23 23:27:13 JST 2013	<input type="button" value="Done"/> <input type="button" value="Delete"/>
write a document	Sun Jun 23 23:27:19 JST 2013	<input type="button" value="Done"/> <input type="button" value="Delete"/>
send a e-mail	Sun Jun 23 23:27:33 JST 2013	<input type="button" value="Delete"/>

[2-2] アプリケーションの業務要件

- B01 未完の TODO は 5 件までしか登録できない
- B02 完了済みの TODO は完了できない

[2-3] アプリケーションの画面遷移



項目番	処理名	補足
1	Todo 全件表示	
2	Todo 新規作成	作成後 1 ヘリダイレクト
3	Todo 完了	完了後 1 ヘリダイレクト
4	Todo 削除	削除後 1 ヘリダイレ

[2-4] アプリケーションの画面要件

■ Todo 全件表示

- TODO を全件表示する
- 未完了の TODO に対しては”Done”と”Delete”用のボタンが付く
- 完了の TODO は打ち消し線で装飾する
- TODO の件名のみ

■ Todo 新規作成

- フォームから送信された TODO を保存する
- TODO の件名は 1 文字以上 30 文字以下であること
- [2-2]の B01 を満たさない場合はエラーコード E001 でビジネス例外をスローする

■ Todo 完了

- 指定の todoId に対応する TODO を完了済みにする
- [2-2]の B02 を満たさない場合はエラーコード E002 でビジネス例外をスローする
- 該当する TODO が存在しない場合はエラーコード E404 で ResourceNotFoundException 例外をスローする

■ Todo 削除

- 指定の todoId に対応する TODO を削除する
- 該当する TODO が存在しない場合はエラーコード E404 で

ResourceNotFound 例外をスローする

[2-5] エラーメッセージ一覧

エラーコード	メッセージ	置換パラメータ
E001	The count of un-finished Todo must not be over {0}.	{0}... max unfinished count
E002	The requested Todo is already finished. (id={0})	{0}... todoId
E404	The requested Todo is not found. (id={0})	{0}... todoId

第3章 環境構築

[3-1] NetBeans のインストール

<https://netbeans.org/downloads/>

から Java EE 版をダウンロードしてください。

NetBeans IDE ダウンロードバンドル					
サポートテクノロジー *	Java SE	Java EE	C/C++	PHP	すべて
④ NetBeansプラットフォームSDK	●	●			●
④ Java SE	●	●			●
④ Java FX	●	●			●
④ Java EE		●			●
④ Java ME					—
④ HTML5		●		●	●
④ Java Card(TM) 3 Connected			●		—
④ C/C++			●		●
④ Groovy					●
④ PHP				●	●
バンドル サーバー					
④ GlassFish Server Open Source Edition 4.0		●			●
④ Apache Tomcat 7.0.34	●				●
ダウンロード		ダウンロード	ダウンロード	ダウンロード	ダウンロード
無償, 76 MB		無償, 190 MB	無償, 48 MB	無償, 48 MB	無償, 203 MB

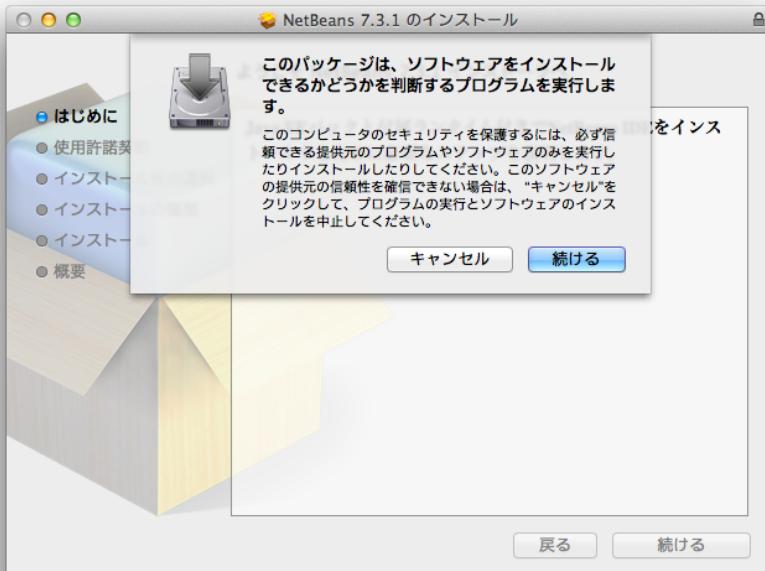
■ Mac 版

ダウンロードした「NetBeans 7.3.1.mpkg」ファイルをダブルクリックしインストールします。



NetBeans 7.3.1.mpkg

「続ける」ボタンを押下してください。



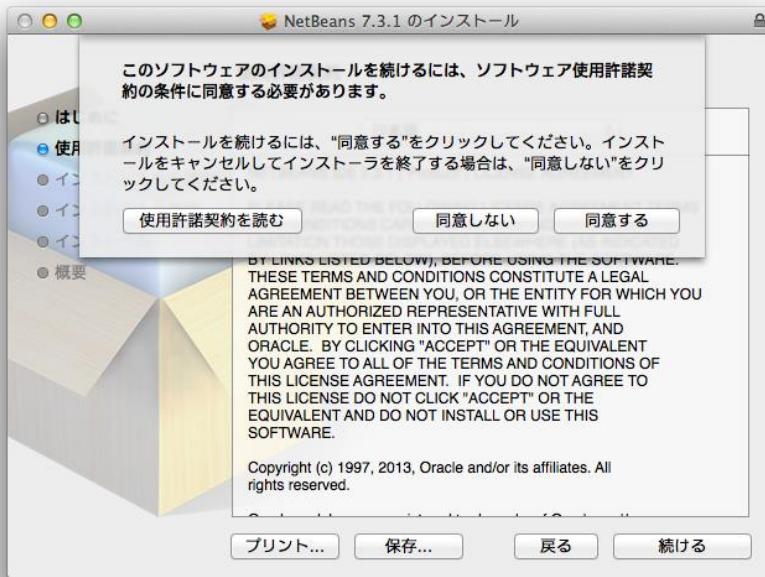
「続ける」ボタンを押下してください。



「続ける」ボタンを押下してください。



「同意する」ボタンを押下してください。



最後に、「インストール」ボタンを押下してください。

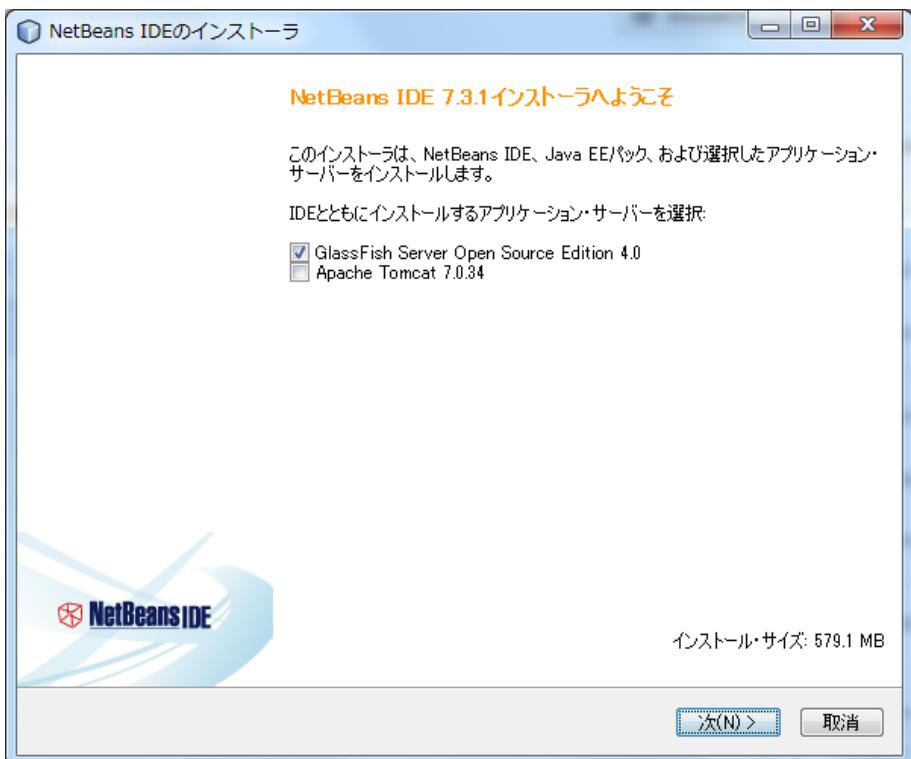


以下略・・

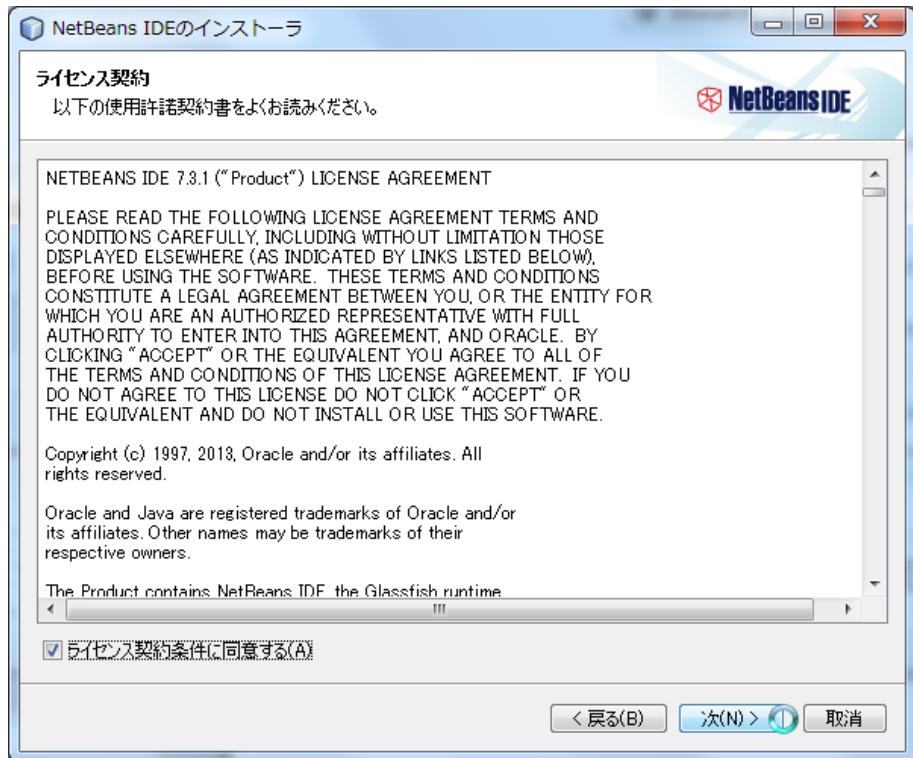
■ Windows 版

ダウンロードした「netbeans-7.3.1-javasee-windows.exe」ファイルをダブルクリックでインストールします。

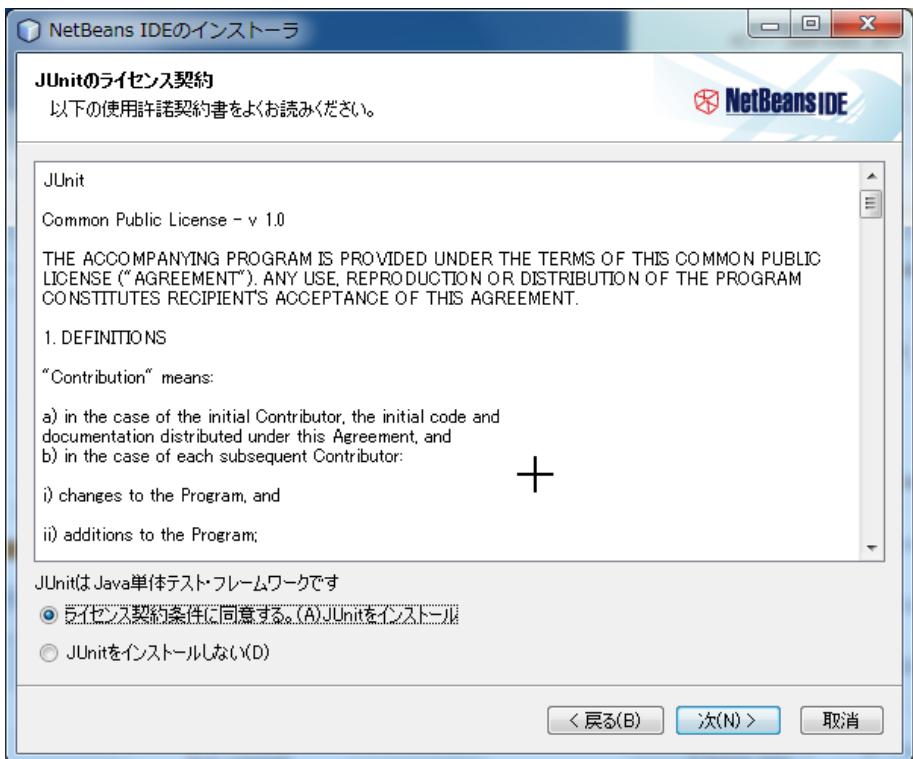
「次(N)>」ボタンを押下してください。



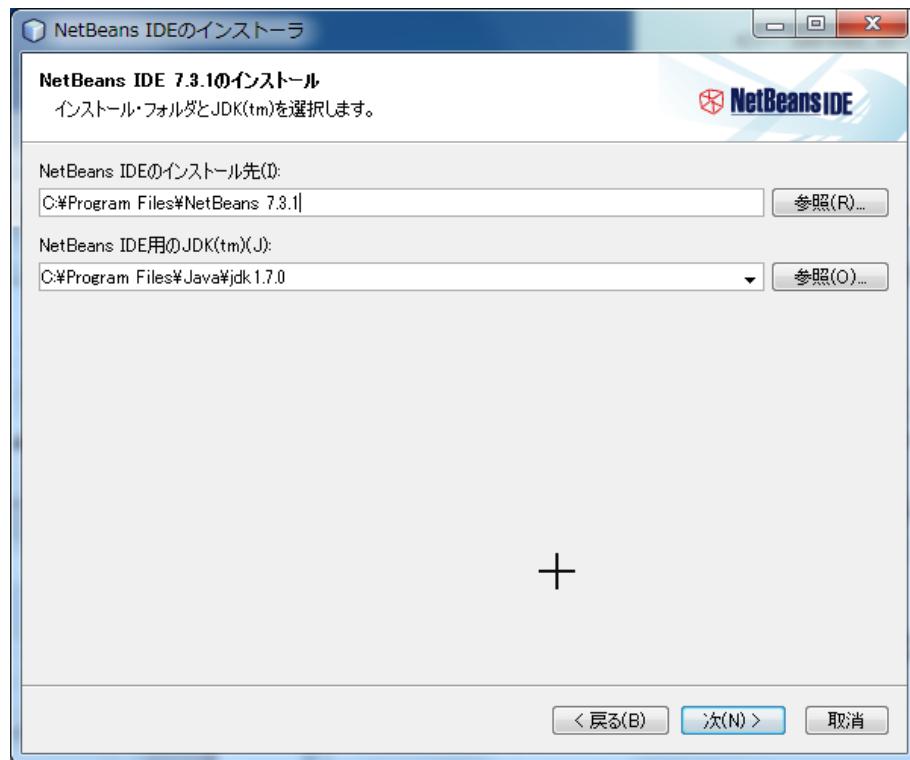
「ライセンス契約条件に同意する」をチェックして、「次(N)>」ボタンを押下してください。



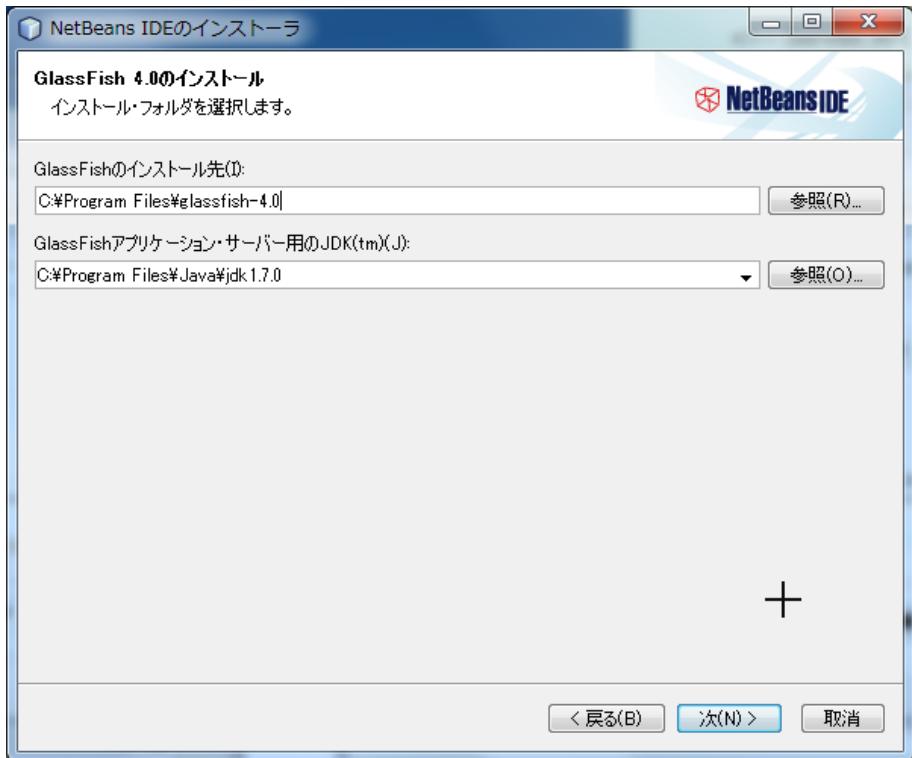
「ライセンス契約条件に同意する。(A)JUnit をインストール」をチェックして、「次(N)>」ボタンを押下してください。



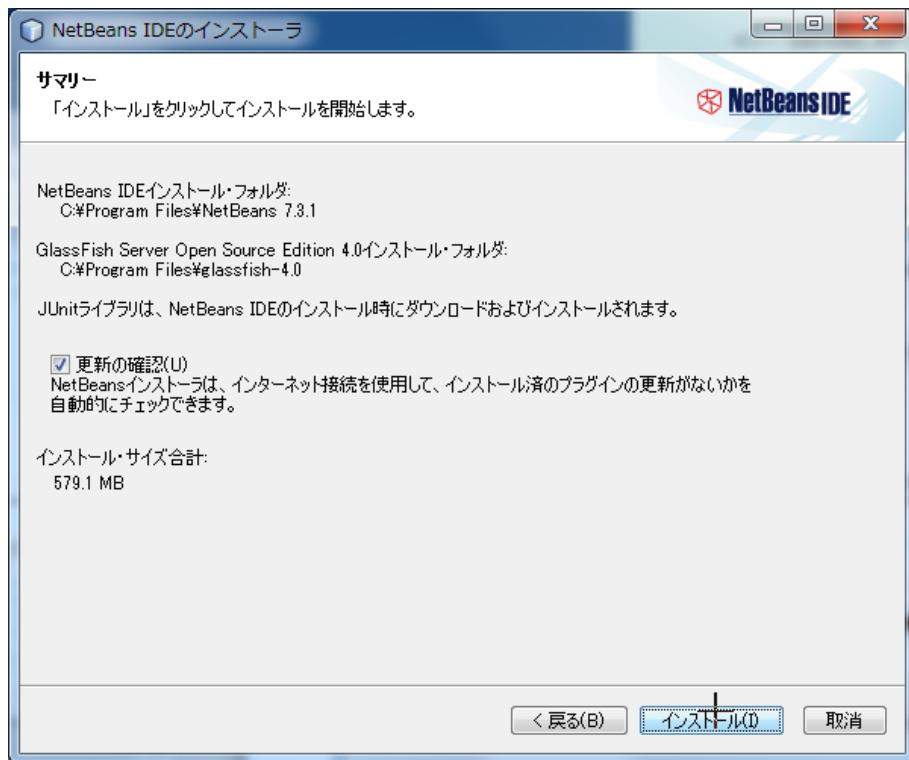
「次(N)>」ボタンを押下してください。



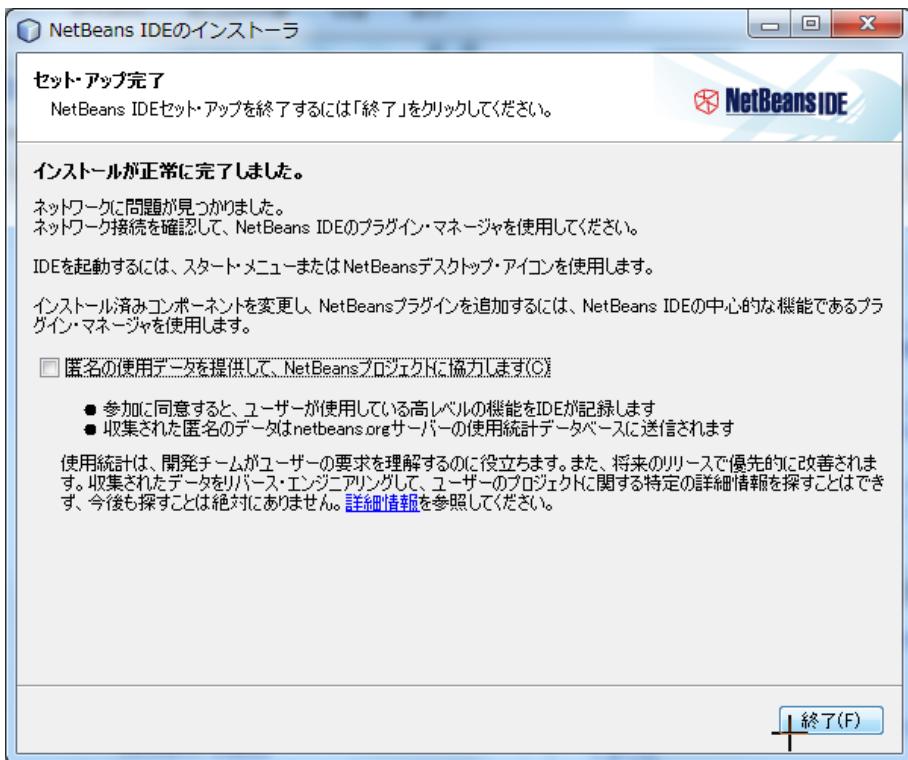
「次(N)>」ボタンを押下してください。



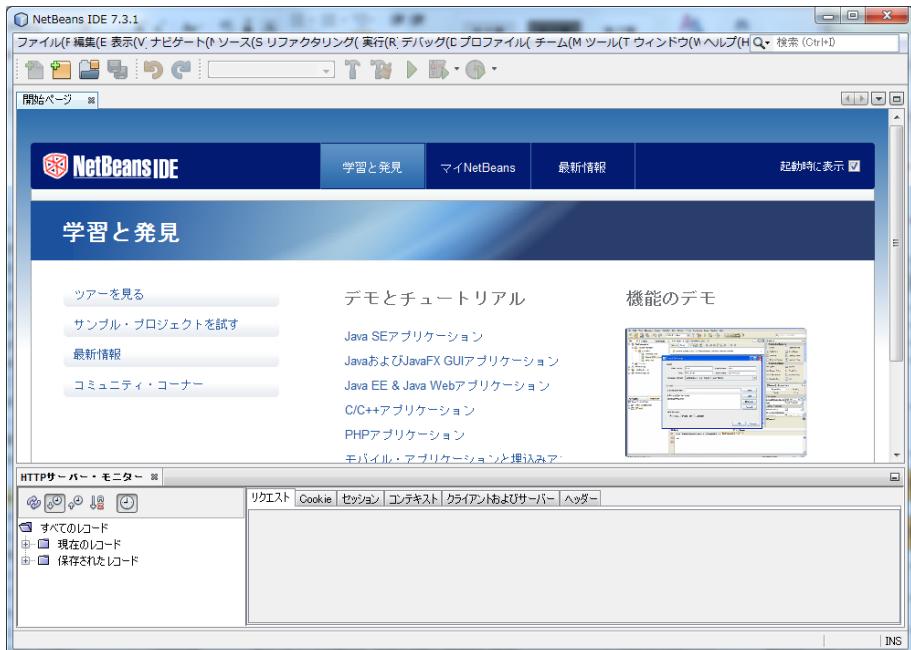
「インストール(I)」ボタンを押下してインストールしてください。



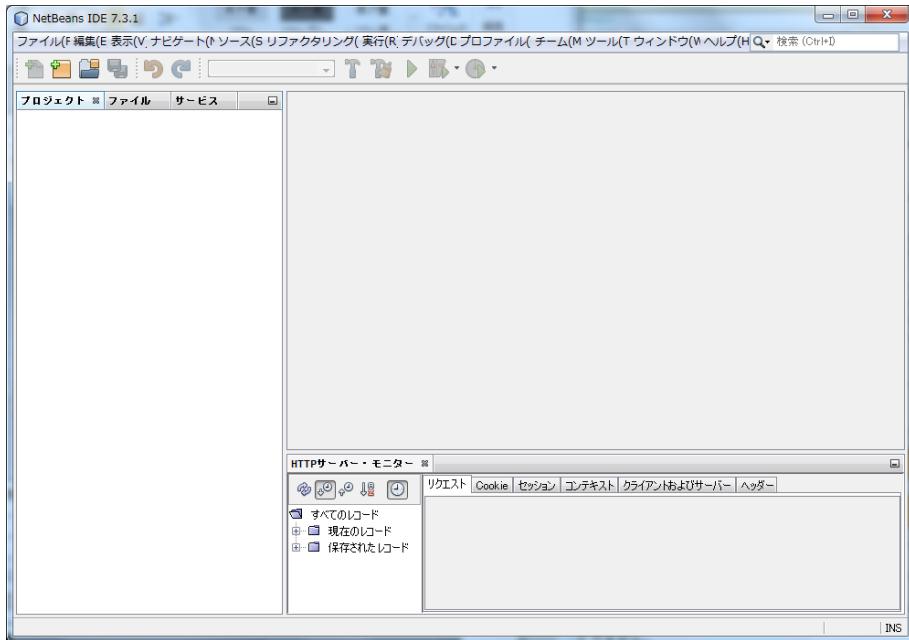
最後に「終了(F)」ボタンを押下しインストールを完了します。



インストールした NetBeans のアイコンをダブルクリックし起動してください。



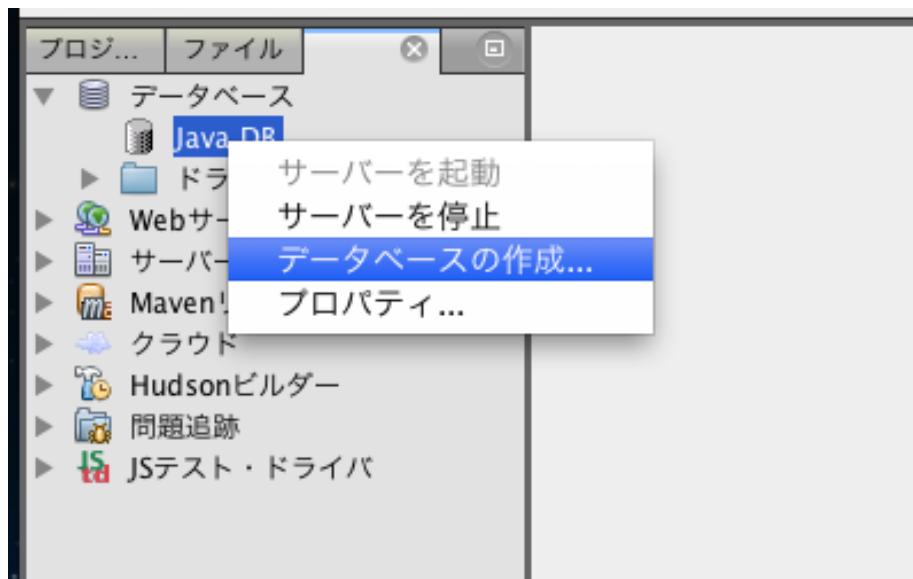
「開始ページ」タブの(×)を押下し、開始ページを閉じてください、するとプロジェクトウインドウが表示されます。



[3-2] データベース作成

本チュートリアルで使用するデータベースを作成します。データベースは NetBeans のインストール時に自動的に組み込まれている Java DB (Derby) を使用します。

「サービス」ウインドウを開き、「データベース」→「Java DB」を右クリックして、「データベースの作成」をクリックしてください。

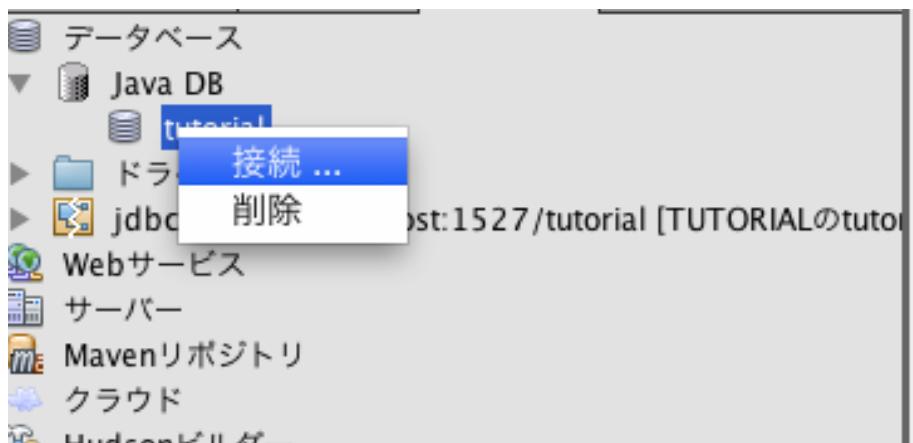


「Java DB データベースを作成」 ウィンドウで下記のを入力して、「OK」ボタンを押下してください。

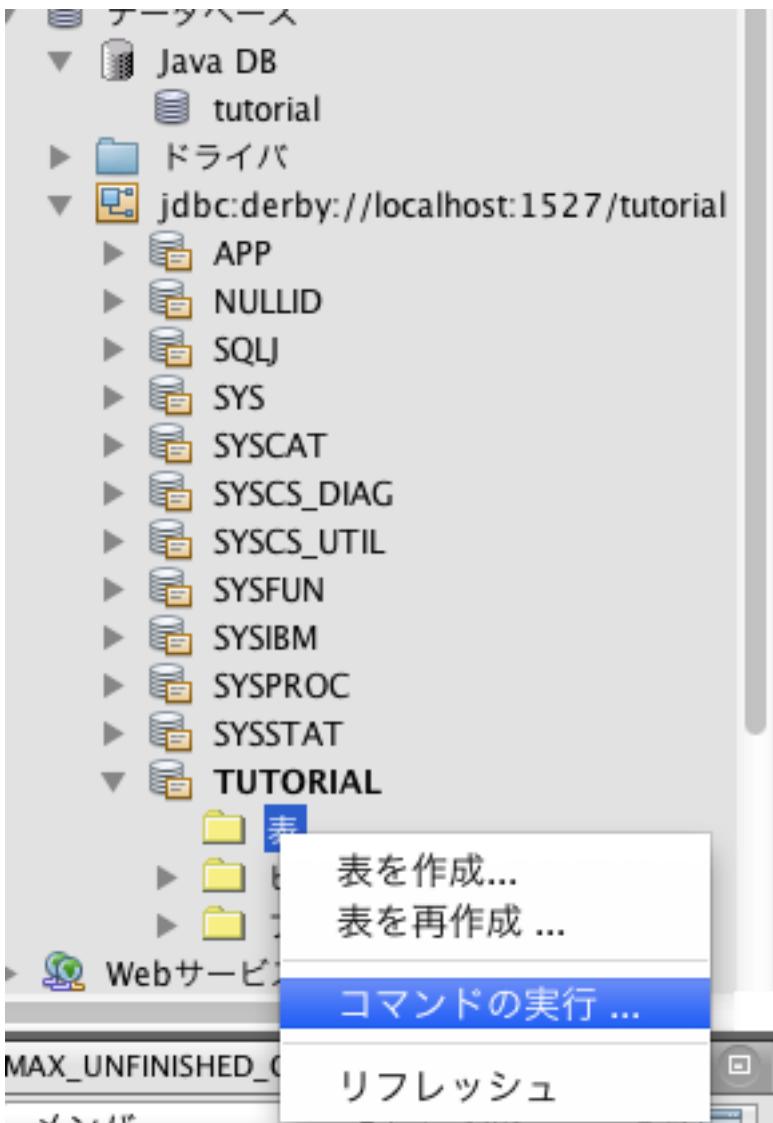
入力項目	入力値
データベース名	tutorial
ユーザ名	tutorial
パスワード	tutorial
パスワードの確認	tutorial



作成すると、「Java DB」のツリー配下に「tutorial」データベースが作成されます。「tutorial」データベースを右クリックして「接続」をクリックしてください。



スキーマ一覧から「TUTORIAL」を選択したのち展開してください。ツリーの一覧から「表」を右クリックで選択したのち、「コマンドの実行 ...」をクリックしてください。



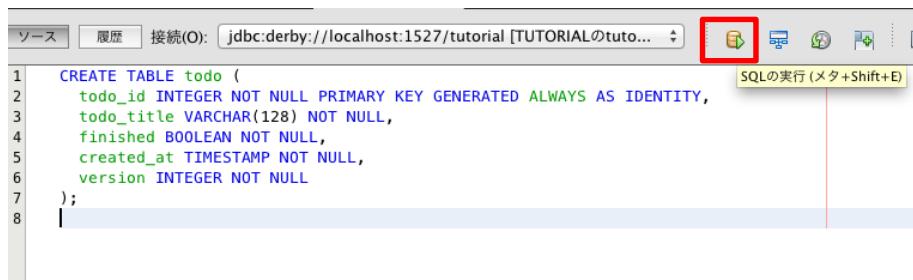
今回は Todo モデルとして以下のフィールドを設けます。

- ID
- タイトル
- 完了フラグ
- 作成時刻
- バージョン(楽観ロック用)

このモデルの DDL は以下です。

```
CREATE TABLE todo (
    todo_id INTEGER NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    todo_title VARCHAR(128) NOT NULL,
    finished BOOLEAN NOT NULL,
    created_at TIMESTAMP NOT NULL,
    version INTEGER NOT NULL
);
```

これを「SQL コマンド」ウインドウに貼り付け、「SQL の実行(メタ + Shift + E)」ボタンを押下してください。



The screenshot shows a SQL command window with the following details:

- Toolbar buttons: Source (ソース), History (履歴), Connect (接続(O: jdbc:derby://localhost:1527/tutorial [TUTORIALのtuto...]), Execute (実行), Stop (停止), Refresh (リフレッシュ), Save (保存), and Help (ヘルプ).
- Execute button: A green button with a white icon and the text "SQLの実行 (メタ+Shift+E)" is highlighted with a red box.
- Code area: The code area contains the SQL DDL statement for creating the "todo" table.
- Status bar: The status bar at the bottom shows the number of rows affected: "8 行" (8 rows).

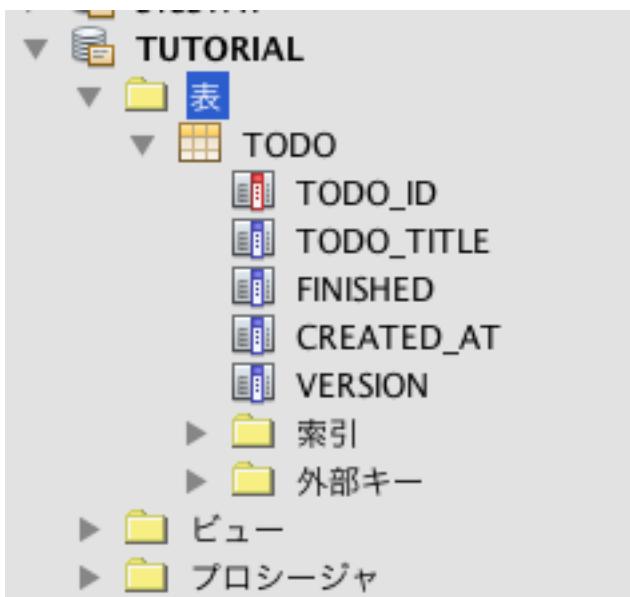
```
1 CREATE TABLE todo (
2     todo_id INTEGER NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
3     todo_title VARCHAR(128) NOT NULL,
4     finished BOOLEAN NOT NULL,
5     created_at TIMESTAMP NOT NULL,
6     version INTEGER NOT NULL
7 );
8 |
```

以下のようなログが出力されればコマンドの実行は成功です。

The screenshot shows the GlassFish Server View interface. At the top, there are tabs for 'テスト結果' (Test Results) and '出力' (Output). The 'Java DBデータベース・プロセス' (Java DB Database Process) tab is active, displaying the following log output:

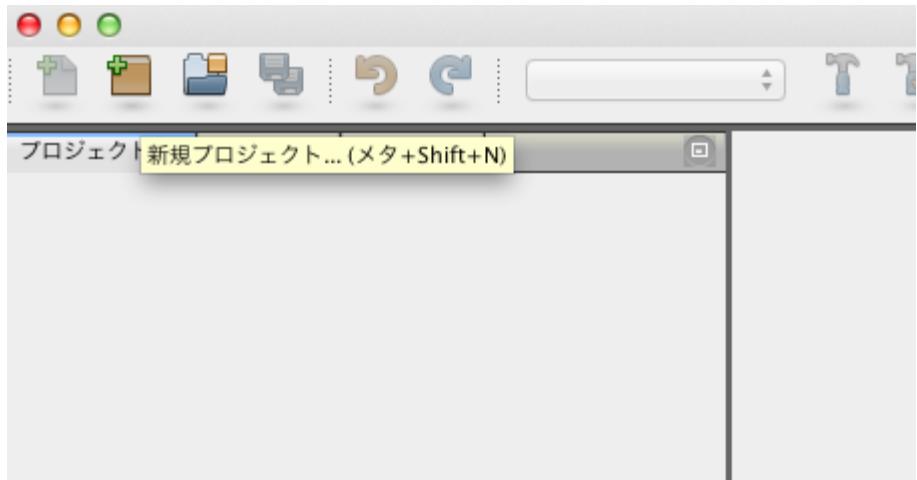
```
0.02秒で実行が成功しました。0行が変更されました。  
1行目1文字目  
0.02秒後に実行が終了しました。0個のエラーが発生しました。
```

「表」の下に「TODO」テーブルが作成されていることを確認してください。

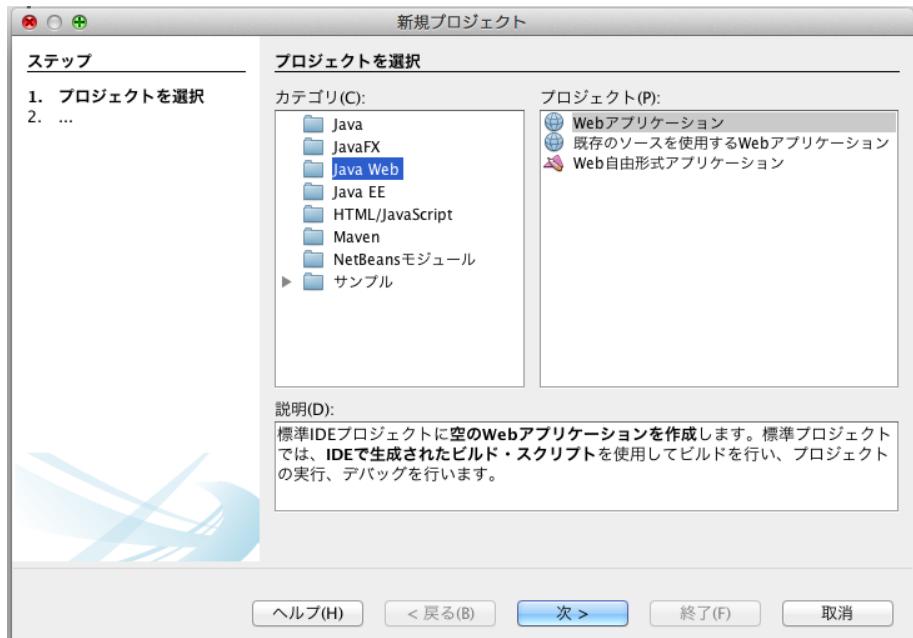


[3-3] プロジェクトの作成

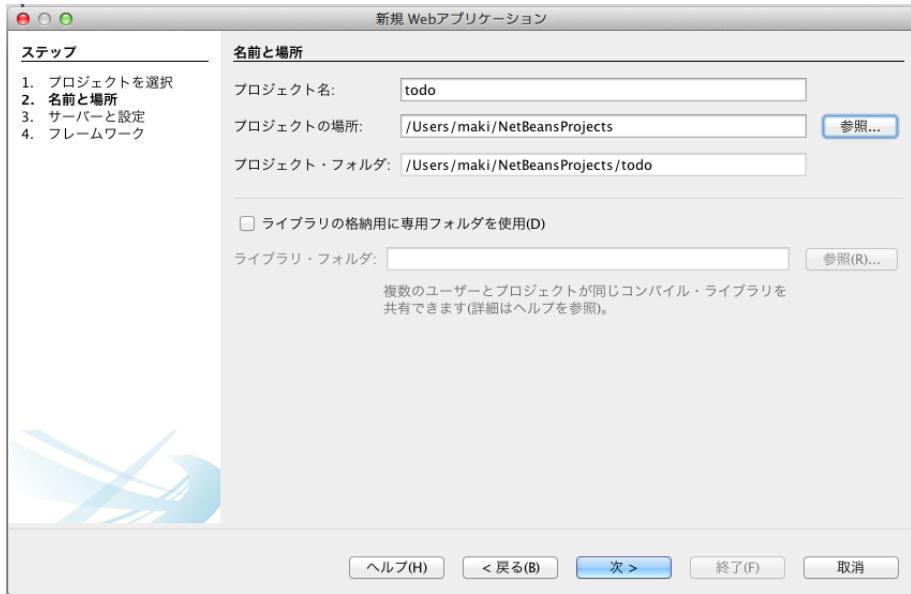
チュートリアルプロジェクトを作成します。新規プロジェクトボタンを押下してください。



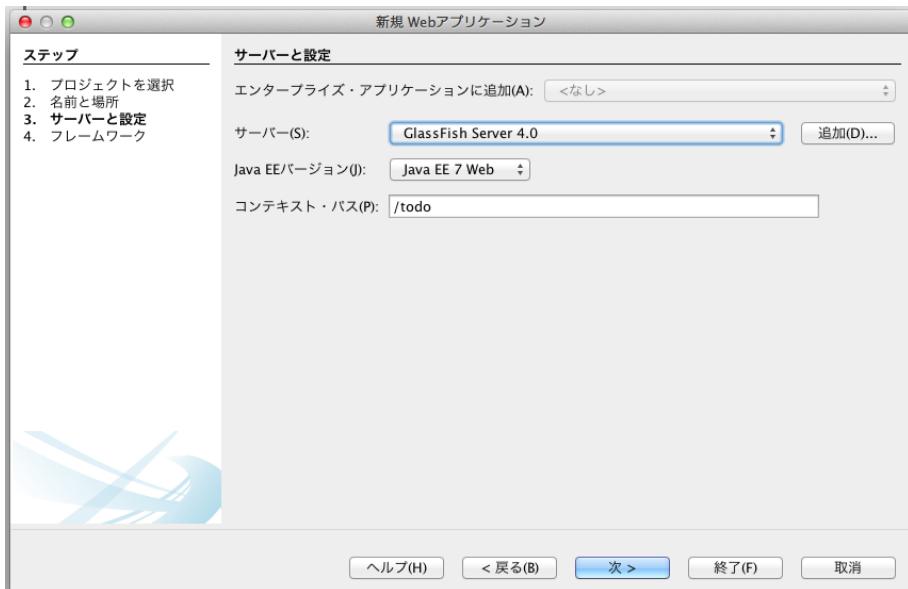
「新規プロジェクト」ウィンドウが表示されます。「Java Web」→「Web アプリケーション」を選択して「次 >」ボタンを押下してください。



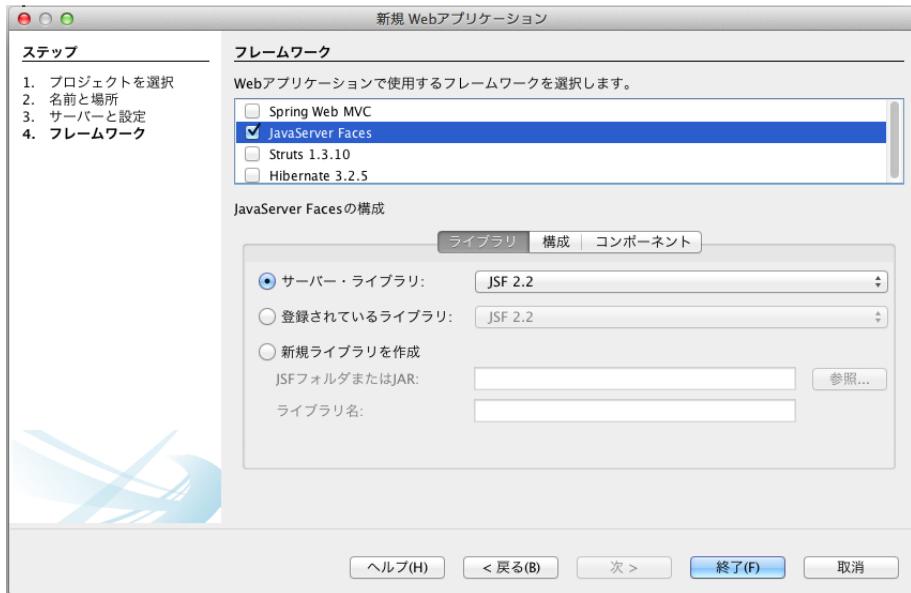
プロジェクト名に「todo」と入力したのち、「次 >」ボタンを押下してください。



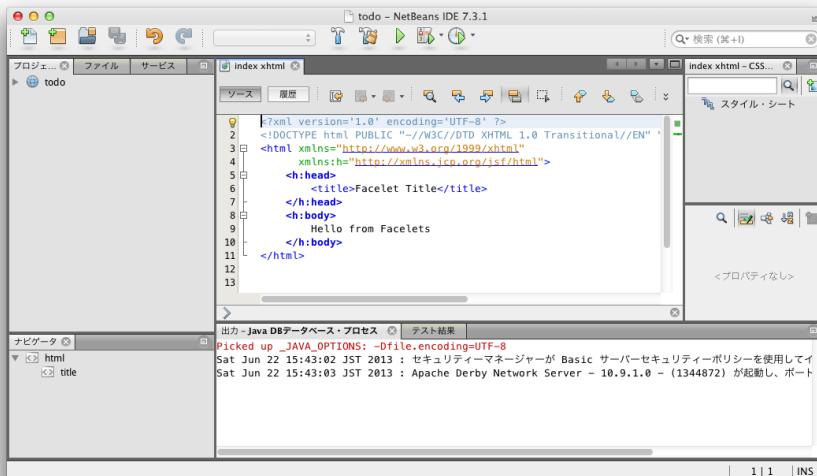
「次 >」ボタンを押下してください。



「フレームワーク」の選択画面より「JavaServer Faces」にチェックし「終了(F)」ボタンを押下してください。

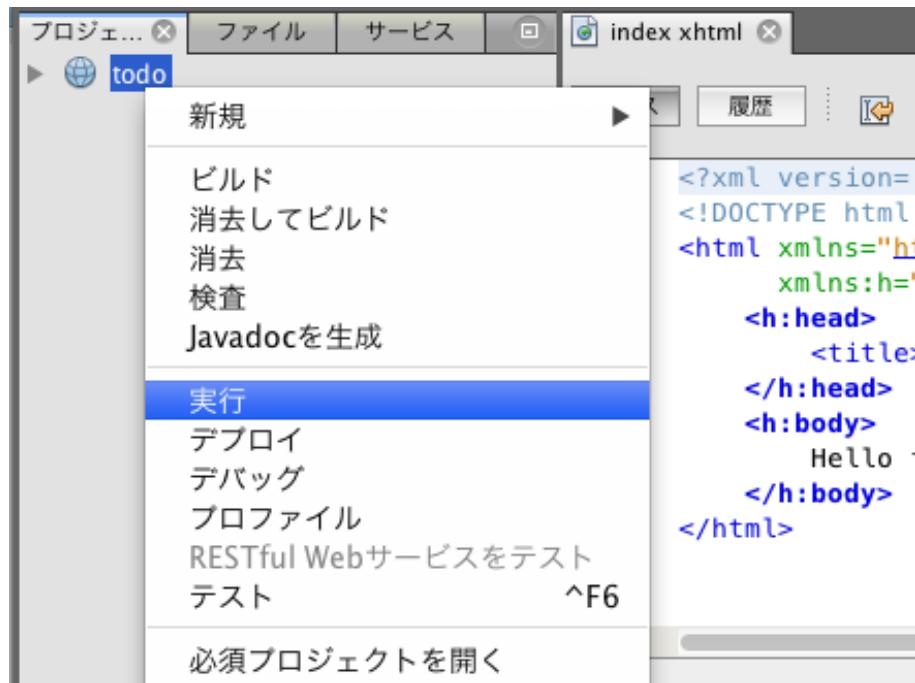


プロジェクトウィンドウに todo プロジェクトが表示されます。

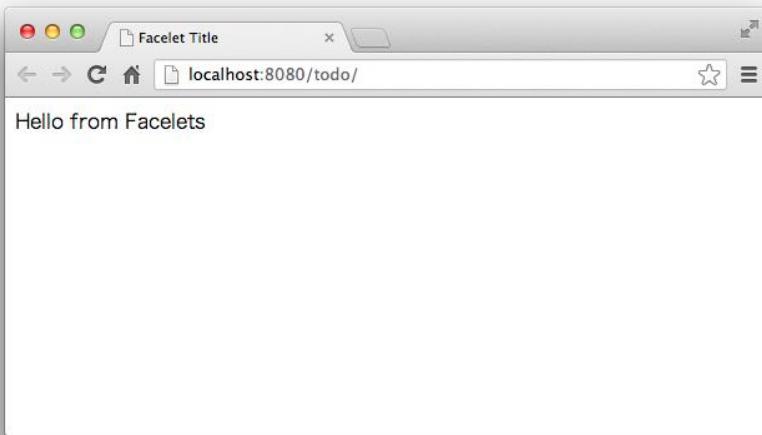


[3-4] 動作確認

todo プロジェクトを右クリックして「実行」をクリックしてください。



ブラウザが自動的に起動し、"Hello from Facelets"が表示されます。



[3-5] プロジェクト構成

次章より以下のようなプロジェクト構成でアプリケーションを作成していく。

src

```
└ todo ... TODO アプリケーションのベースパッケージ
    ┌ app ... アプリケーション(web)層を格納
    |   ┌ common ... アプリケーション層の共通クラス
    |   を格納
    |   ┌ todo ... todo 管理業務に関わる ManagedBean
    |   や JAX-RS のリソースクラスを格納
    |   ┌ domain ... ドメイン層を格納
    |   |   ┌ common ... ドメイン層の共通クラス
    |   |   ┌ model ... DomainObject を格納
    |   ┌ service ... EJB を格納
    |       ┌ todo ... TODO 業務 EJB
```

web

```
└ WEB-INF
    ┌ todo ... todo 管理業務に関わる facelets(xhtml) を格納
```

Memo: ドメイン層をビジネス層、アプリケーション層をプレゼン層と呼ぶ方が自然に感じる場合は読み替えてよい。

第4章 Todo アプリケーションの作成

Todo アプリケーションを作成します。作成する順は下記の通りです。

1. ドメイン層
 - (ア) DomeinObject(JPA の Entity)作成
 - (イ) EJB 作成
2. アプリケーション層
 - (ア) JSF 作成
 - ① ManagedBean 作成
 - ② Facelets 作成
 - (イ) JAX-RS 作成

[4-1] JPA の Entity 作成

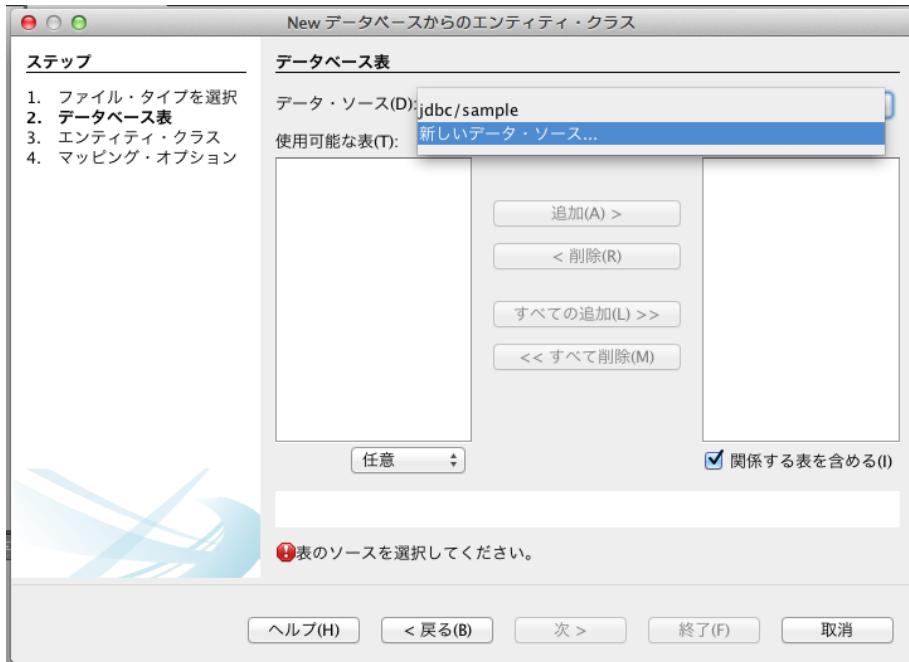
JPA で todo を永続化するために Entity を作成します。今回は既に DB を作成しているため、作成済みの DB からエンティティを自動生成します。

■ リバースエンジニアリングで DB からエンティティ生成

todo プロジェクトを右クリックして「新規」→「その他」で「持続性」→「データベースからのエンティティ・クラス」を選択し、「次 >」ボタンを押下してください。

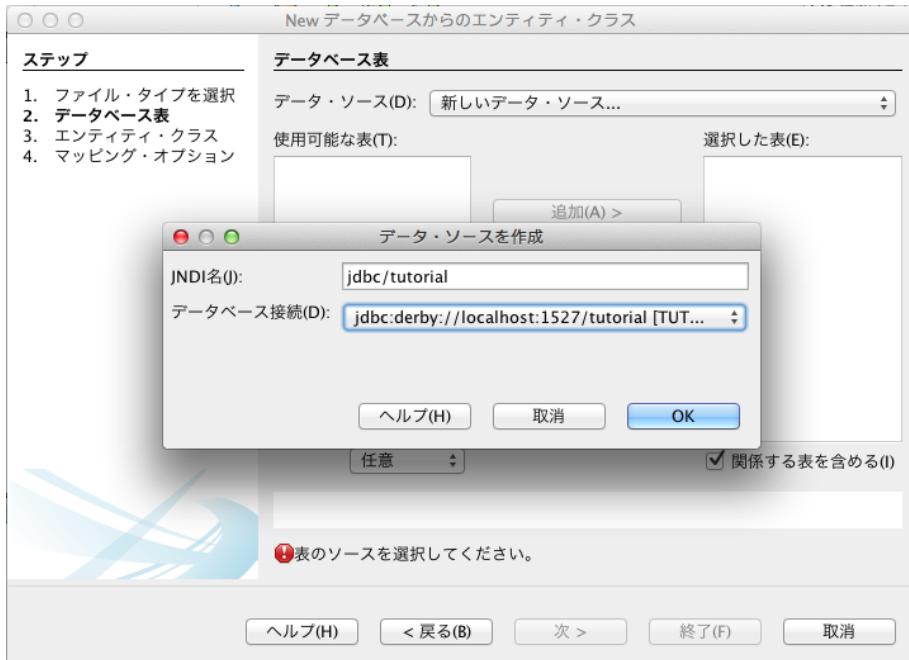


「New データベースからのエンティティ・クラス」ウィンドウにおいて、「データ・ソース(D)」のコンボボックスから「新しいデータ・ソース...」を選択します。



「データ・ソースを作成」ウィンドウ内で下記の内容を入力し「OK」ボタンを押下してください。

入力項目	入力値
JNDI名(J)	jdbc/tutorial
データベース接続(D)	jdbc:derby://localhost:1527/tutorial



「OK」ボタンを押下すると下記の画面が表示され、JNDI 名 : jdbc/tutorial で接続する DB 内に存在する使用可能な表の一覧が表示されます。



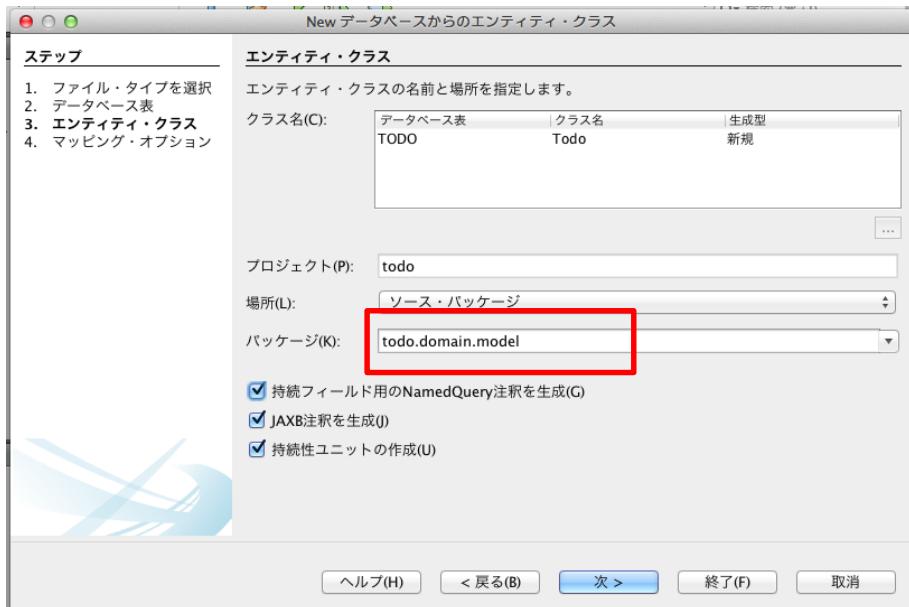
「使用可能な表(T)」から TODO を選択して「追加(A) >」をクリックしてください。

「選択した表(E)」に TODO が移動したことを確認したのち、「次 >」ボタンを押下してください。

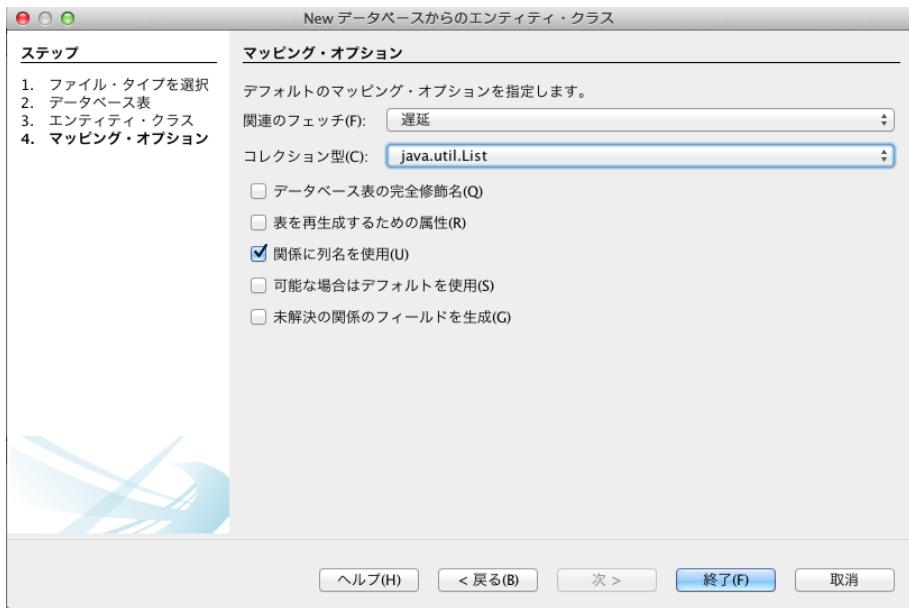


「エンティティ・クラス」の設定画面にて下記を入力し「次 >」ボタンを押下してください。

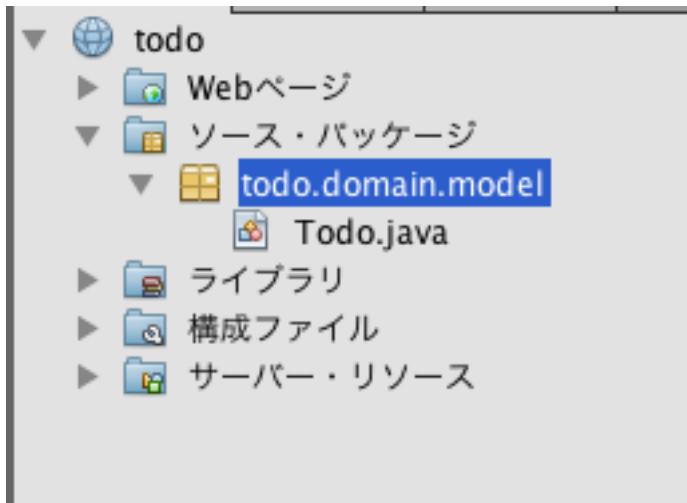
入力項目	入力値
パッケージ	todo.domain.model



最後に「終了(F)」ボタンを押下します。



「todo」プロジェクト内の「ソース・パッケージ」ツリー配下に
Todo クラスが生成されます。



■ 生成されたソースの修正

ウィザードから自動生成されたソース・コードを修正します。

- フィールドとコンストラクタの修正

finished の型を char から boolean に変更します。

```
private String todoTitle;  
@Basic(optional = false)  
@NotNull  
@Column(name = "FINISHED")  
private boolean finished;  
@Basic(optional = false)  
@NotNull  
@Column(name = "CREATED_AT")  
-- -- -- -- --
```

図：フィールドの修正

```
public Todo(Integer todoId, String todoTitle, boolean finished, Date createdAt) {  
    this.todoId = todoId;  
    this.todoTitle = todoTitle;  
    this.finished = finished;  
    this.createdAt = createdAt;  
}
```

図 4-1:コンストラクタの修正

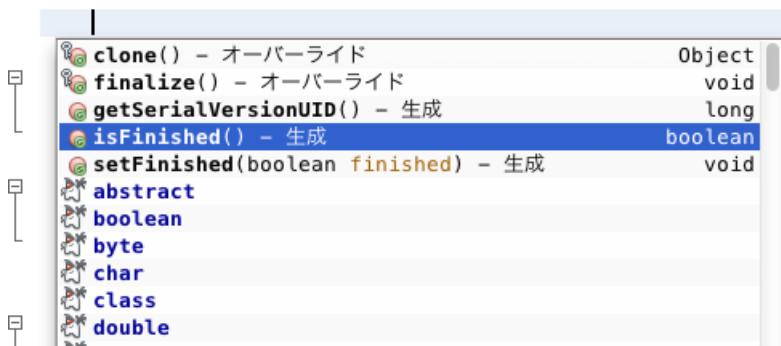
- `getFinished` と `setFinished` を削除し getter/setter 再生成

フィールド「finished」の「boolean 型」への変更に伴い getter/setter を再作成します。まずは、下記の既存コードを削除してください。

```
82 |     public void setTodoTitle(String todoTitle) {
83 |         this.todoTitle = todoTitle;
84 |     }
85 |
86 |
87 |     public char getFinished() {
88 |         return finished;
89 |     }
90 |
91 |     public void setFinished(char finished) {
92 |         this.finished = finished;
93 |     }
94 |
95 |     public Date getCreatedAt() {
96 |         return createdAt;
97 |     }
```

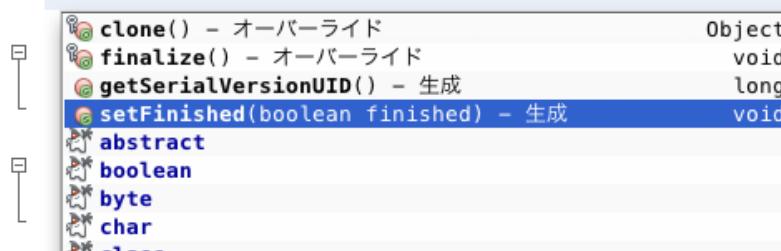
NetBeans のコード補完機能「ctrl+space」でコード補完します。

```
public void setTodoTitle(String todoTitle) {  
    this.todoTitle = todoTitle;  
}
```



```
public void setTodoTitle(String todoTitle) {  
    this.todoTitle = todoTitle;  
}
```

```
public boolean isFinished() {  
    return finished;  
}
```



```
    }

    public boolean isFinished() {
        return finished;
    }

    public void setFinished(boolean finished) {
        this.finished = finished;
    }
```

● @Version アノテーションの追加

JPA の楽観ロックを使用し、バージョンを JPA 側で自動インクリメントさせるために version フィールドに @Version アノテーションを付加します。

```
package todo.domain.model;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
package todo.domain.model;
```

```
import java.io.Serializable;
import java.util.Date;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

@Entity
@Table(name = "TODO")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Todo.findAll", query = "SELECT t FROM Todo t"),
    @NamedQuery(name = "Todo.findByTodoId", query = "SELECT t FROM Todo t WHERE t.todoId = :todoId"),
    @NamedQuery(name = "Todo.findByTodoTitle", query = "SELECT t FROM Todo t WHERE t.todoTitle = :todoTitle"),
    @NamedQuery(name = "Todo.findByFinished", query = "SELECT t FROM Todo t WHERE t.finished = :finished"),
})
```

```
    @NamedQuery(name = "Todo.findByCreatedAt", query = "SELECT t  
FROM Todo t WHERE t.createdAt = :createdAt"),  
    @NamedQuery(name = "Todo.findByVersion", query = "SELECT t FR  
OM Todo t WHERE t.version = :version"))  
public class Todo implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Basic(optional = false)  
    @Column(name = "TODO_ID")  
    private Integer todoId;  
  
    @Basic(optional = false)  
    @NotNull  
    @Size(min = 1, max = 128)  
    @Column(name = "TODO_TITLE")  
    private String todoTitle;  
  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "FINISHED")  
    private boolean finished;  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "CREATED_AT")  
    @Temporal(TemporalType.TIMESTAMP)  
    private Date createdAt;  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "VERSION")  
    @Version  
    private int version;
```

```
public Todo() {  
}  
  
public Todo(Integer todoId) {  
    this.todoId = todoId;  
}  
  
public Todo(Integer todoId, String todoTitle, boolean finished, Date createdAt, int version) {  
    this.todoId = todoId;  
    this.todoTitle = todoTitle;  
    this.finished = finished;  
    this.createdAt = createdAt;  
    this.version = version;  
}  
  
public Integer getTodoId() {  
    return todoId;  
}  
  
public void setTodoId(Integer todoId) {  
    this.todoId = todoId;  
}  
  
public String getTodoTitle() {  
    return todoTitle;  
}  
  
public void setTodoTitle(String todoTitle) {  
    this.todoTitle = todoTitle;  
}
```

```
public boolean isFinished() {
    return finished;
}

public void setFinished(boolean finished) {
    this.finished = finished;
}

public Date getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(Date createdAt) {
    this.createdAt = createdAt;
}

public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

@Override
public int hashCode() {
    int hash = 0;
    hash += (todoId != null ? todoId.hashCode() : 0);
    return hash;
}
```

```
@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the i
d fields are not set
    if (!(object instanceof Todo)) {
        return false;
    }
    Todo other = (Todo) object;
    if ((this.todoId == null && other.todoId != null) || (thi
s.todoId != null && !this.todoId.equals(other.todoId))) {
        return false;
    }
    return true;
}

@Override
public String toString() {
    return "todo.domain.model.Todo[ todoId=" + todoId + " ]";
}

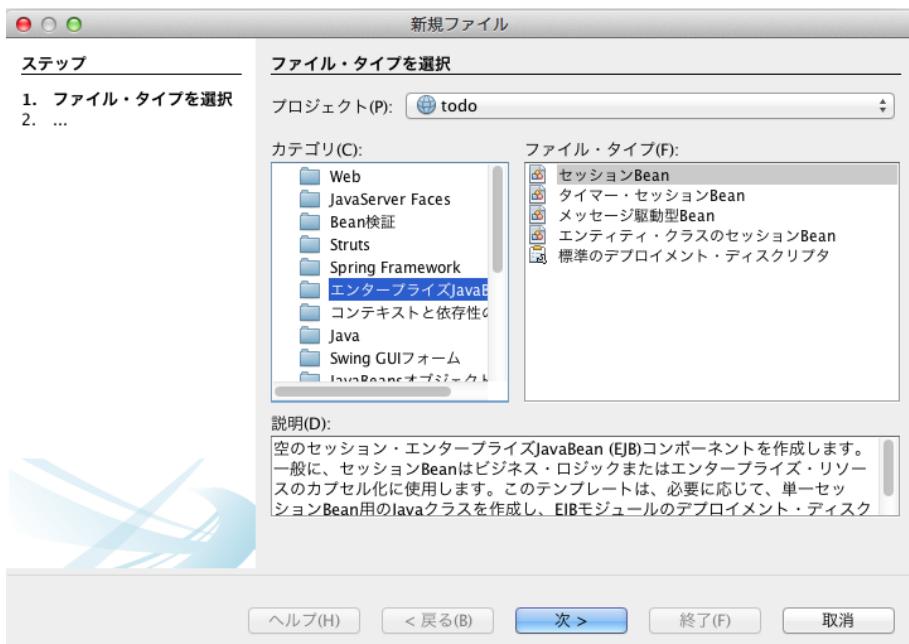
}
```

[4-2] EJB で業務処理を実装

EJB で業務処理を実装します。

■ SessionBean の作成

新規ファイルより「エンタープライズ JavaBean」→「セッション Bean」を選択して「次 >」ボタンを押下します。



「New セッション Bean」のウィンドウで下記を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
EJB名	TodoService
パッケージ	todo.domain.service.todo
セッションのタイプ	ステートレス



自動的に生成されたコードを下記のように修正し Todo の全件取得処理を実装します。

```
package todo.domain.service.todo;
```

```

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import todo.domain.model.Todo;

@Stateless // (1)
public class TodoService {
    // (2)
    @PersistenceContext
    protected EntityManager entityManager;

    public List<Todo> findAll() {
        // (3)
        TypedQuery<Todo> q = entityManager.createNamedQuery("Todo.findAll", Todo.class);
        // (4)
        return q.getResultList();
    }
}

```

項番	説明
(1)	@Stateless アノテーションをつけることでステートレスセッション Bean として認識される。このクラス内での各メソッドはトランザクション管理され、メソッド開始時にトランザクション開始、正常終了時にコミットされる。途中で RuntimeException が発生した場合はトランザクションがロールバックされる。
(2)	@PersistenceContext アノテーションで JPA のエンティティを操作する EntityManager をインジェクションする。

(3)	Todo エンティティを全件取得する NamedQuery を作成する。指定した "Todo.findAll" に対応する Query は Todo クラスに @NamedQuery(name = "Todo.findAll", query = "SELECT t FROM Todo t") と定義されている。TypedQuery 型として変数宣言することでタイプセーフになる。
(4)	Query の結果をリストとして取得する。

■ JUnit の作成

動作確認を兼ねて予め JUnit のテストケースを作成します。

● テストケースの作成

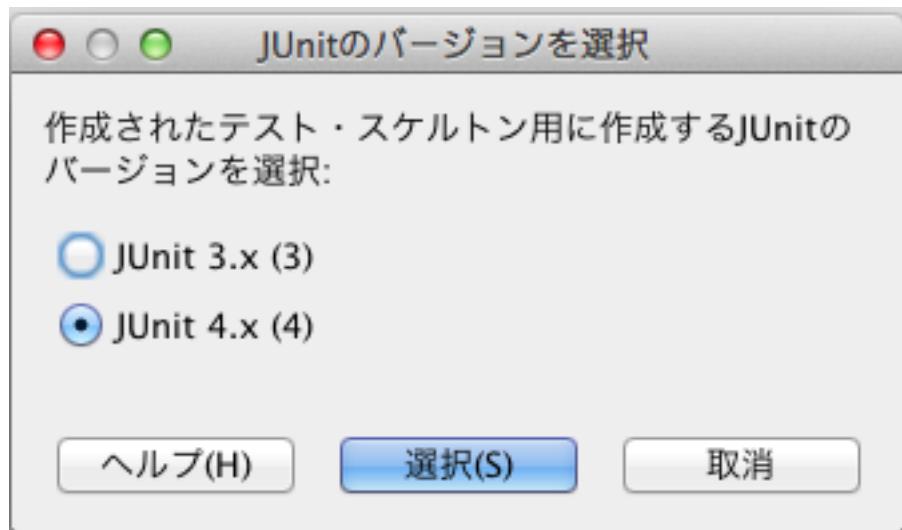
TodoService を右クリックし、「ツール」→「テストを作成」をクリックします。



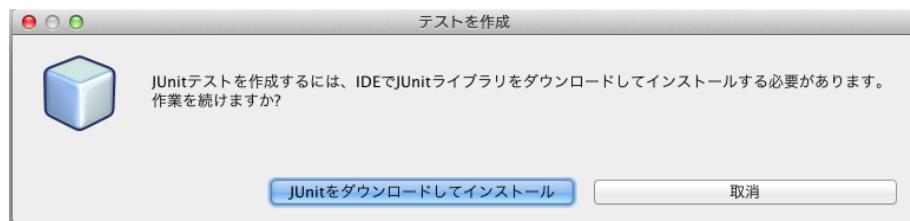
「OK」をクリック。



「JUnit 4.x(4)」を選択し「選択(S)」ボタンを押下します。



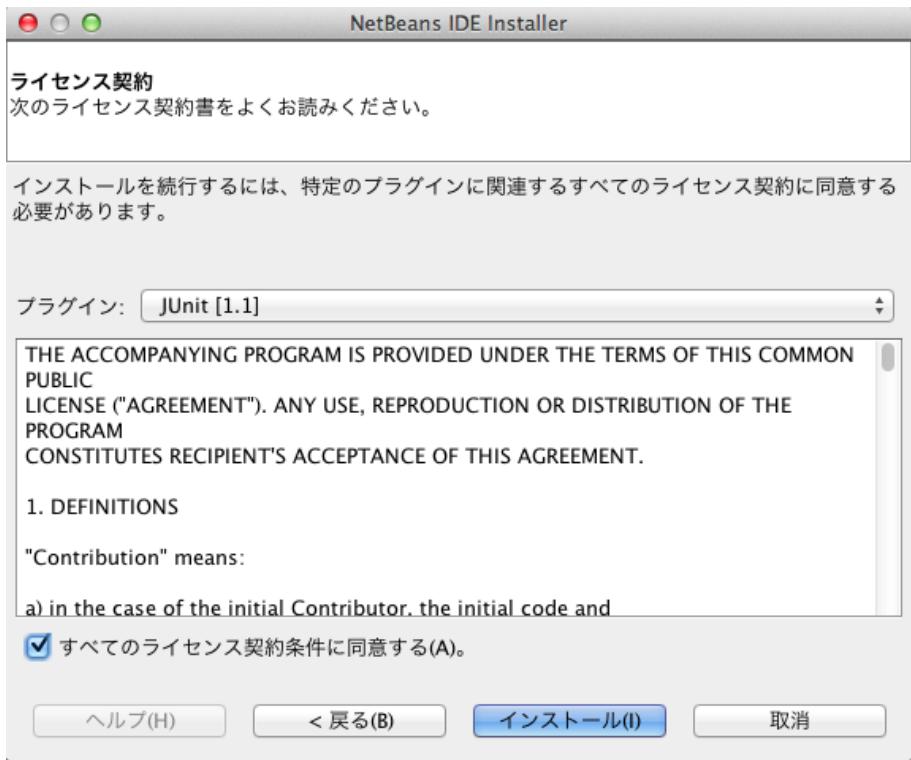
(初回のみ)「JUnit をダウンロードしてインストール」ボタンを押下します。

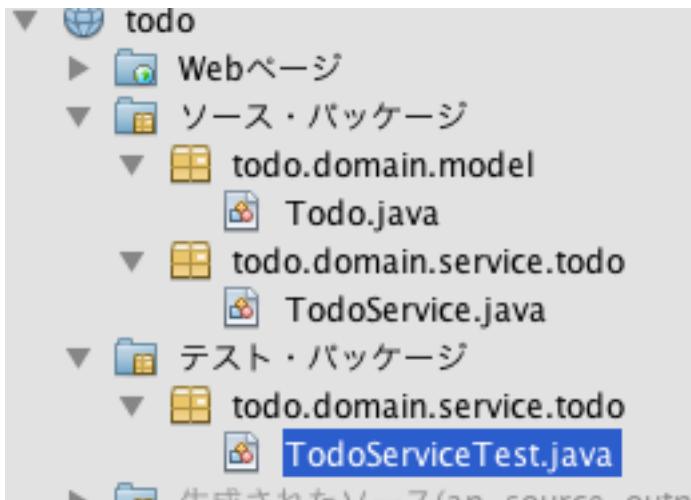


「次 >」ボタンを押下します。



「すべてのライセンス契約条件に同意する(A)。」にチェックし、「インストール(I)」ボタンを押下します。





自動生成されたテストコードを下記のように修正してください。

```
package todo.domain.service.todo;

import java.util.List;
import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
import todo.domain.model.Todo;

public class TodoServiceTest {
    // (1)
    private EJBContainer container;
```

```
private Context context;

public TodoServiceTest() {
}

@BeforeClass
public static void setUpClass() {
}

@AfterClass
public static void tearDownClass() {
}

@Before
public void setUp() {
    // (2)
    container = javax.ejb.embeddable.EJBContainer.createEJBContainer();
    context = container.getContext();
}

@After
public void tearDown() {
    // (3)
    container.close();
}

/**
 * Test of findAll method, of class TodoService.
 */
@Test
```

```

public void testFindAll() throws Exception {
    System.out.println("findAll");
    // (4)
    TodoService instance = (TodoService)context.lookup("java:global/classes/TodoService");
    List<Todo> result = instance.findAll();
    System.out.println(result);
    assertNotNull(result);
}
}

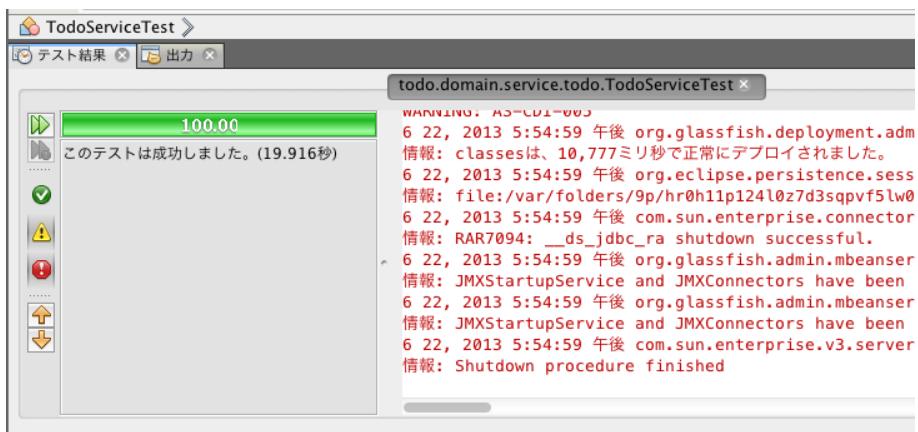
```

項番	説明
(1)	テスト用に組み込みコンテナを使用する。
(2)	前処理で組み込みコンテナを生成し、JNDI コンテキストを取得する。
(3)	後処理でコンテナを破棄する。
(4)	JNDI で EJB を取得する。JNDI 名は”java:global/classes/EJB 名”で取得できる。

テストコードのソースコード内で右クリックし、「ファイルをテスト」を選択してテストを実行します。



テストの実行結果は下記のように表示されます。



■ 業務処理の実装

TodoService に対して追加の業務処理を実装します。ここで実装する処理を下記に示します。

- Todo の全件取得
- Todo の新規作成
- Todo の完了
- Todo の削除

実装する処理と対応するメソッドを下記に示します。

- Todo の全件取得→findAll メソッド
- Todo の新規作成→create メソッド
- Todo の完了→finish メソッド
- Todo の削除→delete メソッド

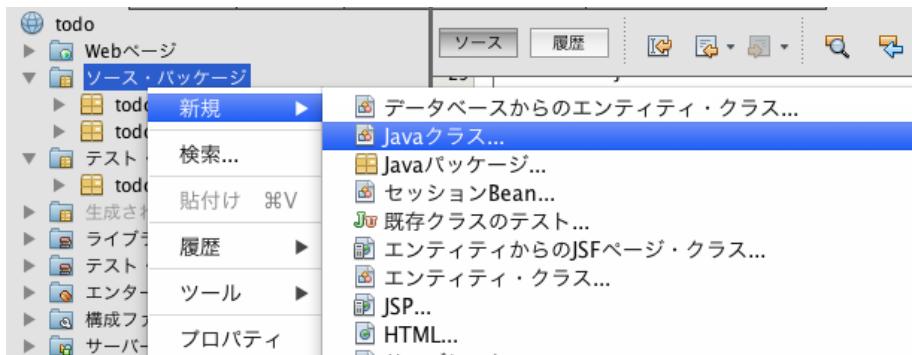
findAll は既に実装しているため、ここでは残りの 3 処理を実装します。

● 例外の作成

業務処理を実装する前に、業務処理で使用する例外クラスを実装します。実装する例外クラスは下記の 2 つです。

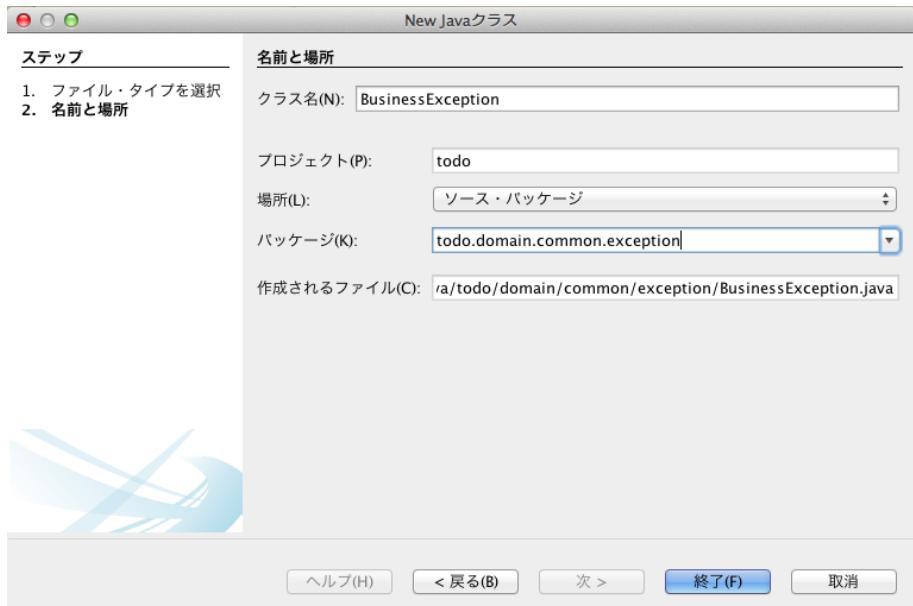
- BusinessException・・・業務エラーが発生した際にスローする例外
- ResourceNotFoundException・・・対象のリソース(エンティティ)が存在しない場合にスローする例外

例外ハンドリングやレスポンスステータスを変更できるようにこの 2 種類の例外は別に実装します。



「New Java クラス」のウィザード画面で下記の内容を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
クラス名	BusinessException
パッケージ	todo.domain.common.exception



```
package todo.domain.common.exception;

import javax.ejb.ApplicationException;

@ApplicationException // (1)
public class BusinessException extends RuntimeException {
```

```

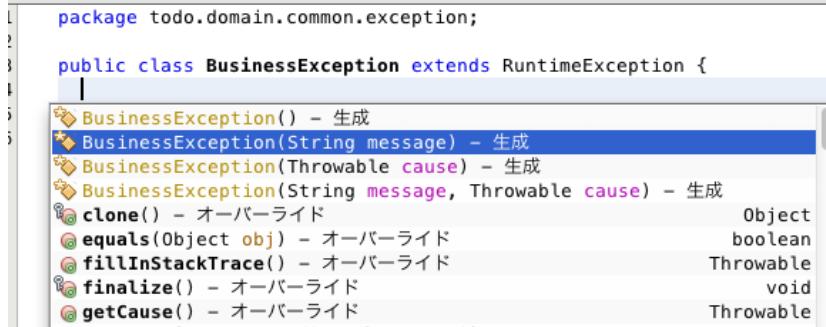
public BusinessException(String message) {
    super(message);
}

}

```

項目番号	説明
(1)	EJB からスローする例外クラスには @ApplicationException アノテーションをつける。このアノテーションがない場合は javax.ejb.EJBException にラップしてスローされる。

Memo: “extends RuntimeException”を書いた後は Ctrl+Space でコンストラクタを自動生成できる。



```

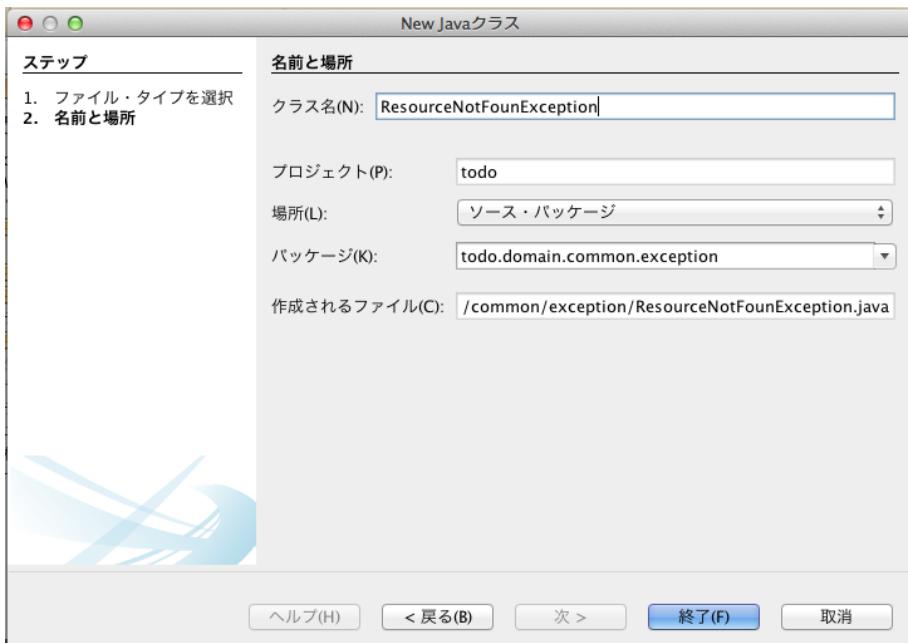
package todo.domain.common.exception;

public class BusinessException extends RuntimeException {
    /**
     * BusinessException() - 生成
     * BusinessException(String message) - 生成
     * BusinessException(Throwable cause) - 生成
     * BusinessException(String message, Throwable cause) - 生成
     * clone() - オーバーライド
     * equals(Object obj) - オーバーライド
     * fillInStackTrace() - オーバーライド
     * finalize() - オーバーライド
     * getCause() - オーバーライド
    */
}

```

同様に `ResourceNotFoundException` を作成します。下記の項目を入力して「終了(F)」ボタンを押下します。

入力項目	入力値
クラス名	<code>ResourceNotFoundException</code>
パッケージ	<code>todo.domain.common.exception</code>



```
package todo.domain.common.exception;

import javax.ejb.ApplicationException;

@ApplicationException
```

```
public class ResourceNotFoundException extends RuntimeException{  
  
    public ResourceNotFoundException(String message) {  
        super(message);  
    }  
  
}
```

● 業務処理メソッドの実装

TodoService の EJB に対して業務処理メソッドを追加します。

```
package todo.domain.service.todo;

import java.util.Date;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import todo.domain.common.exception.BusinessException;
import todo.domain.common.exception.ResourceNotFoundException;
import todo.domain.model.Todo;

@Stateless
public class TodoService {

    private static final long MAX_UNFINISHED_COUNT = 5;
    @PersistenceContext
    protected EntityManager entityManager;

    public List<Todo> findAll() {
        TypedQuery<Todo> q = entityManager.createNamedQuery("Todo.findAll", Todo.class);
        return q.getResultList();
    }

    public Todo findOne(Integer todoId) {
        // (1)
    }
}
```

```

        Todo todo = entityManager.find(Todo.class, todoId);

        if (todo == null) {
            // (2)
            throw new ResourceNotFoundException("[E404] The requested Todo is not found. (id=" + todoId + ")");
        }

        return todo;
    }

    public Todo create(Todo todo) {
        // (3)
        TypedQuery<Long> q = entityManager.createQuery("SELECT COUNT(x) FROM Todo x WHERE x.finished = :finished", Long.class)
            .setParameter("finished", false);

        long unfinishedCount = q.getSingleResult();
        if (unfinishedCount > MAX_UNFINISHED_COUNT) {
            // (4)
            throw new BusinessException("[E001] The count of unfinished Todo must not be over "
                + MAX_UNFINISHED_COUNT + ".");
        }

        todo.setFinished(false);
        todo.setCreatedAt(new Date());
        // (5)
        entityManager.persist(todo);
        return todo;
    }

    public Todo finish(Integer todoId) {
        Todo todo = findOne(todoId);

```

```

        if (todo.isFinished()) {
            throw new BusinessException("[E002] The requested Todo
is already finished. (id="
                + todoId + ")");
        }
        todo.setFinished(true);
        // (6)
        entityManager.merge(todo);
        return todo;
    }

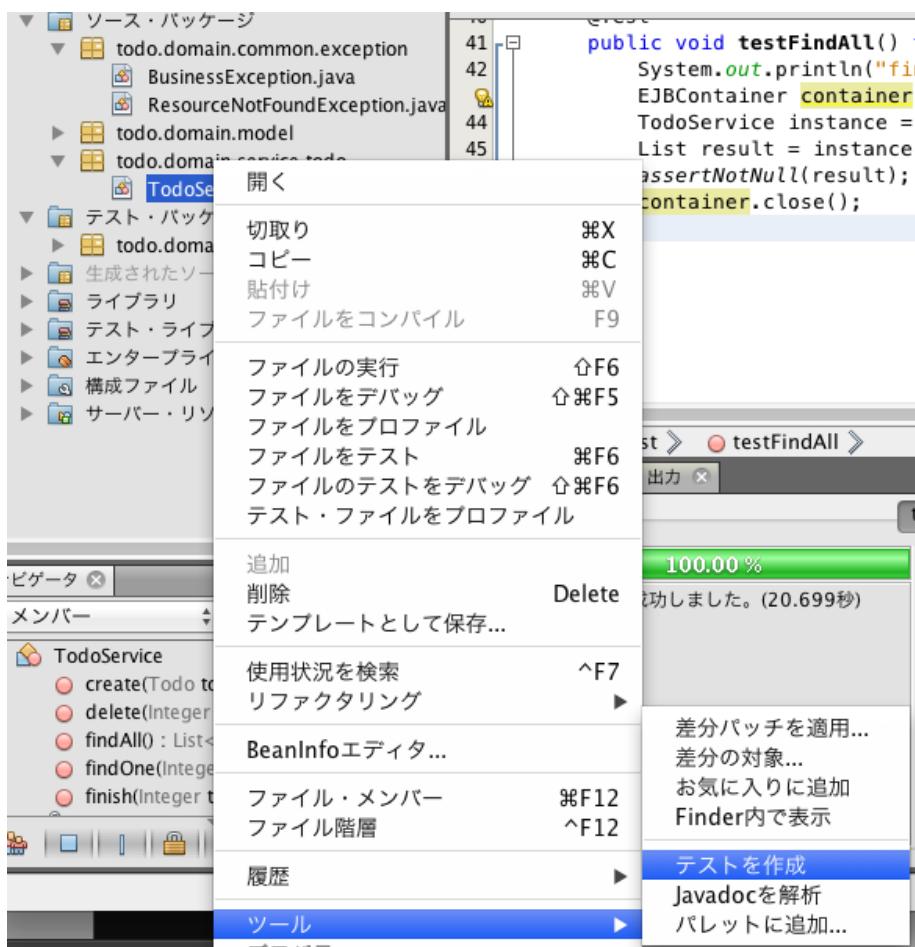
    public void delete(Integer todoId) {
        Todo todo = findOne(todoId);
        // (7)
        entityManager.remove(todo);
    }
}

```

項目番号	説明
(1)	find メソッドにエンティティの主キーを指定することでエンティティの検索を行うことができる。JPQL や SQL を書く必要はない。
(2)	find でエンティティが見つからない場合は null が返る。ここで ResourceNotFoundException をスローする。
(3)	検索用の JPQL を渡し、Query を作成する。パラメータは JPQL 内に":パラメータ名"を記述することで置換できる。 ここでは createQuery メソッドで動的に JPQL を実行する例として実装しているが、毎回 JPQL をコンパイルするオーバーヘッドがあるので、findAll 同様に NamedQuery として定義した方が高性能である。 検索処理においては、リクエストによってクエリが変化する場合を除

	き、NamedQuery を使用した方が良い。
(4)	業務ルールを満たさない場合は BusinessException をスローする。
(5)	<p>persist メソッドにエンティティを渡すことで、エンティティを EntityManager の管理下に置く。これにより、メソッド終了時のトランザクションコミットのタイミングで DB に insert 文が実行される。ID が設定される。insert 用の SQL を各必要がない。</p> <p>明示的に DB に反映させる場合は persist 後に flush メソッドを実行する必要があるが、ここでは必要がないので実行しない。</p>
(6)	<p>(5)の場合と違い、対象のエンティティに既に ID が設定されているため、merge メソッドで EntityManager の管理下に置き、トランザクションコミット時に update 文が実行される。</p> <p>(実際は find をした段階で EntityManager の管理下になるので merge は実行しなくても更新される)</p>
(7)	remove メソッドにエンティティを渡すことで、エンティティを EntityManager の管理から除外する。これにより、メソッド終了時のトランザクションコミットのタイミングで DB に delete 文が実行される。

テストケースを追加します。





testFindOne, testCreate, testFinished, testDelete が追加されます。ここではテストメソッドの修正に対する詳細は省略します。

Memo:

JUnit 内で明示的にトランザクション管理したい場合は以下のように UserTransaction オブジェクトを取得して、コミット、ロールバックを行う。

```
UserTransaction ut = (UserTransaction) context.lookup("java:comp  
/UserTransaction");
```

トランザクション開始

```
ut.begin();
```

トランザクションコミット

```
ut.commit();
```

トランザクションロールバック

```
ut.rollback();
```

必ずロールバックさせたい場合は begin 後、

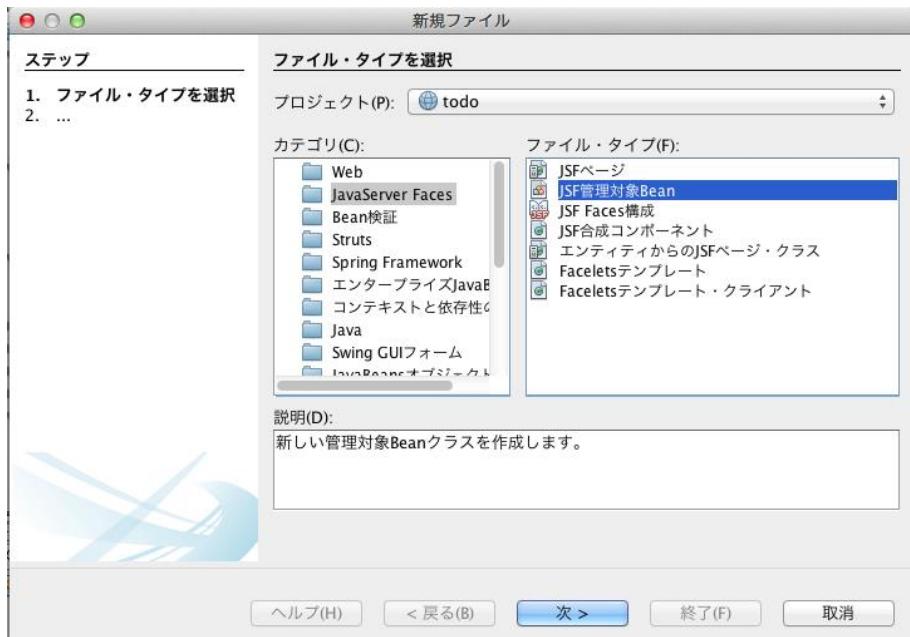
```
ut.setRollbackOnly();
```

を実行すれば良い。

[4-3] JSF で画面を作成

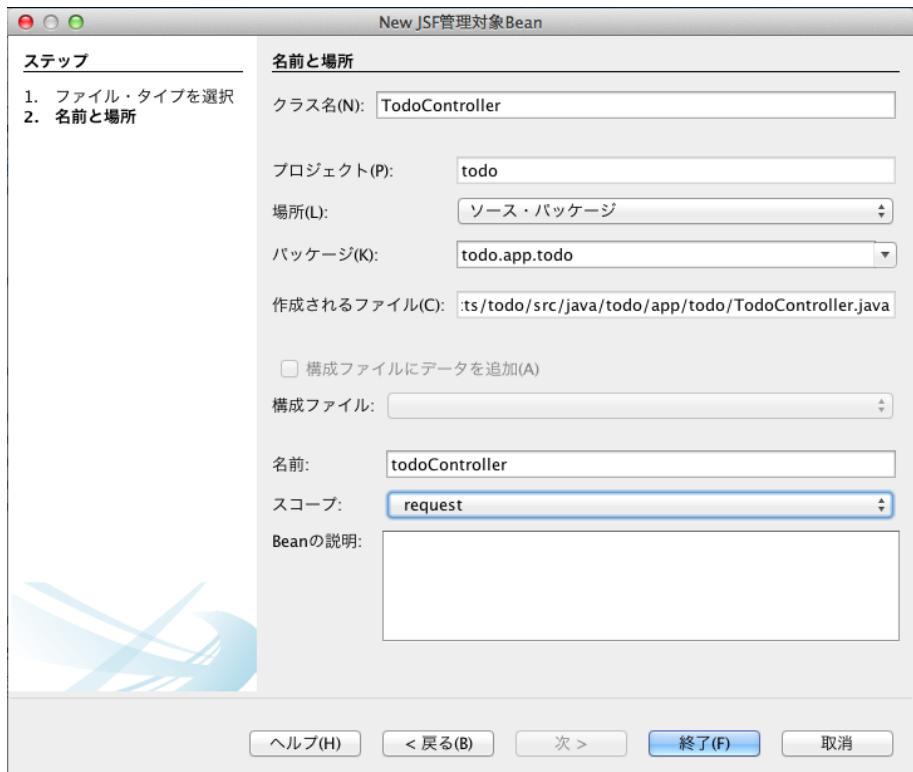
■ ManagedBean の作成

Todo 管理の画面遷移を制御する TodoController を作成します。新規ファイルから「JavaServer Faces」→「JSF 管理対象 Bean」を選択し「次 >」ボタンを押下します。



「New JSF 管理対象 Bean」のウィザード画面で、下記の項目を入力し「週力(F)」ボタンを押下します。

入力項目	入力値
クラス名	TodoController
パッケージ名	todo.app.todo
スコープ	Request



自動生成されたコードは下記のようになります。

```
package todo.app.todo;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

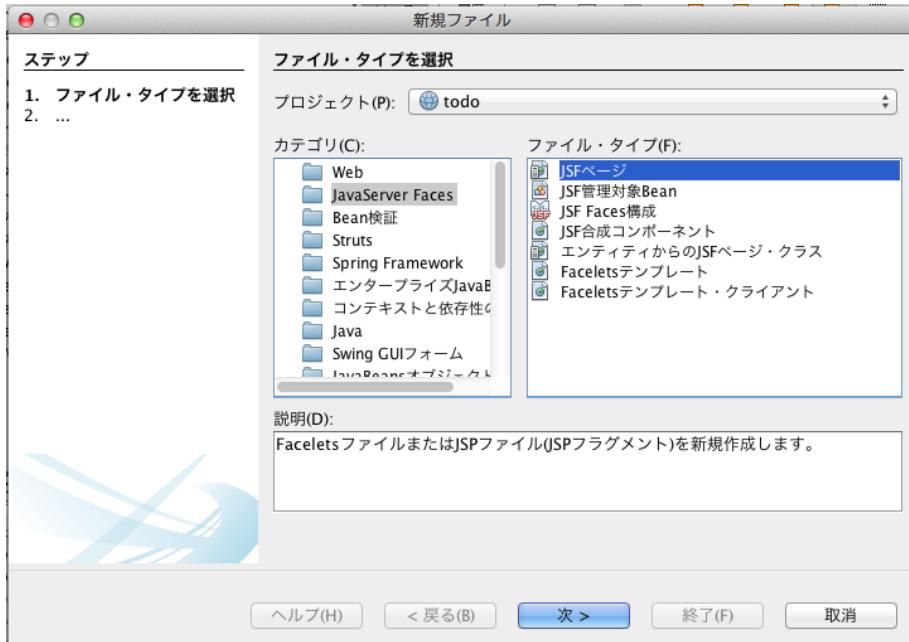
@Named(value = "todoController") // (1)
@RequestScoped // (2)
public class TodoController {

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }
}
```

項目番	説明
(1)	@Named アノテーションを付けることで、CDI の機能によりこのクラスが JSF の管理対象 Bean(ManagedBean)であると認識される。value に渡した値で次に説明する facelets から#{名前}形式のアクセスが出来るようになる。
(2)	ManagedBean のスコープを指定する。ここではリクエストスコープを指定する。毎リクエストでこのオブジェクトが作成されることに鳴る。 他には ApplicationScoped や SessionScopde 等がある。

■ Facelets の作成

View として Facelets(list.xhtml) を作成します。新規ファイルより「JavaServer Faces」→「JSF ページ」を選択し「次 >」ボタンを押下します。



「New JSF ページ」のウィザード画面に下記の項目を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
ファイル名	list
フォルダ	todo
オプション	facelets

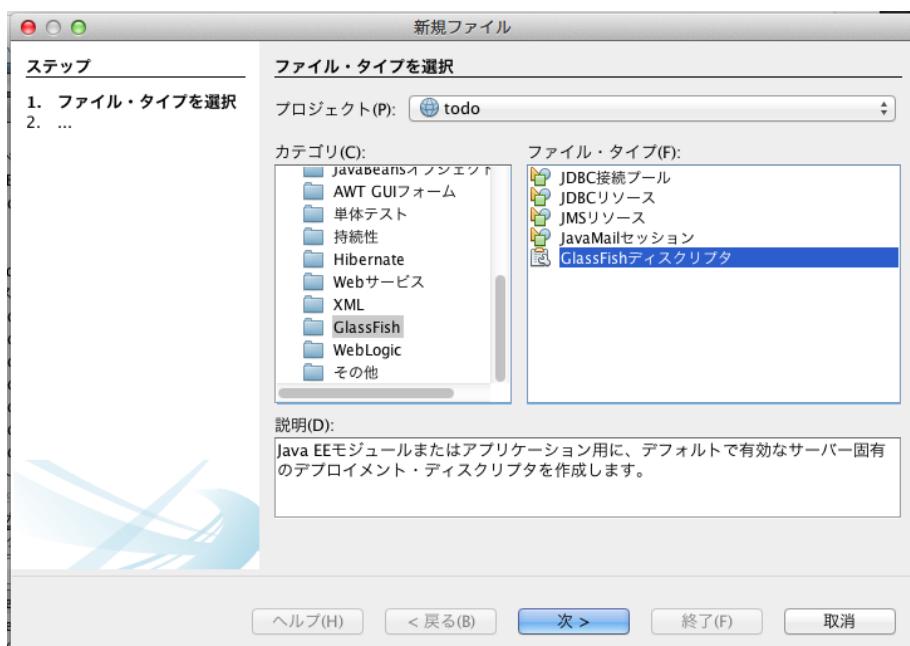


Web ページフォルダに todo/list.xhtml が生成されます。生成されたファイルは後ほど修正します。

■ 日本語の文字化けに対応する

facelets を実装する前に Glassfish の設定を行います。Glassfish のエンコーディングはデフォルトで日本語に対応していないため、リクエスト・レスポンス中の日本語が文字化けします。そこで、エンコーディングを明示的に UTF-8 に設定変更します。

新規ファイルで「Glassfish」→「Glassfish ディスクリプタ」を選択して「次>」ボタンを押下します。



「終了(F)」ボタンを押下します。



glassfish-web.xml に対して下記の 1 行を追加してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD GlassF
ish Application Server 3.1 Servlet 3.0//EN" "http://glassfish.org
/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app error-url="">
    <class-loader delegate="true"/>
    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Keep a copy of the generated servlet class' jav
a code.</description>
        </property>
    </jsp-config>
    <parameter-encoding default-charset="UTF-8" />
</glassfish-web-app>
```

■ Todo 全件表示

それでは画面の実装を行っていきます。まずは TODO の全件表示を行うよう
に修正します。

● TodoController の修正

```
package todo.app.todo;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.inject.Named;
```

```
import javax.enterprise.context.RequestScoped;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Named(value = "todoController")
@RequestScoped
public class TodoController {

    @EJB // (1)
    protected TodoService todoService;

    // (2)
    protected List<Todo> todoList;

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }

    // (3)
    public List<Todo> getTodoList() {
        return todoList;
    }

    @PostConstruct // (4)
    public void init() {
        todoList = todoService.findAll();
    }
}
```

項目番号	説明
(1)	@EJB アノテーションで EJB をインジェクションする。
(2)	画面に表示するための Todo リストを定義する。
(3)	getter を定義して facelets から#{todoController.todoList}としてアクセスできる。
(4)	@PostConstruct アノテーションをつけることでこのクラスが生成されたあとに実行される初期処理を定義できる。コンストラクタとは違い、インジェクションされたフィールドを使用できる。 @RequestScope の場合、毎リクエストで実行されることになる。

● list.xhtml の修正

先ほど作成した list.xhtml を以下のように修正してください。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
  <h:head>
    <title>Todo List</title>
    <style type="text/css">
      .strike {
        text-decoration: line-through;
      }
    </style>
  </h:head>
  <h:body>

    <h2>Todo List</h2>
```

```

<!-- (1) -->
<h:dataTable value="#{todoController.todoList}" var="todo">
<!-- (2) -->
<h:column>
<!-- (3) -->
<f:facet name="header">
<h:outputText value="Titile" />
</f:facet>
<!-- (4) -->
<h:outputText value="#{todo.todoTitle}" rendered="#{!todo.finished}" />
<h:outputText value="#{todo.todoTitle}" rendered="#{todo.finished}" class="strike" />
</h:column>
<h:column>
<f:facet name="header">
<h:outputText value="Created At" />
</f:facet>
<h:outputText value="#{todo.createdAt}" />
</h:column>

</h:dataTable>
</h:body>
</html>

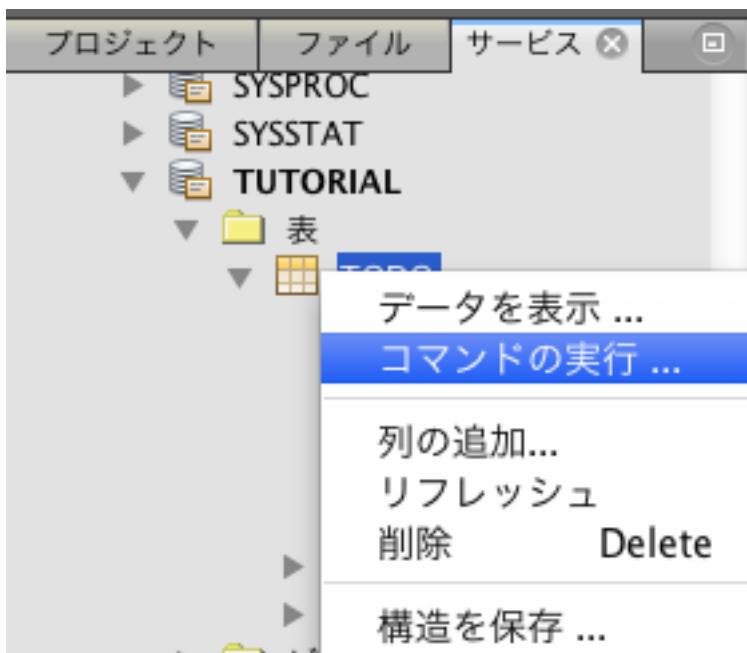
```

項目番	説明
(1)	<h:dataTable>タグでデータ一覧用のテーブルを作成する。value 属性に対象のリストオブジェクトを指定する。var 属性で各データの名前を指定する。
(2)	<h:column>タグでデータの列を作成する。

(3)	<f:facet>タグで列のヘッダーを作成する。
(4)	<h:outputText> タグで文字列(Todo のタイトル)を出力する。 rendered 属性で出力するための条件を指定できる。ここでは todo が 完了しているかどうかでスタイルを帰る必要があるため rendered に finished の条件を渡している。finished が true の場合は css に strike クラスが指定されるようにする。

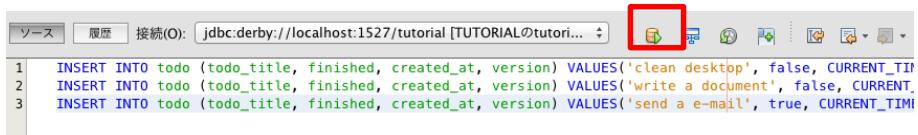
● テストデータの投入

一覧表示を確認するために DB にテストデータを挿入します。サービスウィンドウを開き、Tutorial スキーマを開き、「表」→「TODO」を右クリックして「コマンドの実行」をクリックします。



下記の INSERT 文を貼付けて「SQL の実行」ボタンを押下します。

```
INSERT INTO todo (todo_title, finished, created_at, version) VALUES('clean desktop', false, CURRENT_TIMESTAMP, 1);
INSERT INTO todo (todo_title, finished, created_at, version) VALUES('write a document', false, CURRENT_TIMESTAMP, 1);
INSERT INTO todo (todo_title, finished, created_at, version) VALUES('send a e-mail', true, CURRENT_TIMESTAMP, 1);
```



挿入したデータを確認するために、「TODO」をクリックして「データの表示 ...」をクリックします。



実行すると下記のようにデータの一覧を表示します。

ソース 戻り 接続(O): jdbc:derby://localhost:1527/tutorial [TUTORIALのtutori...]

```

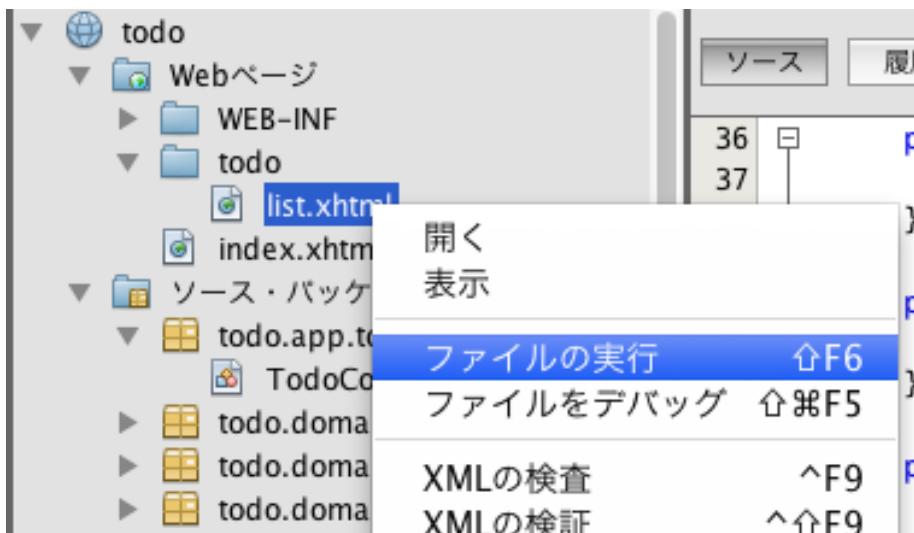
1 select * from TUTORIAL.TODO;
2

```

select * from TUTORIAL.TODO

#	TITLE	FINISHED	CREATED_AT	VERSION
1	... clean desktop	<input type="checkbox"/>	2013-06-23 00:09:42.543	1
2	... write a document	<input type="checkbox"/>	2013-06-23 00:09:42.555	1
3	... send a e-mail	<input checked="" type="checkbox"/>	2013-06-23 00:09:42.564	1

データを挿入後、アプリケーションを実行して全件表示画面を出力します。
list.xhtml を右クリックして「ファイルの実行」をクリックしてください。



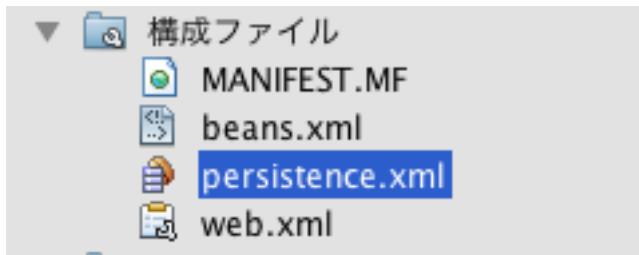
ブラウザが自動的に起動し、以下のようにテーブルが表示されていることを確認します。

Title	Created At
clean desktop	Sun Jun 23 00:47:56 JST 2013
write a document	Sun Jun 23 00:47:56 JST 2013
send a e-mail	Sun Jun 23 00:47:56 JST 2013

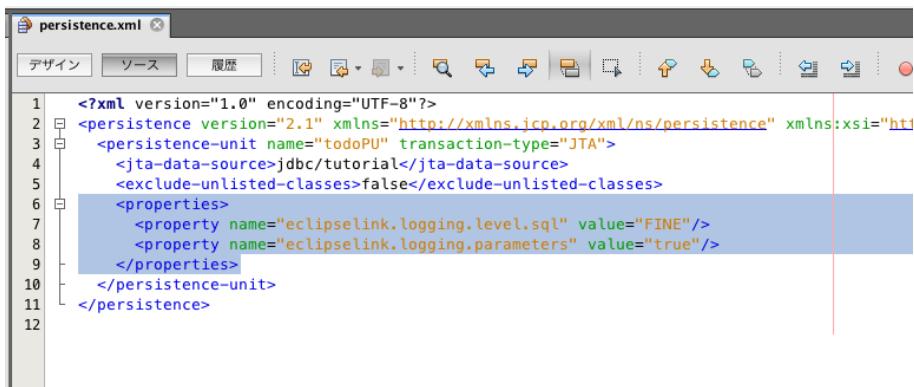
● SQL ログを出力する

開発時にデバッグ、チューニング等の目的のため、JPA で発行されている実際の SQL の内容を確認することはとても有用です。そこで SQL ログ出力設定を行い SQL の内容を確認できるように設定変更します。

persistence.xml を開いてプロパティを変更してください。



「ソース」ボタンを押下してください。



The screenshot shows the Eclipse IDE interface with the title bar "persistence.xml". Below the title bar is a toolbar with various icons. The main area displays the XML code for persistence.xml. The code is as follows:

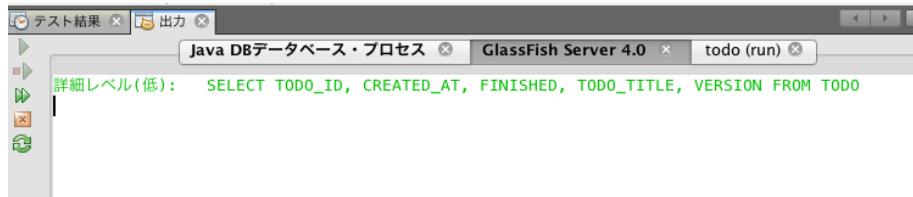
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd">
  <persistence-unit name="todoPU" transaction-type="JTA">
    <jta-data-source>jdbc/tutorial</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.logging.level.sql" value="FINE"/>
      <property name="eclipselink.logging.parameters" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

設定ファイルに対して、下記の修正をおこないます。

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence_2_1.xsd">
  <persistence-unit name="todoPU" transaction-type="JTA">
    <jta-data-source>jdbc/tutorial</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.logging.level.sql" value="FINE"/>
    </>
      <property name="eclipselink.logging.parameters" value="true"/>
    </>
    </properties>
  </persistence-unit>
```

```
</persistence>
```

設定終了後、EntityManager の操作に対して、以下のように SQL ログが 출력されます。



■ Todo 新規作成

次に、「新規作成フォーム」を作成します。

● TodoController の修正

TodoController に対して下記を追加してください。

```
package todo.app.todo;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Named(value = "todoController")
@RequestScoped
public class TodoController {
```

```
@EJB  
protected TodoService todoService;  
  
// (1)  
protected Todo todo = new Todo();  
protected List<Todo> todoList;  
  
/**  
 * Creates a new instance of TodoController  
 */  
public TodoController() {  
}  
  
public Todo getTodo() {  
    return todo;  
}  
  
public List<Todo> getTodoList() {  
    return todoList;  
}  
  
@PostConstruct  
public void init() {  
    todoList = todoService.findAll();  
}  
  
// (2)  
public String create() {  
    todoService.create(todo);  
    // (3)  
    return "list?faces-redirect=true";  
}
```

```
    }  
}
```

項目番	説明
(1)	新規作成フォームに対応する Bean を定義する。
(2)	新規作成用のボタンが押された際に実行する処理を定義する。フォームに入力された値を格納する todo オブジェクトを EJB に渡す。
(3)	正常終了後、list.xhtml にリダイレクトする。faces-redirect=true をつけることにより、リダイレクトすることができる。

● list.xhtml の修正

次に list.xhtml に対して修正を加えます。

```
<?xml version='1.0' encoding='UTF-8' ?>  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "h  
ttp://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://xmlns.jcp.org/jsf/html"  
      xmlns:f="http://xmlns.jcp.org/jsf/core">  
  
<h:head>  
    <title>Todo List</title>  
    <style type="text/css">  
        .strike {  
            text-decoration: line-through;  
        }  
    </style>  
</h:head>  
  
<h:body>  
    <h2>Create Todo</h2>  
    <!-- (1) -->  
    <h:form>
```

```

<h:panelGrid columns="2">
    <h:outputLabel value="Title: " />
    <!-- (2) -->
    <h:inputText value="#{todoController.todo.todoTitle}" />
    <!-- (3) -->
    <h:commandButton value="Create" action="#{todoController.create}" />
</h:panelGrid>
</h:form>

<h2>Todo List</h2>

<h:dataTable value="#{todoController.todoList}" var="todo">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Title" />
        </f:facet>
        <h:outputText value="#{todo.todoTitle}" rendered="#{!todo.finished}" />
        <h:outputText value="#{todo.todoTitle}" rendered="#{todo.finished}" class="strike" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>
        <h:outputText value="#{todo.createdAt}" />
    </h:column>

```

```
</h:dataTable>  
</h:body>  
</html>
```

項番	説明
(1)	<h:form>タグでフォームを作成する。
(2)	<h:inputText>タグでテキストフィールドを作成する。value 属性に 対応する ManagedBean のフィールド名を指定する。この例では TodoController の todo フィールド野中の todoTitle フィールドと対 応していることを意味する。
(3)	<h:commandButton>タグで実行ボタンを作成する。action 属性にボ タンを押した際に実行されるメソッド名を指定する。

list.xhtml を再表示し、「Title」に TODO の内容を入力して「Create」ボタンを押下してください。Todo List に入力内容が追加されます。

The screenshot shows a web browser window titled "Todo List". The address bar indicates the URL is "localhost:8080/todo/faces/todo/list.xhtml". The main content area displays a "Create Todo" form with a "Title" input field containing "read a book" and a "Create" button. Below this, a "Todo List" section is shown with three items:

Title	Created At
clean desktop	Sun Jun 23 00:47:56 JST 2013
write a document	Sun Jun 23 00:47:56 JST 2013
send a e-mail	Sun Jun 23 00:47:56 JST 2013

The screenshot shows a web application window titled "Todo List". At the top, there is a header bar with standard OS X-style icons (red, yellow, green) and a title "Todo List". Below the header, the URL "localhost:8080/todo/faces/todo/list.xhtml" is displayed. The main content area has two sections: "Create Todo" and "Todo List". The "Create Todo" section contains a "Title:" label and a text input field, followed by a "Create" button. The "Todo List" section displays a table with two columns: "Title" and "Created At". The table data is as follows:

Title	Created At
clean desktop	Sun Jun 23 00:47:56 JST 2013
write a document	Sun Jun 23 00:47:56 JST 2013
send a e-mail	Sun Jun 23 00:47:56 JST 2013
read a book	Sun Jun 23 01:25:42 JST 2013

● 入力エラーメッセージの表示

次にフォームの入力内容を検証し、エラー・メッセージを出力するように、コードを修正します。入力内容の検証は Bean Validation にて実装します。Bean Validation の入力検証ルール設定は、JPA のエンティティを自動生成した際に、一緒に付加されています。

例えば todoTitle のフィールドに対して「null の不許可、1 文字以上、128 文字以下」というルールを Bean Validation で以下のように、@NotNull アノテーションと @Size アノテーションで実現しています。

```
@NotNull  
@Size(min = 1, max = 128)  
@Column(name = "TODO_TITLE")  
private String todoTitle;
```

facelets の form 内で参照している Bean(CDI)に対して、Bean Validation のルールが指定されている場合、action を実行する段階で入力値の検証が自動的に行われます。検証に失敗した際に表示するエラー・メッセージは、エラー・メッセージ表示用のタグを追加し表示されます。

list.xhtml を以下のように修正する。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
        <title>Todo List</title>
        <style type="text/css">
            .strike {
                text-decoration: line-through;
            }
            .alert {
                border: 1px solid;
                padding: 3px;
                width: 80%;
            }
            .alert-error {
                background-color: #c60f13;
                border-color: #970b0e;
                color: white;
            }
        </style>
    </h:head>
    <h:body>
```

```

<!-- (1) -->
<h:messages layout="table" styleClass="alert alert-error"
/>

<h2>Create Todo</h2>
<h:form>
    <h:panelGrid columns="2">
        <h:outputLabel value="Title: " />
        <h:inputText value="#{todoController.todo.todoTitle}" />
        <h:commandButton value="Create" action="#{todoController.create}" />
    </h:panelGrid>
</h:form>

<h2>Todo List</h2>

<h:dataTable value="#{todoController.todoList}" var="todo"
">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Title" />
        </f:facet>
        <h:outputText value="#{todo.todoTitle}" rendered="#{!todo.finished}" />
        <h:outputText value="#{todo.todoTitle}" rendered="#{todo.finished}" class="strike" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>

```

```

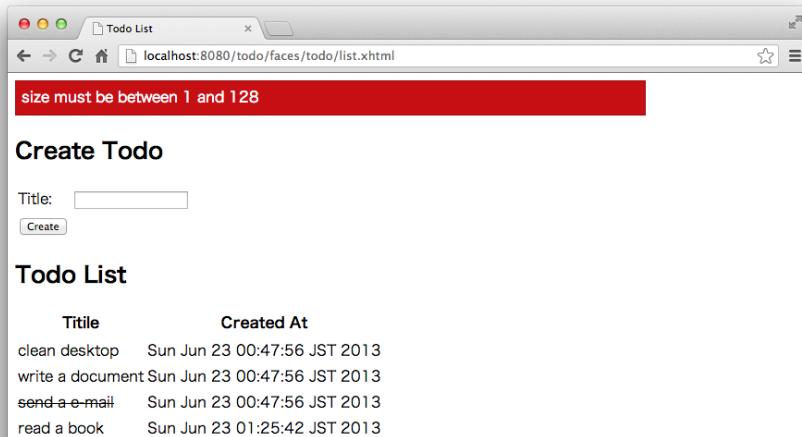
        <h:outputText value="#{todo.createdAt}" />
    </h:column>

</h:dataTable>
</h:body>
</html>

```

項目番	説明
(1)	<h:messages>タグでメッセージを表示する。layout 属性で table を指定しているため、テーブル形式で出力される。そのテーブルに指定する CSS クラスを styleCss 属性で指定する。

例えば、「Title」の入力項目に対し空のまま「Create」ボタンを押下すると
以下のようにエラーメッセージが表示されます。



Memo:テキストフィールドの横にエラーメッセージを出力したい場合は、以下のよう

に変更すればよい。

```
<h:form>
    <h:panelGrid columns="3">
        <h:outputLabel value="Title: " />
        <h:inputText id="title" value="#{todoController.todo.todoTitle}" />
    </h:panelGrid>
    <h:message for="title" errorClass="text-error" />
    <h:commandButton value="Create" action="#{todoController.create}" />
</h:form>
```

● 業務例外をハンドリングする

入力検証時のエラー・メッセージ表示と同様、EJB で BusinessException が発生した場合に、エラーメッセージを表示するように修正します。

```
package todo.app.todo;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import todo.domain.common.exception.BusinessException;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;
```

```
@Named(value = "todoController")
@RequestScoped
public class TodoController {

    @EJB
    protected TodoService todoService;
    protected Todo todo = new Todo();
    protected List<Todo> todoList;

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }

    public Todo getTodo() {
        return todo;
    }

    public List<Todo> getTodoList() {
        return todoList;
    }

    @PostConstruct
    public void init() {
        todoList = todoService.findAll();
    }

    public String create() {
        try {
            todoService.create(todo);
        }
    }
}
```

```

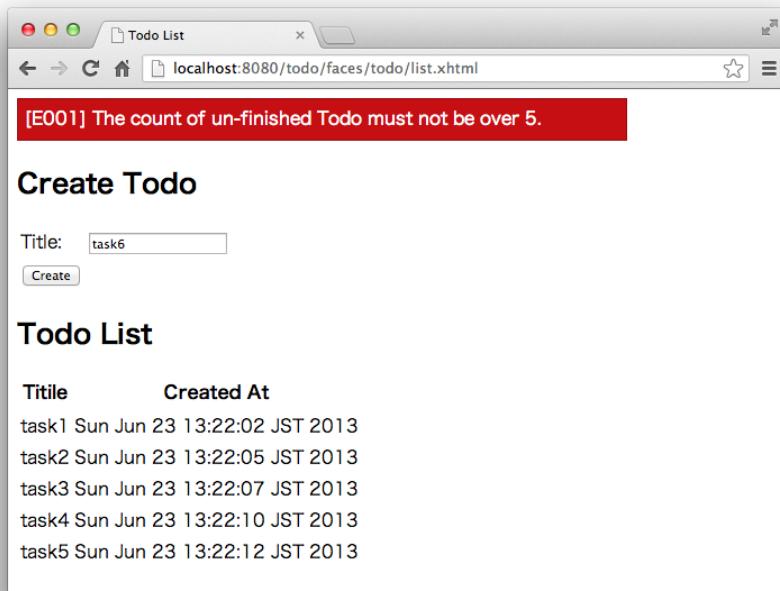
} catch (BusinessException e) {
    // (1)
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage
e.SEVERITY_ERROR, e.getMessage(), null));
    return "list.xhtml";
}
return "list.xhtml?faces-redirect=true";
}
}

```

項目番号	説明
(1)	BusinessException をキャッチして FacesContext に FacesMessage を追加する。SEVERITY_ERROR を指定する。

入力検証のエラー・メッセージ表示用に設定した<h:messages>タグは FacesContext に追加した FacesMessage も表示するため、xhtml はそのままでも良い。

上記の実装を行うと「Todo」を 6 件追加とすると次のようなエラーメッセージが表示されるようになります。



Memo: JSF は ExceptionHandler の仕組みがあるが、MangedBean 単位でハンドリングできない(?)ため、ここでは try-cacth 方式で説明している。

● 正常終了メッセージを出力する

エラーメッセージだけでなく、正常終了した場合の結果メッセージも出力します。

```
package todo.app.todo;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.inject.Named;
```

```
import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import todo.domain.common.exception.BusinessException;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Named(value = "todoController")
@RequestScoped
public class TodoController {

    @EJB
    protected TodoService todoService;
    protected Todo todo = new Todo();
    protected List<Todo> todoList;

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }

    public Todo getTodo() {
        return todo;
    }

    public List<Todo> getTodoList() {
        return todoList;
    }

    @PostConstruct
```

```

public void init() {
    todoList = todoService.findAll();
}

public String create() {
    try {
        todoService.create(todo);
    } catch (BusinessException e) {
        FacesContext.getCurrentInstance()
            .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, e.getMessage(), null));
        return "list.xhtml";
    }
    // (1)
    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);
    // (2)
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Created successfully!", null));
}

return "list.xhtml?faces-redirect=true";
}
}

```

項目番号	説明
(1)	リダイレクト先にもメッセージが残るように FacesContext の Flash スコープを使用する設定を行う。Flash スコープはリダイレクト後の次の 1 回だけアクセスできるスコープである。
(2)	成功メッセージ用の FacesMessage として SEVERITY_INFO を設定

して、FacesContext に追加する。

FacesContext に設定したメッセージの種別で出力するメッセージのスタイルが変わるように xhtml を修正する。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "h
ttp://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
    <title>Todo List</title>
    <style type="text/css">
        .strike {
            text-decoration: line-through;
        }
        .alert {
            border: 1px solid;
            padding: 3px;
            width: 80%;
        }
        .alert-error {
            background-color: #c60f13;
            border-color: #970b0e;
            color: white;
        }
        .alert-success {
            background-color: #5da423;
            border-color: #457a1a;
        }
    </style>
</h:head>
<h:body>
    <h3>Todos</h3>
    <table border="1">
        <thead>
            <tr>
                <th>Title</th>
                <th>Description</th>
                <th>Due Date</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Buy milk</td>
                <td>Get some milk at the supermarket</td>
                <td>2023-10-01</td>
                <td>
                    <button type="button" value="Edit" />
                    <button type="button" value="Delete" />
                </td>
            </tr>
            <tr>
                <td>Call mom</td>
                <td>Ask mom how she's doing</td>
                <td>2023-10-02</td>
                <td>
                    <button type="button" value="Edit" />
                    <button type="button" value="Delete" />
                </td>
            </tr>
            <tr>
                <td>Walk dog</td>
                <td>Take the dog for a walk</td>
                <td>2023-10-03</td>
                <td>
                    <button type="button" value="Edit" />
                    <button type="button" value="Delete" />
                </td>
            </tr>
        </tbody>
    </table>
</h:body>
</html>
```

```

        color: white;
    }

</style>

</h:head>

<h:body>

    <!-- (1) -->
    <h:messages layout="table" styleClass="alert alert-success"
s" rendered="#{facesContext.maximumSeverity.ordinal == 0}" />

    <!-- (2) -->
    <h:messages layout="table" styleClass="alert alert-error"
rendered="#{facesContext.maximumSeverity.ordinal > 0}"/>

    <h2>Create Todo</h2>
    <h:form>

        <h:panelGrid columns="2">
            <h:outputLabel value="Title: " />
            <h:inputText value="#{todoController.todo.todoTitle}" />
            <h:commandButton value="Create" action="#{todoController.create}" />
        </h:panelGrid>
    </h:form>

    <h2>Todo List</h2>

    <h:dataTable value="#{todoController.todoList}" var="todo">
        <h:column>
            <f:facet name="header">
                <h:outputText value="Title" />
            </f:facet>
            <h:outputText value="#{todo.todoTitle}" rendered=""

```

```

#{!todo.finished} " />

        <h:outputText value="#{todo.todoTitle}" rendered="#{todo.finished}" class="strike" />

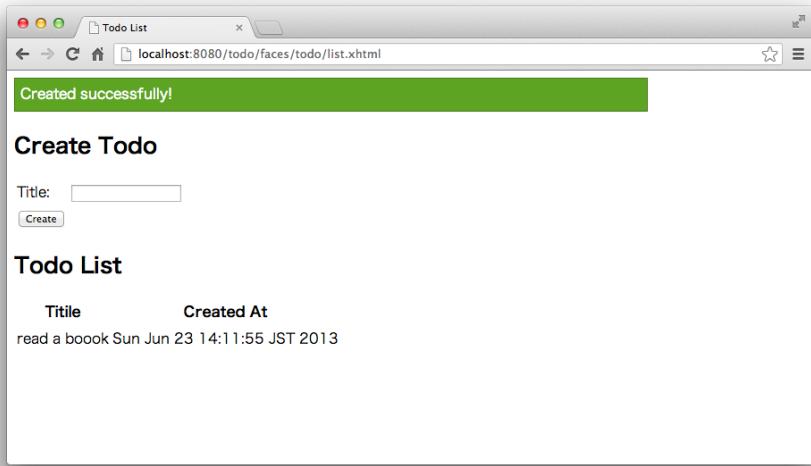
    </h:column>
    <h:column>

        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>
        <h:outputText value="#{todo.createdAt}" />
    </h:column>

</h:dataTable>
</h:body>
</html>

```

項目番号	説明
(1)	成功メッセージ用の<h:messages>タグを別途用意する。表示条件として SERVERITY に関する条件を rendered 属性に指定する。
(2)	SEVERITY が ERROR 以上の場合のみエラーメッセージになるように条件を追加する。



■ Todo 完了

次に完了処理を行うための画面遷移を追加します。

● TodoController の修正

"Create TODO" とほぼ同様、 TodoController に対して finish メソッドを追加します。

```
package todo.app.todo;

import java.util.List;
import javax.annotation.PostConstruct;
import javax.ejb.EJB;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
```

```
import todo.domain.common.exception.BusinessException;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Named(value = "todoController")
@RequestScoped
public class TodoController {

    @EJB
    protected TodoService todoService;
    protected Todo todo = new Todo();
    protected List<Todo> todoList;

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }

    public Todo getTodo() {
        return todo;
    }

    public List<Todo> getTodoList() {
        return todoList;
    }

    @PostConstruct
    public void init() {
        todoList = todoService.findAll();
    }
}
```

```

public String create() {
    try {
        todoService.create(todo);
    } catch (BusinessException e) {
        FacesContext.getCurrentInstance()
            .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, e.getMessage(), null));
        return "list.xhtml";
    }
    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Created successfully!", null));
}

return "list.xhtml?faces-redirect=true";
}

public String finish(Integer todoId) {
    try {
        todoService.finish(todoId);
    } catch (BusinessException e) {
        FacesContext.getCurrentInstance()
            .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, e.getMessage(), null));
        return "list.xhtml";
    }
    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);
    FacesContext.getCurrentInstance()

```

```
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Finished successfully!", null));  
  
    return "list.xhtml?faces-redirect=true";  
}  
}
```

● list.xhtml の修正

xhtml に完了ボタンを追加する。

```
<?xml version='1.0' encoding='UTF-8' ?>  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://xmlns.jcp.org/jsf/html"  
      xmlns:f="http://xmlns.jcp.org/jsf/core">  
  
<h:head>  
    <title>Todo List</title>  
    <style type="text/css">  
        .strike {  
            text-decoration: line-through;  
        }  
        .alert {  
            border: 1px solid;  
            padding: 3px;  
            width: 80%;  
        }  
        .alert-error {  
            background-color: #c60f13;  
            border-color: #970b0e;  
            color: white;  
        }  
    </style>  
  </h:head>  
  <h:body>  
    <h3>Todos</h3>  
    <table border="1">  
        <thead>  
            <tr>  
                <th>Title</th>  
                <th>Description</th>  
                <th>Actions</th>  
            </tr>  
        </thead>  
        <tbody>  
            <tr>  
                <td>Buy milk</td>  
                <td>Buy some milk at the supermarket</td>  
                <td>  
                    <h:button value="Edit" type="button"/>  
                    <h:button value="Delete" type="button"/>  
                </td>  
            </tr>  
            <tr>  
                <td>Buy bread</td>  
                <td>Buy some bread at the supermarket</td>  
                <td>  
                    <h:button value="Edit" type="button"/>  
                    <h:button value="Delete" type="button"/>  
                </td>  
            </tr>  
        </tbody>  
    </table>  
    <h3>Completed</h3>  
    <table border="1">  
        <thead>  
            <tr>  
                <th>Title</th>  
                <th>Description</th>  
            </tr>  
        </thead>  
        <tbody>  
            <tr>  
                <td>Buy milk</td>  
                <td>Buy some milk at the supermarket</td>  
            </tr>  
            <tr>  
                <td>Buy bread</td>  
                <td>Buy some bread at the supermarket</td>  
            </tr>  
        </tbody>  
    </table>  
    <h3>Alert</h3>  
    <div class="alert" style="display: none;">  
        This is an alert message.  
    </div>  
    <h3>Error</h3>  
    <div class="alert-error" style="display: none;">  
        This is an error message.  
    </div>  
  </h:body>  
</html>
```

```

.alert-success {
    background-color: #5da423;
    border-color: #457ala;
    color: white;
}

</style>

</h:head>
<h:body>

    <h:messages layout="table" styleClass="alert alert-succes
s" rendered="#{facesContext.maximumSeverity.ordinal == 0}" />
    <h:messages layout="table" styleClass="alert alert-error"
rendered="#{facesContext.maximumSeverity.ordinal > 0}"/>

    <h2>Create Todo</h2>
    <h:form>

        <h:panelGrid columns="2">
            <h:outputLabel value="Title: " />
            <h:inputText value="#{todoController.todo.todoTitl
e}" />
            <h:commandButton value="Create" action="#{todoCont
roller.create}" />
        </h:panelGrid>
    </h:form>

    <h2>Todo List</h2>

    <h:dataTable value="#{todoController.todoList}" var="todo
">
        <h:column>
            <f:facet name="header">
                <h:outputText value="Titile" />
            </f:facet>

```

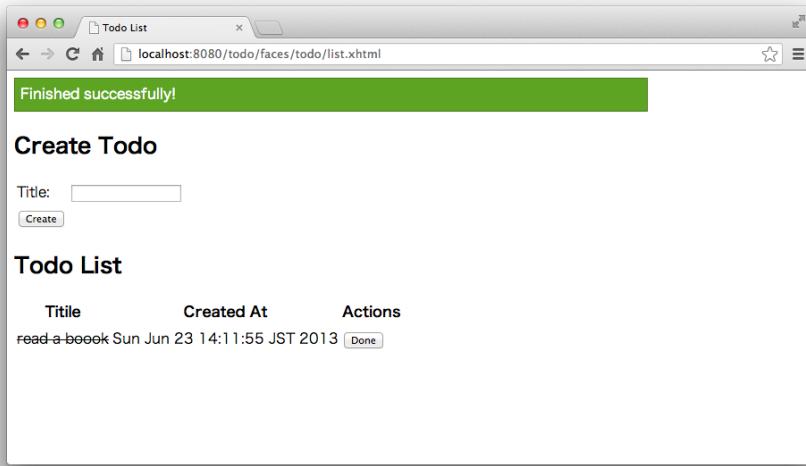
```

        <h:outputText value="#{todo.todoTitle}" rendered="#{!todo.finished}" />
        <h:outputText value="#{todo.todoTitle}" rendered="#{todo.finished}" class="strike" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>
        <h:outputText value="#{todo.createdAt}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Actions" />
        </f:facet>
        <h:form>
            <!-- (1) -->
            <h:commandButton value="Done" action="#{todoController.finish(todo.todoId)}" />
        </h:form>
    </h:column>

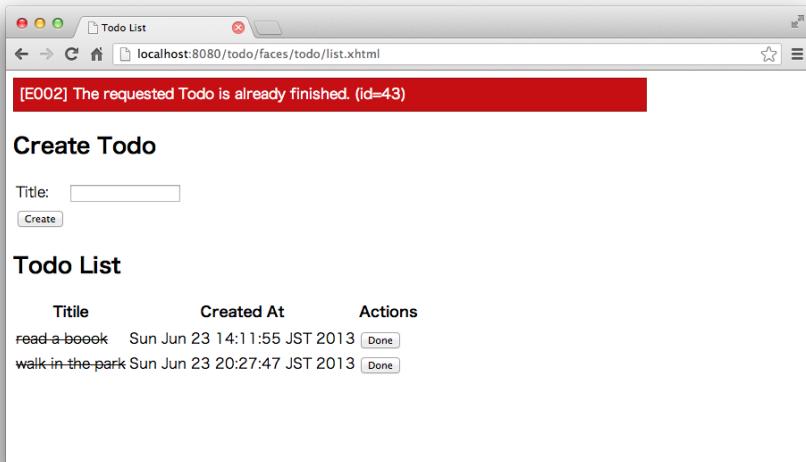
</h:dataTable>
</h:body>
</html>

```

項目番号	説明
(1)	完了ボタンを作成する。finish メソッドに todoId も渡す。



Todo を一件作成したのち、「Done」ボタンを押下し完了します。その後、もう一度「Done」ボタンを押下すると E002 のエラーが表示されます。



完了済みの場合、”Done”ボタンを非表示にするための設定を行います。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
        <title>Todo List</title>
        <style type="text/css">
            .strike {
                text-decoration: line-through;
            }
            .alert {

```

```

        border: 1px solid;
        padding: 3px;
        width: 80%;
    }

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457ala;
    color: white;
}


```

</style>

</h:head>

<h:body>

```

<h:messages layout="table" styleClass="alert alert-success"
    rendered="#{facesContext.maximumSeverity.ordinal == 0}" />

<h:messages layout="table" styleClass="alert alert-error"
    rendered="#{facesContext.maximumSeverity.ordinal > 0}"/>

<h2>Create Todo</h2>
<h:form>

    <h:panelGrid columns="2">
        <h:outputLabel value="Title: " />
        <h:inputText value="#{todoController.todo.todoTitle}" />
        <h:commandButton value="Create" action="#{todoController.create}" />
    </h:panelGrid>
</h:form>

```

```

<h2>Todo List</h2>

<h: dataTable value="#{todoController.todoList}" var="todo
">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Titile" />
        </f:facet>
        <h:outputText value="#{todo.todoTitle}" rendered="#
{!todo.finished}" />
        <h:outputText value="#{todo.todoTitle}" rendered="#
{todo.finished}" class="strike" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>
        <h:outputText value="#{todo.createdAt}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Actions" />
        </f:facet>
        <h:form>
            <!-- (1) -->
            <h:commandButton value="Done" action="#{todoCon
troller.finish(todo.todoId)}" rendered="#{!todo.finished}" />
        </h:form>
    </h:column>

```

```
</h:body>  
</html>
```

項番	説明
(1)	finished が false の場合のみボタンが表示されるように rendered 属性に条件を設定する。

■ Todo 削除

最後に削除用の画面遷移を追加します。

● TodoController の修正

finish 同様に delete メソッドを追加します。

```
package todo.app.todo;  
  
import java.util.List;  
import javax.annotation.PostConstruct;  
import javax.ejb.EJB;  
import javax.inject.Named;  
import javax.enterprise.context.RequestScoped;  
import javax.faces.application.FacesMessage;  
import javax.faces.context.FacesContext;  
import todo.domain.common.exception.BusinessException;  
import todo.domain.model.Todo;  
import todo.domain.service.todo.TodoService;  
  
@Named(value = "todoController")
```

```
@RequestScoped
public class TodoController {

    @EJB
    protected TodoService todoService;
    protected Todo todo = new Todo();
    protected List<Todo> todoList;

    /**
     * Creates a new instance of TodoController
     */
    public TodoController() {
    }

    public Todo getTodo() {
        return todo;
    }

    public List<Todo> getTodoList() {
        return todoList;
    }

    @PostConstruct
    public void init() {
        todoList = todoService.findAll();
    }

    public String create() {
        try {
            todoService.create(todo);
        } catch (BusinessException e) {

```

```

        FacesContext.getCurrentInstance()
            .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, e.getMessage(), null));
        return "list.xhtml";
    }

    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);

    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Created successfully!", null));

    return "list.xhtml?faces-redirect=true";
}

public String finish(Integer todoId) {
    try {
        todoService.finish(todoId);
    } catch (BusinessException e) {
        FacesContext.getCurrentInstance()
            .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_ERROR, e.getMessage(), null));
        return "list.xhtml";
    }

    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);

    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Finished successfully!", null));

    return "list.xhtml?faces-redirect=true";
}

```

```
public String delete(Integer todoId) {
    todoService.delete(todoId);
    FacesContext.getCurrentInstance().getExternalContext().getFlash().setKeepMessages(true);
    FacesContext.getCurrentInstance()
        .addMessage(null, new FacesMessage(FacesMessage.SEVERITY_INFO, "Deleted successfully!", null));
}

return "list.xhtml?faces-redirect=true";
}
```

● list.xhtml の修正

完了ボタン同様に削除ボタンを追加します。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
    <title>Todo List</title>
    <style type="text/css">
        .strike {
            text-decoration: line-through;
        }
        .alert {

```

```

        border: 1px solid;
        padding: 3px;
        width: 80%;
    }

.alert-error {
    background-color: #c60f13;
    border-color: #970b0e;
    color: white;
}

.alert-success {
    background-color: #5da423;
    border-color: #457ala;
    color: white;
}

</style>

</h:head>

<h:body>

    <h:messages layout="table" styleClass="alert alert-success"
    s" rendered="#{facesContext.maximumSeverity.ordinal == 0}" />

    <h:messages layout="table" styleClass="alert alert-error"
    rendered="#{facesContext.maximumSeverity.ordinal > 0}"/>

    <h2>Create Todo</h2>

    <h:form>

        <h:panelGrid columns="2">
            <h:outputLabel value="Title: " />
            <h:inputText value="#{todoController.todo.todoTitle}" />
            <h:commandButton value="Create" action="#{todoController.create}" />
        </h:panelGrid>
    </h:form>

```

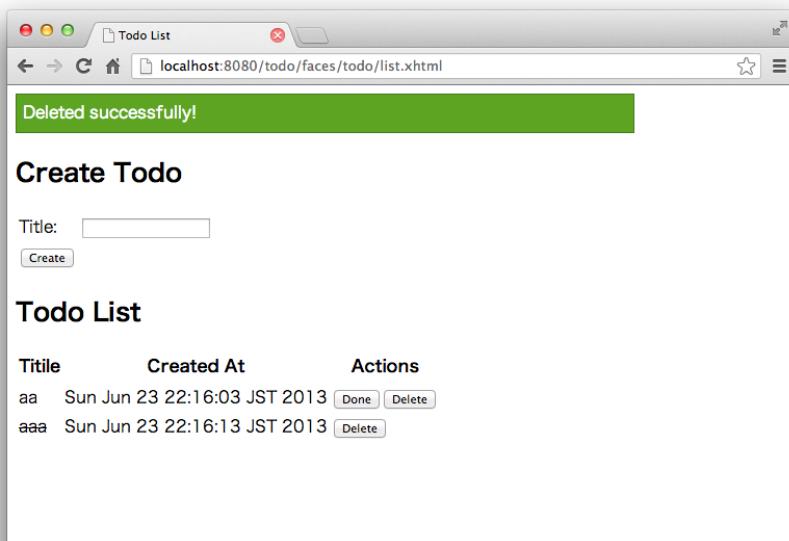
```

<h2>Todo List</h2>

<h: dataTable value="#{todoController.todoList}" var="todo
">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Titile" />
        </f:facet>
        <h:outputText value="#{todo.todoTitle}" rendered="#
{!todo.finished}" />
        <h:outputText value="#{todo.todoTitle}" rendered="#
{todo.finished}" class="strike" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Created At" />
        </f:facet>
        <h:outputText value="#{todo.createdAt}" />
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Actions" />
        </f:facet>
        <h:form>
            <h:commandButton value="Done" action="#{todoCon
troller.finish(todo.todoId)}" rendered="#{!todo.finished}" />
            <h:commandButton value="Delete" action="#{todoC
ontroller.delete(todo.todoId)}" />
        </h:form>
    </h:column>

```

```
</h:DataTable>  
</h:body>  
</html>
```

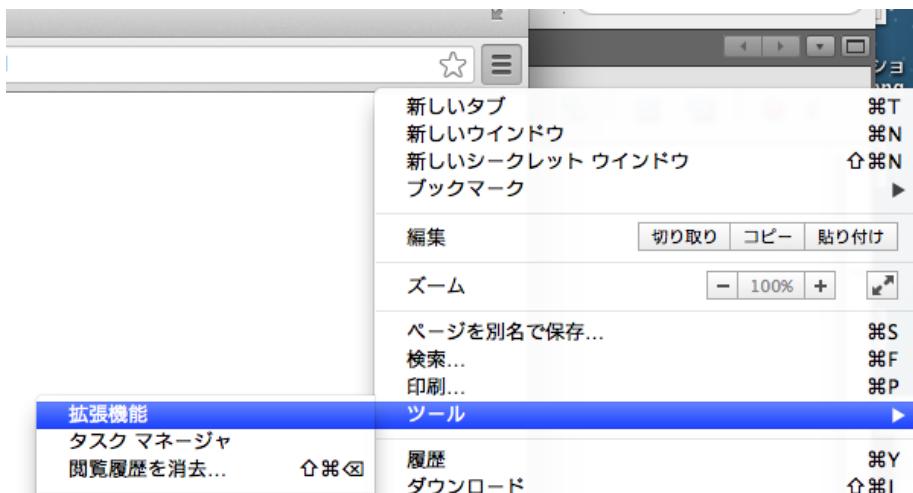


[4-4] JAX-RS で REST API を作成

次に Todo 管理業務処理を JAX-RS で REST-API として公開します。

■ Dev HTTP Client のインストール

まず REST クライアントとして Chrome の拡張機能である「Dev HTTP Client」をインストールします。Chrome の「ツール」→「拡張機能」を選択してください。



「他の拡張機能を見る」のリンクを押下してください。



検索フォームに「dev http client」を入力して検索します。

Chrome ウェブストア

dev http client

ホーム

人気のアイテム

サークル

Dev HTTP Client に対して「CHROME に追加」ボタンを押下します。

アプリ

アプリの他の検索結果

Dev HTTP Client

URL: sprintapi.org ✓

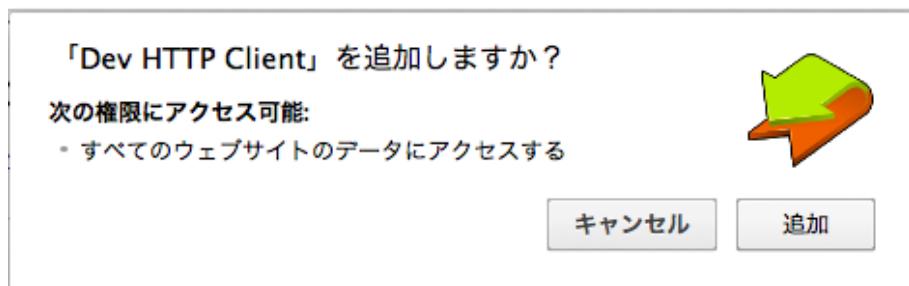
Easily construct custom HTTP requests, save them permanently, take advantage of variables and contexts.

+ CHROME に追加

デベロッパー ツール

★★★★★

「追加」ボタンを押下します。



新しいタブを作成する画面で Dev HTTP Client が追加されます。



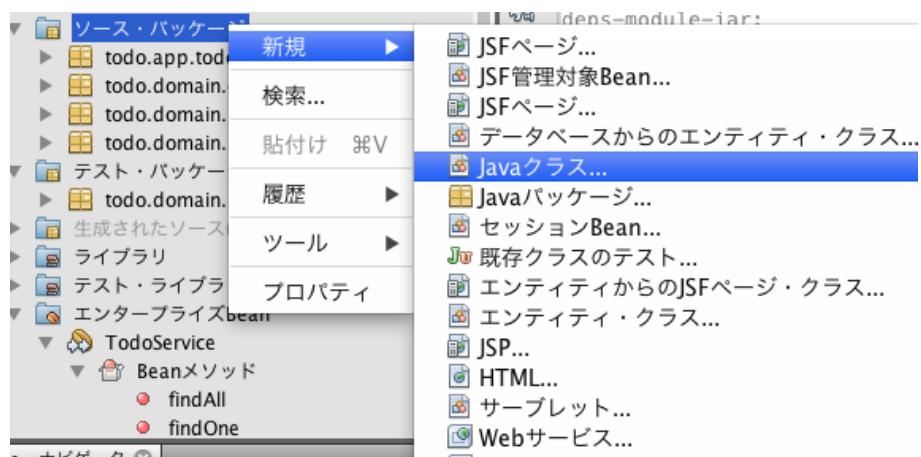
Store



Dev HTTP ...

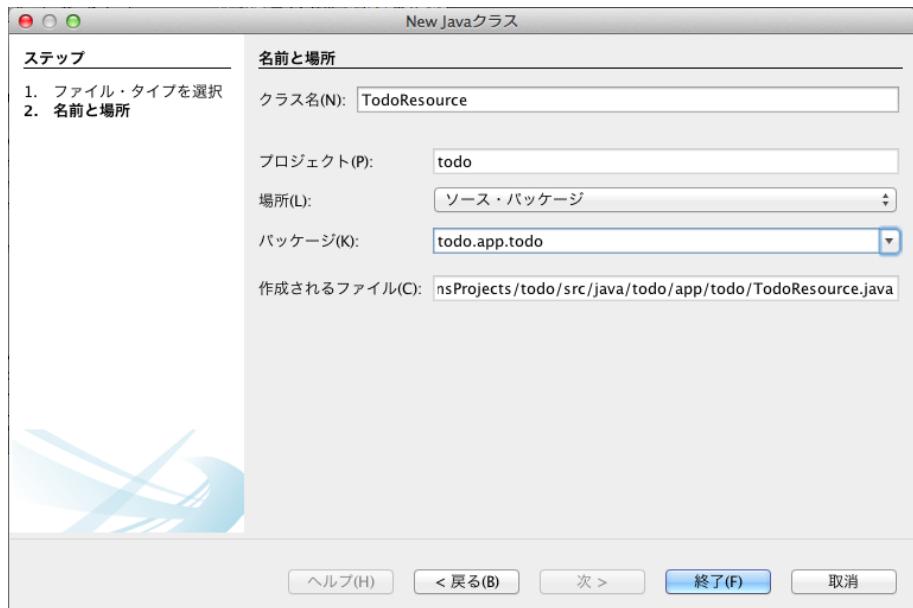
■ リソースクラスの作成

JAX-RS のリソースクラスを作成します。「Java クラス...」を選択し新しく Java のクラスを作成してください。



「New Java クラス」のウィザード画面にて下記の内容を入力し「終了(F)」ボタンを押下してください。

入力項目	入力値
クラス名	TodoResource
パッケージ	todo.app.todo



TodoResource クラスに下記の REST API を実装します。

HTTP メソッド	パス	ステータスコード	説明
GET	/todos	200 OK	Todo の全件取得
GET	/todos/{todoId}	200 OK	Todo の一件取得
POST	/todos	201 Created	Todo の新規作成

PUT	/todos/{todoId}	200 OK	Todo の更新(完了)
DELETE	/todos/{todoId}	204 No Content	Todo の削除

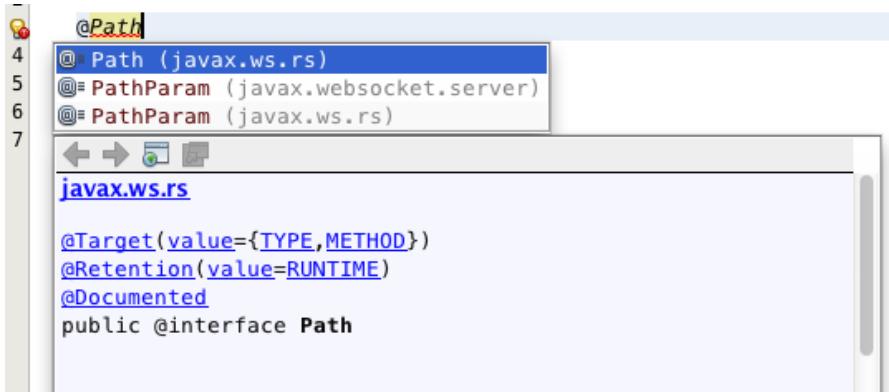
まずは TodoResource クラスに@Path アノテーションを付加します。

```
package todo.app.todo;

@Path
public class TodoResource {

}
```

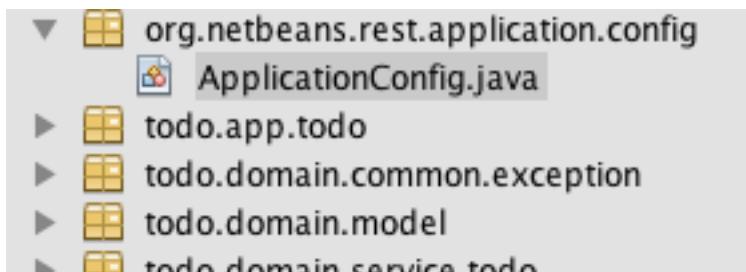
Ctrl+Space でコード補完します。



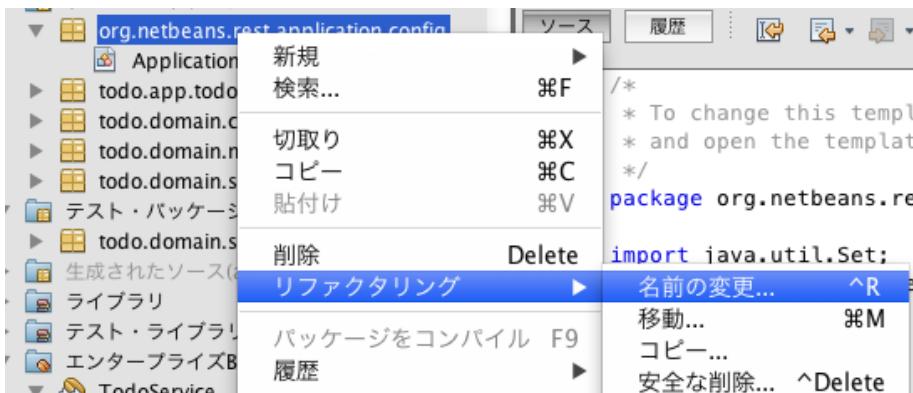
@Path のアノテーション内に”todos”を指定します。

```
1 package todo.app.todo;
2
3 import javax.ws.rs.Path;
4
5 @Path("todos")
6 public class TodoResource {
7
8 }
9
```

左の黄色のランプを選択し「Java EE6 仕様を使用して REST を構成します」を選択します。org.netbeans.rest.application.config パッケージに ApplicationConfig クラスが自動的に生成されます。

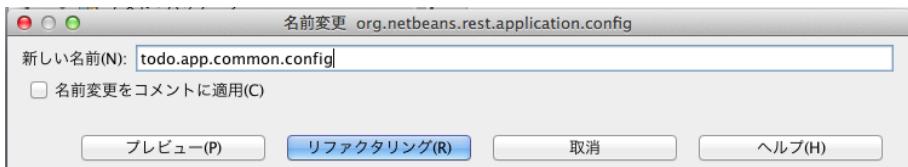


このクラスをリファクタリングしてパッケージ名を変更します。パッケージ名を右クリックして「リファクタリング」→「名前の変更」を選択してください。



「名前変更」の画面にて下記を入力して「リファクタリング(R)」ボタンを押下してください。

入力項目	入力値
新しい名前	todo.app.common.config



ApplicationConfig クラスは下記のようになります。このクラスにより、リソースクラスが REST API として公開されます。また@ApplicationPath に指定された”webresources”が REST API の上位パスとなります。（@Path に指定した”todos”と合わせて contextPath/webresources/todos が Todo リソースに対するパスとなります）

`addRestResourceClasses` メソッドは NetBeans によって自動で変更されるため、修正しないでください。

```
package todo.app.common.config;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<Class<?>>();
        // following code can be used to customize Jersey 2.0 JSON provider:
        try {
            Class jsonProvider = Class.forName("org.glassfish.jersey.jackson.JacksonFeature");
            // Class jsonProvider = Class.forName("org.glassfish.jersey.moxy.json.MoxyJsonFeature");
            // Class jsonProvider = Class.forName("org.glassfish.jersey.jettison.JettisonFeature");
            resources.add(jsonProvider);
        } catch (ClassNotFoundException ex) {
            java.util.logging.Logger.getLogger(getClass().getName()).log(java.util.logging.Level.SEVERE, null, ex);
        }
        addRestResourceClasses(resources);
    }
}
```

```
        return resources;
    }

    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically re-generated by NetBeans REST support t
o populate
     * given list with all resources defined in the project.
    */
    private void addRestResourceClasses(Set<Class<?>> resources)
{
    resources.add(todo.app.todo.TodoResource.class);
}

}
```

■ GET Todos

GET Todos(全件取得)の実装を行います。

● TodoResource の修正

getTodos メソッドを作成して全件取得用の処理を実装します。

```
package todo.app.todo;

import java.util.List;
import javax.ejb.EJB;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
```

```

import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB // (1)
    protected TodoService todoService;

    @GET // (2)
    @Produces(MediaType.APPLICATION_JSON) // (3)
    public List<Todo> getTodos() {
        return todoService.findAll(); // (4)
    }
}

```

項番	説明
(1)	@EJB アノテーションで EJB をインジェクションする。
(2)	@GET アノテーションを付けることで getTodos メソッドが HTTP メソッドの GET に対応していることを示す。
(3)	@Produces アノテーションでレスポンスの Content-Type を指定する。ここでは”application/json”を指定。
(4)	findAll の結果がシリализされ JSON をして出力される。

Dev HTTP Client を開いて URL

“localhost:8080/todo/webresources/todos”を入力します、メソッドに GET を指定して”Send”ボタンを押下してください。

The screenshot shows the Dev HTTP Client interface. In the REQUEST tab, a GET request is made to the URL "localhost:8080/todo/webresources/todos". The BODY section notes that only POST, PUT, PATCH methods can hold content. In the RESPONSE tab, a 200 OK status is shown with an elapsed time of 21ms. The Headers section includes Content-Length: 294 bytes, Content-Type: application/json, Date: 2013 Jun 23 23:37:47 -953ms, Server: GlassFish Server Open Source Edition 4.0, and X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server ...). The BODY section displays a JSON array of three todo items:

```
[{"todoId": 65, "todoTitle": "clean desktop", "finished": false, "createdAt": 1371997633281, "version": 1}, {"todoId": 66, "todoTitle": "Write a document", "finished": false, "createdAt": 1371997639709, "version": 1}, {"todoId": 68, "todoTitle": "send a e-mail", "finished": true, "createdAt": 1371997653355, "version": 2}]
```

■ GET Todo

GET Todo(1件取得)の実装を行います。

● TodoResource の修正

getTodo メソッドを実装します。

```
package todo.app.todo;

import java.util.List;
```

```
import javax.ejb.EJB;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Todo> getTodos() {
        return todoService.findAll();
    }

    @GET
    @Path("{todoId}") // (1)
    @Produces(MediaType.APPLICATION_JSON)
    public Todo getTodo(@PathParam("todoId") Integer todoId) { // (2)
        return todoService.findOne(todoId);
    }
}
```

項番	説明
(1)	@Path でパスの続きを指定する。”{パラメータ名}”と書くことにより、メソッド引数の@PathParam でパラメータを受け取ることができる。
(2)	@Path で設定したパラメータを受け取るための@PathParam に対応するパラメータ名を指定する。

Dev HTTP Client を開いて URL

“localhost:8080/todo/webresources/todos/XXXX”を入力します、メソッドに GET を指定して”Send”ボタンを押下してください。XXXX には先ほど GET Todos で出力された Todo のうちのどれかを指定してください。

下記のように一件分の Todo が JSON 形式で出力されます。

The screenshot shows the Dev HTTP Client interface. In the REQUEST section, a GET request is made to `localhost:8080/todo/webresources/todos/65`. The BODY tab is selected, showing the JSON response:

```
{
    "todoId": 65,
    "todoTitle": "clean desktop",
    "finished": false,
    "createdAt": 1371997633281,
    "version": 1
}
```

In the RESPONSE section, the status is **200 OK**. The Headers show standard HTTP headers like Date, Server, and Content-Type. The BODY tab displays the same JSON response as above. The elapsed time is listed as 166ms.

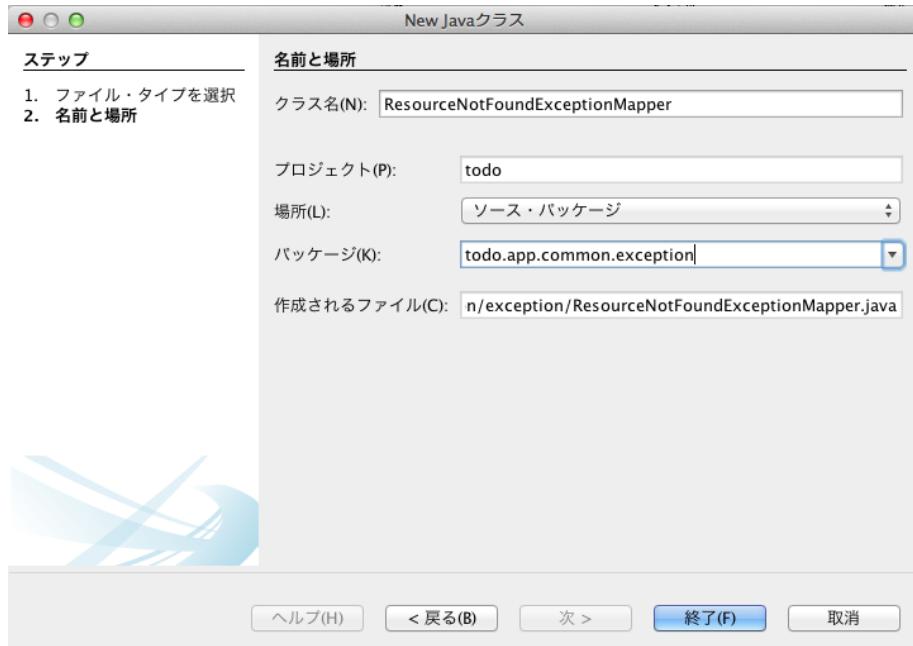
● 例外ハンドラの作成

ResourceNotFoundException が発生した場合に「404 NotFound」でエラー・メッセージを含む JSON 形式のデータを返すように例外ハンドラを作成します。

「New Java クラス」画面で下記の項目を入力し「終了(F)」ボタンを押します。

入力項目	入力値
クラス名	ResourceNotFoundExceptionMapper
パッケージ	todo.app.common.exception

JAX-RS の例外ハンドラとして javax.ws.rs.ext.ExceptionMapper インタフェースを実装します。



```
package todo.app.common.exception;

import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
```

```

import javax.ws.rs.ext.Provider;
import todo.domain.common.exception.ResourceNotFoundException;

@Provider // (1)
public class ResourceNotFoundExceptionMapper implements ExceptionMapper<ResourceNotFoundException> { // (2)

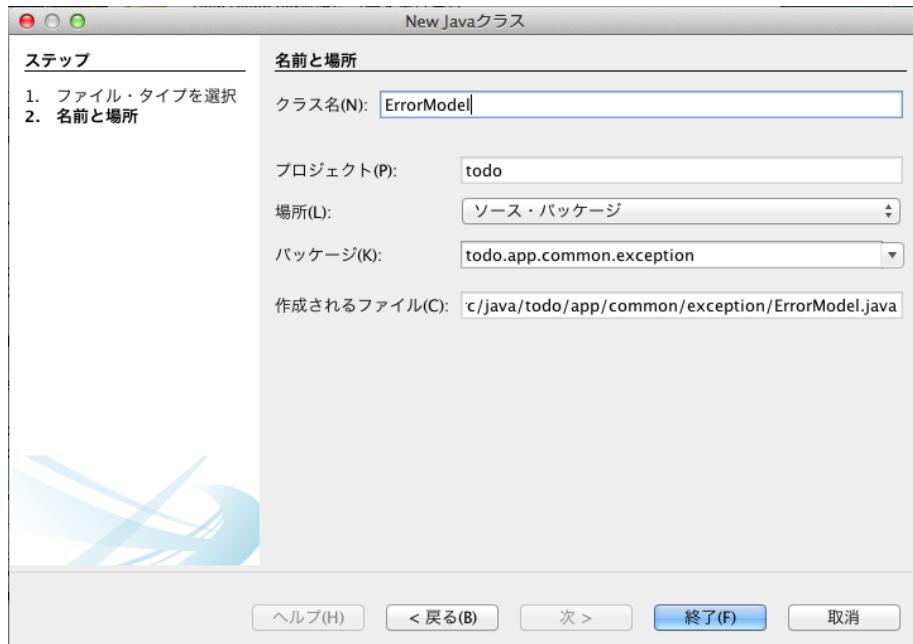
    @Override
    public Response toResponse(ResourceNotFoundException exception) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}

```

項番	説明
(1)	@Provider アノテーションを付けることで、JAX-RS で REST API として公開される。このアノテーションを付けると NetBeans によって ApplicationConfig に設定が自動で追加される。
(2)	この例外ハンドラが扱う例外クラスをジェネリクスで指定する。

エラー発生時に返す JSON 形式のデータを表現する Java Bean を作成します。「New Java クラス」 ウィンドウで下記の項目を入力し「終了(F)」 ボタンを押下します。

入力項目	入力値
クラス名	ErrorModel
パッケージ	todo.app.common.exception



以下のようにエラーメッセージのリストを持つように実装します。

```
package todo.app.common.exception;
```

```
import java.util.List;

public class ErrorModel {
    private final List<String> errorMessages;

    public ErrorModel(List<String> errorMessages) {
        this.errorMessages = errorMessages;
    }

    public List<String> getErrorMessages() {
        return errorMessages;
    }
}
```

```
package todo.app.common.exception;

import java.util.Arrays;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;
import todo.domain.common.exception.ResourceNotFoundException;

@Provider
public class ResourceNotFoundExceptionMapper implements ExceptionMapper<ResourceNotFoundException> {

    @Override
    public Response toResponse(ResourceNotFoundException exception) {
```

```

        ErrorModel errorModel = new ErrorModel(Arrays.asList(exce
ption.getMessage()));

        return Response.status(Response.Status.NOT_FOUND)
                .entity(errorModel).build(); // (1)

    }

}

```

項目番	説明
(1)	javax.ws.rs.core.Response クラスを利用して返却することで、返却するオブジェクトとステータスコードを指定できる。ここでは ErrorModel オブジェクトと 404 を指定する。

Dev HTTP Client で存在しない todoId を指定してリクエストを送信すると「404 Not Found」エラーで E404 のエラーメッセージを含む JSON 形式のデータが返却されます。

The screenshot shows the DEV HTTP CLIENT interface. In the REQUEST tab, a GET request is made to the URL `localhost:8080/todo/webresources/todos/1000`. The BODY section is empty, indicated by the message `Not available, only POST, PUT, PATCH method can hold a content.`. In the RESPONSE tab, the status is **404 Not Found**. The Headers section shows standard HTTP headers. The BODY section displays a JSON error message: `{"errorMessages": [{"id":1000, "message": "The requested Todo is not found."}]}`. The elapsed time is listed as 17ms.

■ POST Todos

POST Todos(1件作成)を実装します。新規作成した Todo への URL がレスポンスの Location ヘッダーに含まれるように実装します。

● TodoResource の修正

postTodos メソッドに新規作成処理を実装します。

```
package todo.app.todo;

import java.net.URI;
import java.util.List;
import javax.ejb.EJB;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Todo> getTodos() {
        return todoService.findAll();
    }
}
```

```

@GET
@Path("/{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo getTodo(@PathParam("todoId") Integer todoId) {
    return todoService.findOne(todoId);
}

@POST // (1)
@Produces(MediaType.APPLICATION_JSON)
public Response postTodos(Todo todo, @Context UriInfo uriInfo) { // (2)
    Todo createdTodo = todoService.create(todo);
    Integer todoId = createdTodo.getTodoId();
    URI newUri = uriInfo.getRequestUriBuilder()
        .path(todoId.toString()).build(); // (3)
    return Response.created(newUri).entity(createdTodo).build();
} // (4)
}

```

項番	説明
(1)	@POST アノテーションを付けることで postTodos メソッドが HTTP メソッドの POST に対応していることを示す。
(2)	@Context を付けることで URI に関する情報を持つ UriInfo オブジェクトを取得できる。
(3)	UriInfo を使用して、新規作成した Todo を取得するための URL を生成する。
(4)	Response.created メソッドに新規作成した Todo の URI を渡することで Location ヘッダの設定およびレスポンスステータス(201)の設定が行われる。

Dev HTTP Client を開いて URL

“localhost:8080/todo/webresources/todos”を入力します、メソッドに POST を指定し、HEADER に”Content-Type: application/json”を設定し、BODY に”{"todoTitle": "タイトル名"}”を設定して”Send”ボタンを押下します。

新規作成された Todo が JSON で返却されます。ステータスコードが 201 で
HEADERS に ”Location: http://localhost:8080/todo/webresources/todo/XXXX” が設定されていることを確認してください。XXXX には新規作成された Todo の ID が入ります。

REQUEST

HTTP ://localhost:8080/todo/webresources/todos

POST Send Save Reset

HEADERS: Content-Type: application/json

BODY: {"todoTitle": "Sleep"}

DEV HTTP CLIENT 0.8.9.8

REQUEST

HTTP ://localhost:8080/todo/webresources/todos

POST Send Save Reset

HEADERS: Content-Type: application/json

BODY: {"todoTitle": "Sleep"}

DEV HTTP CLIENT 0.8.9.8

RESPONSE

201 Created

elapsed time: 168ms

HEADERS: Content-Type: application/json

BODY: {"todoId": 69, "todoTitle": "Sleep", "finished": false, "createdDate": 137200002309, "version": 1}

download length: 88 bytes

● 例外ハンドラの追加

ResourceNotFoundExceptionMapper と同様に BusinessException 用の例外ハンドラである BusinessExceptionMapper を作成します。

ステータスコードを 409 Conflict にします。

```
package todo.app.common.exception;  
  
import java.util.Arrays;
```

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;
import todo.domain.common.exception.BusinessException;

@Provider
public class BusinessExceptionMapper implements ExceptionMapper<
BusinessException> {

    @Override
    public Response toResponse(BusinessException exception) {
        ErrorModel errorModel = new ErrorModel(Arrays.asList(exce
ption.getMessage()));
        return Response.status(Response.Status.CONFLICT)
            .entity(errorModel).build();
    }
}
```

未完了の Todo が 6 件以上になるように POST を繰り返すと「409 Conflict」エラーが発生し、E001 のエラーメッセージを含む JSON 形式のデータが返却されます。

The screenshot shows the DEV HTTP CLIENT interface. In the REQUEST section, a PUT request is made to `localhost:8080/todo/webresources/todos` with a JSON body containing `{"todoTitle": "Sleep"}`. In the RESPONSE section, a 409 Conflict status is returned, indicating that the count of unfinished todos must not be over 5, with the error message `[E001] The count of un-finished Todo must not be over 5.`

■ PUT Todo

PUT Todo(1 件更新)を実装します。

● TodoResource の修正

`putTodo` メソッドを作成し、完了処理を実装します。

```
package todo.app.todo;

import java.net.URI;
import java.util.List;
import javax.ejb.EJB;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
```

```
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Todo> getTodos() {
        return todoService.findAll();
    }

    @GET
    @Path("{todoId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Todo getTodo(@PathParam("todoId") Integer todoId) {
        return todoService.findOne(todoId);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
```

```

    public Response postTodos(Todo todo, @Context UriInfo uriInfo
o) {
    Todo createdTodo = todoService.create(todo);
    Integer todoId = createdTodo.getTodoId();
    URI newUri = uriInfo.getRequestUriBuilder()
        .path(todoId.toString()).build();
    return Response.created(newUri).entity(createdTodo).build
();
}

@PUT // (1)
@Path("/{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo putTodo(@PathParam("todoId") Integer todoId) {
    Todo todo = todoService.finish(todoId);
    return todo;
}
}

```

項目番	説明
(1)	@PUT アノテーションを付けることで putTodo メソッドが HTTP メソッドの PUT に対応していることを示す。

Dev HTTP Client を開いて URL

“localhost:8080/todo/webresources/todos/XXXX”を入力します、メソッドに PUT を指定して”Send”ボタンを押下します。XXXX には先ほど GET Todos で出力された Todo のうちのどれかを指定してください。

ボタンを押下すると、更新された Todo の JSON が返却されます。finished が true になり、version がインクリメントされていることを確認してください。

REQUEST

HTTP :// localhost:8080/todo/webresources/todos/65 PUT Send Save Reset

HEADERS form raw BODY length: 0 bytes

RESPONSE

200 OK elapsed time 196ms

HEADERS formatted raw BODY length: 95 bytes

Date: Sun, 23 Jun 2013 22:18:46 GMT
Server: GlassFish Server Open Source Edition 4.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.0 Java/Oracle Corporation/1.7)
Content-Length: 95
Content-Type: application/json

download

```
{"todoId": 65, "todoTitle": "clean desktop", "finished": true, "lastEdited": 1371997633281, "version": 2}
```

連続して Send ボタンを押すと E002 のエラーメッセージが返却されます。

REQUEST

HTTP :// localhost:8080/todo/webresources/todos/65 PUT Send Save Reset

HEADERS form raw BODY length: 0 bytes

RESPONSE

406 Not Acceptable elapsed time 15ms

HEADERS formatted raw BODY length: 76 bytes

Date: Sun, 23 Jun 2013 22:20:19 GMT
Server: GlassFish Server Open Source Edition 4.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.0 Java/Oracle Corporation/1.7)
Content-Length: 76
Content-Type: application/json

download

```
{"errorMessages": [{"E002": "The requested Todo is already finished. (id=65)"}]}
```

■ DELETE Todo

DELETE Todo(1件削除)を実装します。

● TodoResource の修正

deleteTodo メソッドを追加し削除処理を実装します。

```
package todo.app.todo;

import java.net.URI;
import java.util.List;
import javax.ejb.EJB;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;
```

```

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Todo> getTodos() {
    return todoService.findAll();
}

@GET
@Path("{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo getTodo(@PathParam("todoId") Integer todoId) {
    return todoService.findOne(todoId);
}

@POST
@Produces(MediaType.APPLICATION_JSON)
public Response postTodos(Todo todo, @Context UriInfo uriInfo) {
    Todo createdTodo = todoService.create(todo);
    Integer todoId = createdTodo.getTodoId();
    URI newUri = uriInfo.getRequestUriBuilder()
        .path(todoId.toString()).build();
    return Response.created(newUri).entity(createdTodo).build();
}

@PUT
@Path("{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo putTodo(@PathParam("todoId") Integer todoId) {
    Todo todo = todoService.finish(todoId);
}

```

```

        return todo;
    }

}

@DELETE
@Path("/{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public void deleteTodo(@PathParam("todoId") Integer todoId)
{ // (1)
    todoService.delete(todoId);
}
}

```

項目番号	説明
(1)	返り値を void にすることでステータスコードを 204 No Content にできる。

Dev HTTP Client を開いて URL

“localhost:8080/todo/webresources/todos/XXXX”を入力し、メソッドに DELETE を指定して”Send”ボタンを押下してください。XXXX には先ほど GET Todos で出力された Todo のうちのどれかを指定します。

The screenshot shows the Dev HTTP Client interface. In the REQUEST section, the URL is set to "localhost:8080/todo/webresources/todos/65". The method is selected as "DELETE". The BODY section is empty with a note: "Not available, only POST, PUT, PATCH method can hold a content.". In the RESPONSE section, the status is "204 No Content". The Headers show standard HTTP headers. The BODY section displays the message "NO CONTENT".

■ 入力チェックの追加

Bean Validationによる入力チェックはJAX-RSでも実装できます。

● TodoResourceに@Validを追加

入力チェック対象のオブジェクトに@Validアノテーションを付加します。

```
package todo.app.todo;

import java.net.URI;
import java.util.List;
import javax.ejb.EJB;
import javax.validation.Valid;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import todo.domain.model.Todo;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;
```

```

@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Todo> getTodos() {
    return todoService.findAll();
}

@GET
@Path("{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo getTodo(@PathParam("todoId") Integer todoId) {
    return todoService.findOne(todoId);
}

@POST
@Produces(MediaType.APPLICATION_JSON)
public Response postTodos(@Valid Todo todo, @Context UriInfo
uriInfo) { // (1)
    Todo createdTodo = todoService.create(todo);
    Integer todoId = createdTodo.getTodoId();
    URI newUri = uriInfo.getRequestUriBuilder()
        .path(todoId.toString()).build();
    return Response.created(newUri).entity(createdTodo).build();
}

@PUT
@Path("{todoId}")
@Produces(MediaType.APPLICATION_JSON)
public Todo putTodo(@PathParam("todoId") Integer todoId) {
    Todo todo = todoService.finish(todoId);
}

```

```

        return todo;
    }

    @DELETE
    @Path("{todoId}")
    @Produces(MediaType.APPLICATION_JSON)
    public void deleteTodo(@PathParam("todoId") Integer todoId) {
        todoService.delete(todoId);
    }
}

```

項目番	説明
(1)	Bean Validation のアノテーションがついている Todo クラスに対して入力チェックを行うために@Valid を付ける。

● 例外ハンドラの追加

Bean Validation による入力チェックでエラーが発生した場合は ConstraintViolationException がスローされます。この例外に対してもこれまで同様に例外ハンドラ(ConstraintViolationExceptionMapper)を作成します。

```

package todo.app.common.exception;

import java.util.ArrayList;
import java.util.List;
import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationException;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;

```

```

import javax.ws.rs.ext.Provider;

@Provider
public class ConstraintValidationExceptionMapper implements ExceptionMapper<ConstraintViolationException> {

    @Override
    public Response toResponse(ConstraintViolationException exception) {
        List<String> messages = new ArrayList<>();
        for (ConstraintViolation<?> cv : exception.getConstraintViolations()) { // (1)
            messages.add(cv.getMessage());
        }
        ErrorModel errorModel = new ErrorModel(messages);
        return Response.status(Response.Status.BAD_REQUEST)
            .entity(errorModel).build();
    }
}

```

項番	説明
(1)	入力チェックエラーに関する情報は ConstraintViolationException.getConstraintViolations() メソッド で取得できる。
(2)	入力チェックエラーに対しては 400 Bad Request エラーを返す。

todoTitle を空にして POST すると以下のように「400 Bad Request」エラーで入力チェックエラーメッセージが返却されます。2件のエラーメッセージが出力されることに注意してください。

REQUEST

HTTP POST /localhost:8080/todo/webresources/todos

HEADERS Content-Type: application/json

BODY {"todoTitle": ""}

RESPONSE

400 Bad Request

elapsed time 22ms

HEADERS Connection: close
Content-Length: 70 bytes
Content-Type: application/json
Date: 2013 Jun 24 13:15:06 -0500
Server: GlassFish Server Open Source Edition 4.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server ...)

BODY {"errorMessages": [{"may not be null", "size must be between 1 and 128"}]}

● バリデーショングループの作成

現状だと、todoTitile 以外のフィールド(createdAt)のチェックも実行されてしまうため、新規作成時にのみチェックが動作するようなグループを作成し、必要なフィールドのみチェックするように修正してください。

以下のような空インターフェースを作成します。

```
package todo.domain.model.group;

public interface Create { }
```

● Todo エンティティの修正

入力チェックルールのアノテーションにグループも加えます。

```
package todo.domain.model;

import todo.domain.model.group.Create;
```

```
import java.io.Serializable;
import java.util.Date;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.validation.groups.Default;
import javax.xml.bind.annotation.XmlRootElement;

@Entity
@Table(name = "TODO")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Todo.findAll", query = "SELECT t FROM Todo t"),
    @NamedQuery(name = "Todo.findById", query = "SELECT t FROM Todo t WHERE t.todoId = :todoId"),
    @NamedQuery(name = "Todo.findByTodoTitle", query = "SELECT t FROM Todo t WHERE t.todoTitle = :todoTitle"),
    @NamedQuery(name = "Todo.findByFinished", query = "SELECT t FROM Todo t WHERE t.finished = :finished")})
```

```
@NamedQuery(name = "Todo.findByCreatedAt", query = "SELECT t  
FROM Todo t WHERE t.createdAt = :createdAt"))  
  
public class Todo implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Basic(optional = false)  
    @Column(name = "TODO_ID")  
    private Integer todoId;  
  
    @Basic(optional = false)  
    @NotNull(groups={Default.class, Create.class})  
    @Size(min = 1, max = 128, groups={Default.class, Create.clas  
s}) // (1)  
    @Column(name = "TODO_TITLE")  
    private String todoTitle;  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "FINISHED")  
    private boolean finished;  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "CREATED_AT")  
    @Temporal(TemporalType.TIMESTAMP)  
    private Date createdAt;  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "VERSION")  
    @Version  
    private int version;
```

```
public Todo() {  
}  
  
public Todo(Integer todoId) {  
    this.todoId = todoId;  
}  
  
public Todo(Integer todoId, String todoTitle, boolean finished,  
Date createdAt, int version) {  
    this.todoId = todoId;  
    this.todoTitle = todoTitle;  
    this.finished = finished;  
    this.createdAt = createdAt;  
    this.version = version;  
}  
  
public Integer getTodoId() {  
    return todoId;  
}  
  
public void setTodoId(Integer todoId) {  
    this.todoId = todoId;  
}  
  
public String getTodoTitle() {  
    return todoTitle;  
}  
  
public void setTodoTitle(String todoTitle) {  
    this.todoTitle = todoTitle;  
}
```

```
public boolean isFinished() {
    return finished;
}

public void setFinished(boolean finished) {
    this.finished = finished;
}

public Date getCreatedAt() {
    return createdAt;
}

public void setCreatedAt(Date createdAt) {
    this.createdAt = createdAt;
}

public int getVersion() {
    return version;
}

public void setVersion(int version) {
    this.version = version;
}

@Override
public int hashCode() {
    int hash = 0;
    hash += (todoId != null ? todoId.hashCode() : 0);
    return hash;
}
```

```

@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the i
d fields are not set
    if (!(object instanceof Todo)) {
        return false;
    }
    Todo other = (Todo) object;
    if ((this.todoId == null && other.todoId != null) || (thi
s.todoId != null && !this.todoId.equals(other.todoId))) {
        return false;
    }
    return true;
}

@Override
public String toString() {
    return "todo.domain.model.Todo[ todoId=" + todoId + " ]";
}

}

```

項目番号	説明
(1)	<p>group 属性にルールが適用されるグループを指定する。Default はデフォルトのグループ名であり、グループを明示的に指定しない場合に適用される。</p> <p>この例のように設定すると、グループに Create を指定して入力チェックを実行すると todoTitle フィールドに対してのみ@NotNull と @Size に入力チェックが実施される。</p>

● TodoResource に@ConvertGroup を追加

```
package todo.app.todo;

import java.net.URI;
import java.util.List;
import javax.ejb.EJB;
import javax.validation.Valid;
import javax.validation.groups.ConvertGroup;
import javax.validation.groups.Default;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import todo.domain.model.Todo;
import todo.domain.model.group.Create;
import todo.domain.service.todo.TodoService;

@Path("todos")
public class TodoResource {

    @EJB
    protected TodoService todoService;
```

```

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Todo> getTodos() {
        return todoService.findAll();
    }

    @GET
    @Path("{todoId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Todo getTodo(@PathParam("todoId") Integer todoId) {
        return todoService.findOne(todoId);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response postTodos(@Valid @ConvertGroup(from = Default.class, to = Create.class) Todo todo, @Context UriInfo uriInfo)
    { // (1)
        Todo createdTodo = todoService.create(todo);
        Integer todoId = createdTodo.getTodoId();
        URI newUri = uriInfo.getRequestUriBuilder()
            .path(todoId.toString()).build();
        return Response.created(newUri).entity(createdTodo).build();
    }

    @PUT
    @Path("{todoId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Todo putTodo(@PathParam("todoId") Integer todoId) {

```

```

        Todo todo = todoService.finish(todoId);

        return todo;

    }

    @DELETE
    @Path("{todoId}")
    @Produces(MediaType.APPLICATION_JSON)
    public void deleteTodo(@PathParam("todoId") Integer todoId) {
        todoService.delete(todoId);
    }
}

```

項目番	説明
(1)	@ConvertGroup でバリデーショングループを指定する。 ※ここは正しいか要確認・・・

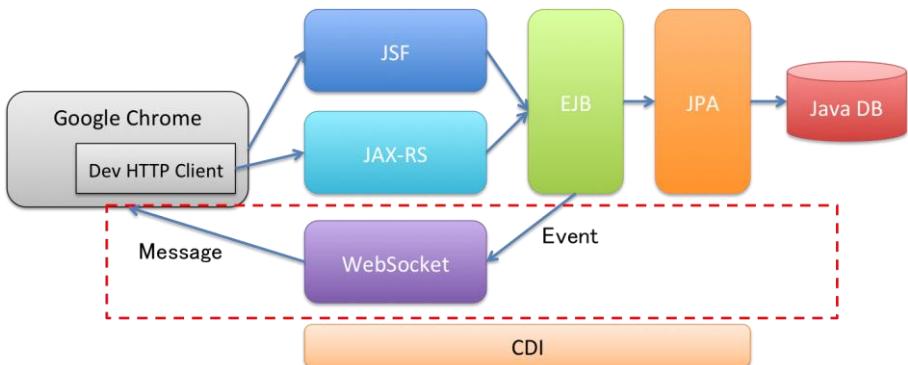
todoTitle を空にして POST を実行すると「400 Bad Request」のエラーメッセージが 1 件のみ(todoTitle に関するエラー)表示されます。

The screenshot shows the DEV HTTP CLIENT interface. In the REQUEST section, a POST request is being made to `localhost:8080/todo/webresources/todos`. The BODY contains a JSON object with a single field: `{"todoTitle": ""}`. In the RESPONSE section, the status is **400 Bad Request**. The response headers include `Connection: close`, `Content-Length: 62 bytes`, `Content-Type: application/json`, `Date: 2013 Jun 24 13:16:59 -156ms`, `Server: GlassFish Server Open Source Edition 4.0`, and `X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server ...)`. The response body is a JSON object with an `errorMessages` key containing an array of validation errors: `"size must be between 1 and 128"`.

これにて REST-API の実装は完了です。

第5章 WebSocket でリアルタイムイベントモニタリング機能追加

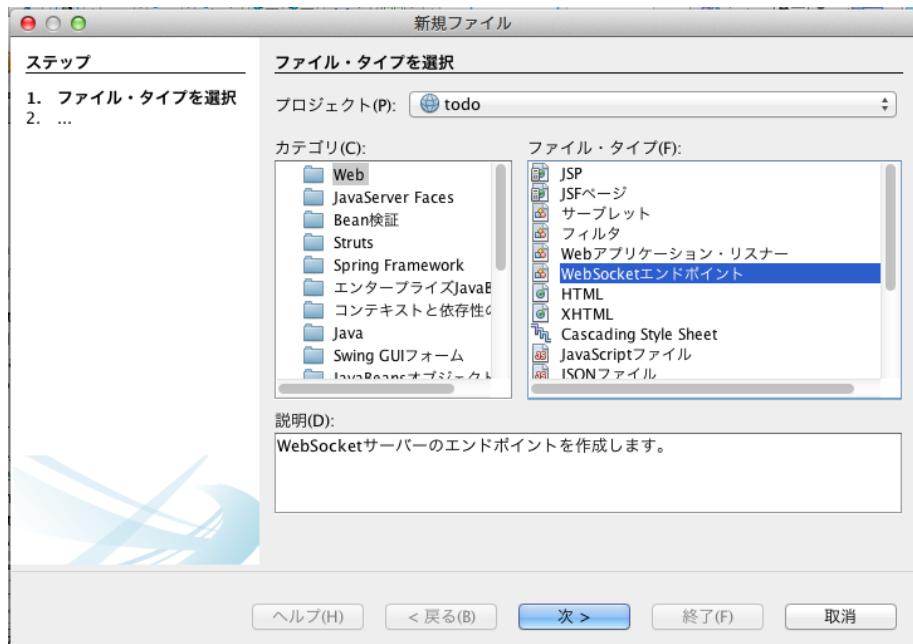
ここまで作成したアプリケーションに機能追加してトランザクションイベント(新規作成、更新、削除)のモニタリングを行います。リアルタイムでモニタリングするために Java EE7 からサポートされた WebSocket を用います。WebSocket を追加する事で全体アーキテクチャは下記の構成になります。イベント処理には CDI を使用します。



[5-1] WebSocket エンドポイントの作成

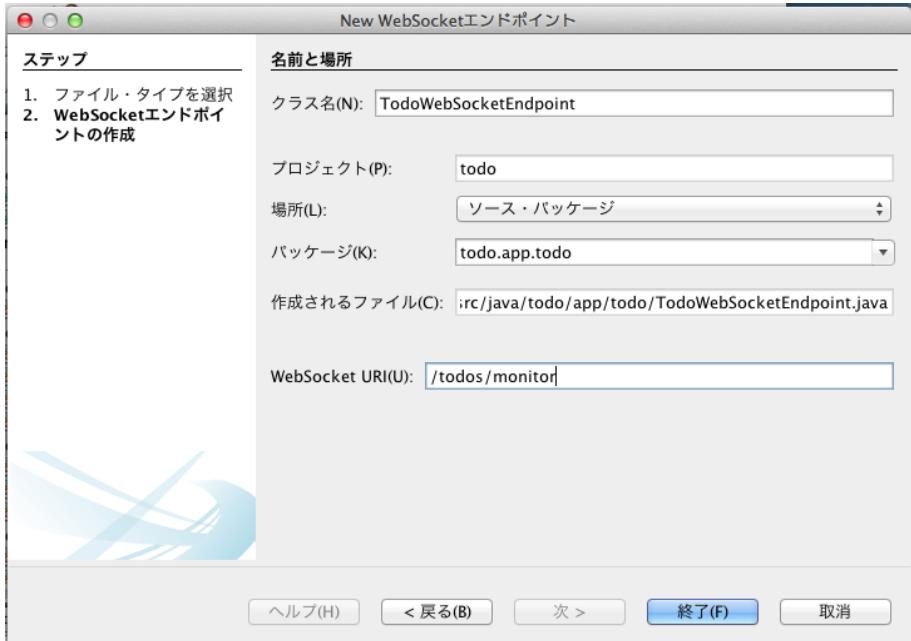
■ エンドポイントクラスを作成

新規ファイルで「Web」→「WebSocket エンドポイント」を選択して「次 >」ボタンを押下します。



「New WebSocket エンドポイント」のウィンドウで下記を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
クラス名	TodoWebSocketEndpoint
パッケージ	todo.app.todo
WebSocket URI	/todos/monitor



自動生成されたクラスを元に、下記のように修正します。

```
package todo.app.todo;

import java.io.IOException;
import java.util.Collections;
```

```
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/todos/monitor")
public class TodoWebSocketEndPoint {

    static final Set<Session> sessions = Collections.synchronized
Set(new HashSet<Session>()); // (1)
    private static final Logger logger = Logger.getLogger(TodoWeb
SocketEndPoint.class.getName());

    @OnOpen // (2)
    public void onOpen(Session session) {
        try {
            session.getBasicRemote().sendText("opened"); // (4)
            logger.log(Level.INFO, "opened {0}", session);
            sessions.add(session);
            logger.log(Level.INFO, "{0} sessions", sessions.size
());
        } catch (IOException ex) {
            logger.log(Level.SEVERE, null, ex);
        }
    }
}
```

```

    @OnClose // (3)
    public void onClose(final Session session) {
        logger.log(Level.INFO, "closed {0}", session);
        sessions.remove(session);
        logger.log(Level.INFO, "{0} sessions", sessions.size());
    }

    @OnError // (4)
    public void onError(Throwable e) {
        logger.log(Level.SEVERE, "Unexcepted Exception happened!
", e);
    }
}

```

項番	説明
(1)	WebSocket クライアント(Peer)を保持するための Set。
(2)	@OnOpen アノテーションをつけてクライアントとの接続開始時の処理を定義する。ここではテキストメッセージを送信したあと Set に Session オブジェクトを追加する。
(3)	@OnClose アノテーションをつけてクライアントとの接続終了時の処理を定義する。ここでは Set から Session オブジェクトを削除する。
(4)	@OnError アノテーションで例外ハンドリング処理を定義する。

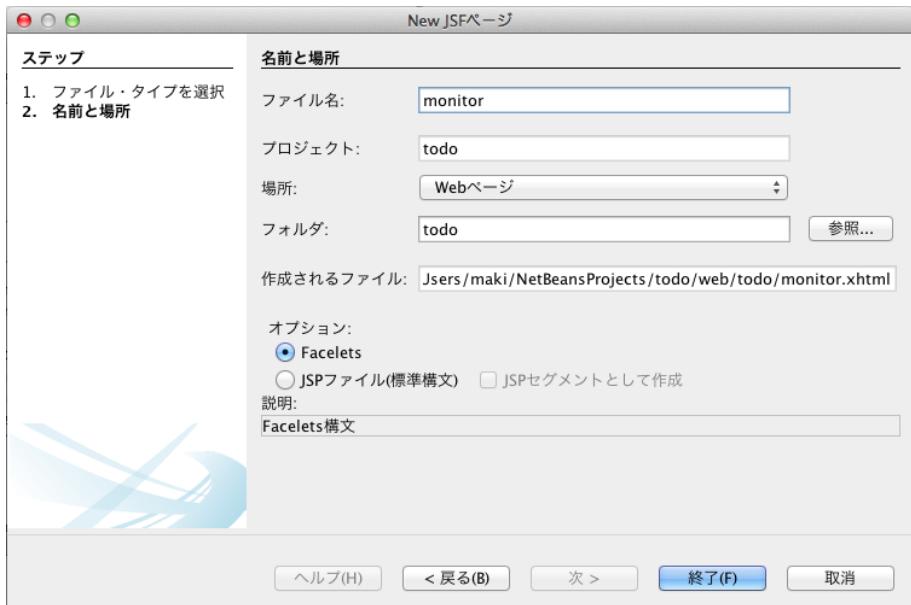
■ WebSocket の JavaScript API でモニタ画面作成

モニタリングを行う画面を作成します。ここでは Facelets で作成しますが、素の HTML でも問題ありません。

新規ファイルより「JavaServer Faces」→「JSF ページ」を選択して「次 >」ボタンを押下します。

「New JSF ページ」 ウィンドウで下記の項目を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
ファイル名	monitor
フォルダ	todo
オプション	Facelets



自動生成された `xhtml` ファイルに対して下記のように修正します。

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

```

<h:head>
    <title>Todo Monitor</title>
    <script>
        var TodoMonitor = function(path, targetId)
        {
            this.path = path;
            this.target = document.getElementById(targetId);
        };
        TodoMonitor.prototype.open = function()
        {
            if (!this.session) {
                this.session = new WebSocket('ws://' + this.path);
            } // (1)
            var self = this;
            this.session.onmessage = function(evt) {
                self.target.innerHTML = new Date(evt.timeStamp) + ':' + evt.data + '<br/>' + self.target.innerHTML;
            }; // (2)
            this.session.onclose = function(evt) {
                self.target.innerHTML = new Date(evt.timeStamp) + ': closed<br />' + self.target.innerHTML;
            }; // (3)
            this.session.onerror = function(evt) {
                self.target.innerHTML = new Date(evt.timeStamp) + ': exception happened!<br />' + self.target.innerHTML;
            }; // (4)
        }
        return false;
    };
    TodoMonitor.prototype.close = function() {
        if (this.session) {
            this.session.close(); // (5)
        }
    };

```

```

        this.session = null;
    }
    return false;
};

</script>

</h:head>

<h:body>
    <h1>Todo Monitor</h1>
    <div id="result" style="border: 1px solid; height: 300px; overflow: scroll;"></div>
    <button onclick="todoMonitor.open()">Open</button>
    <button onclick="todoMonitor.close()">Close</button>
    <script>
        var todoMonitor = new TodoMonitor(document.location.host + '/todo/todos/monitor', 'result');
        window.onunload = function() {
            todoMonitor.close();
        };
    </script>
</h:body>
</html>

```

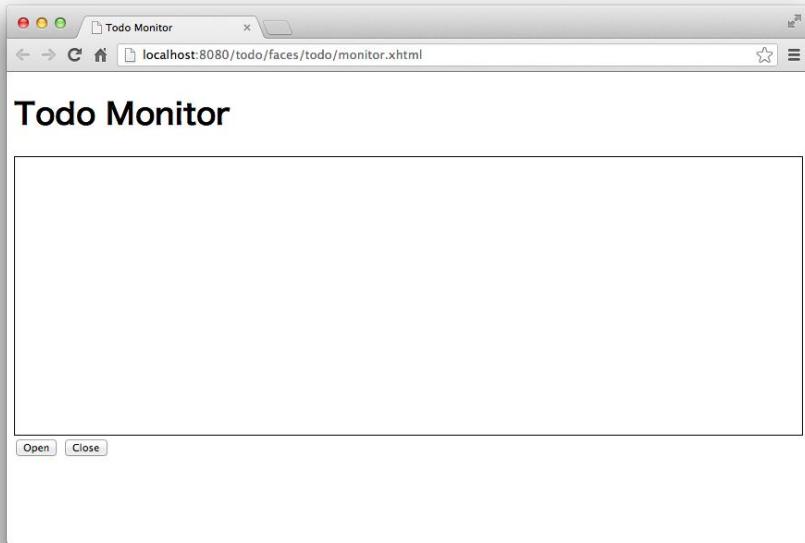
項目番号	説明
(1)	標準の WebSocket オブジェクトを作成して、接続を開始する。
(2)	onmessage プロパティにメッセージを受信した際のイベントハンドラを設定する。
(3)	onclose プロパティに接続を終了した際のイベントハンドラを設定する。
(4)	onerror プロパティにエラーが発生した際のイベントハンドラを設定する。

- (5) 接続を終了する。

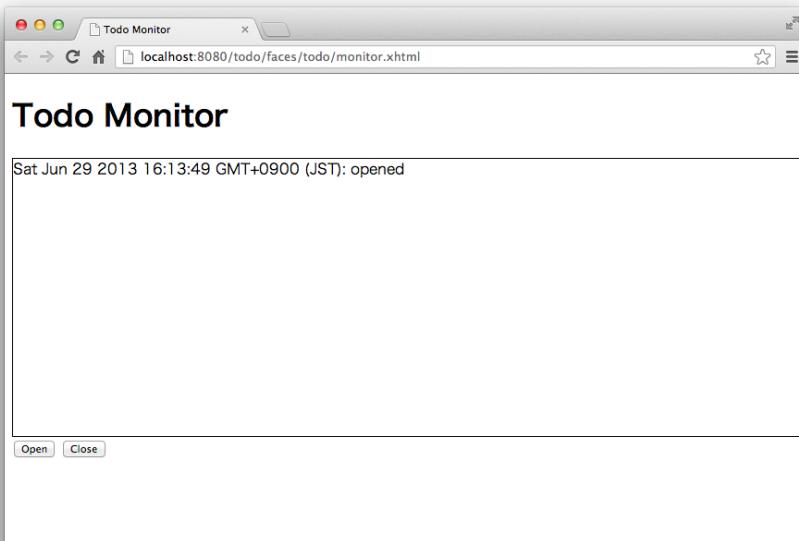
monitor.xhtml を右クリックして「ファイルを実行」を選択します。



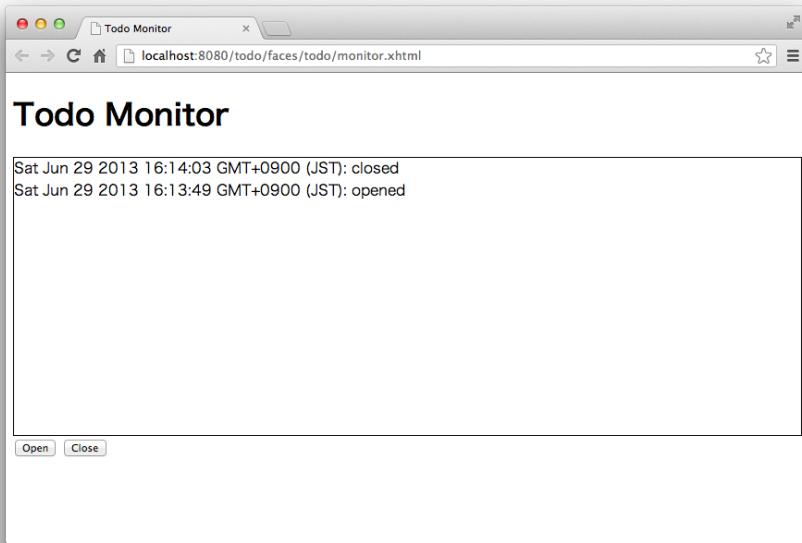
「ファイルを実行」すると自動的にブラウザが起動しモニタ画面が表示されます。



ここで「Open」ボタンを押下すると WebSocket のサーバー・エンドポイントに接続します。



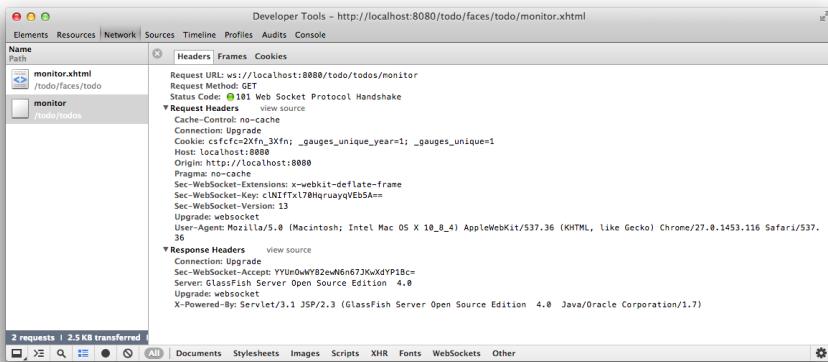
「Close」ボタンを押下すると接続が解除されます。



この時、GlassFish のサーバー・ログには下記のようなログが出力されます。

```
情報: opened SessionImpl{uri=/todo/todos/monitor, id='5d115c90-3411-468d-a942-46d9e7dc1f3c', endpoint=EndpointWrapper{endpointClass=null, endpoint=org.glassfish.tyrus.core.AnnotatedEndpoint@46e505ac, uri='/todo/todos/monitor', contextPath='/todo'}}  
情報: 1 sessions  
情報: closed SessionImpl{uri=/todo/todos/monitor, id='5d115c90-3411-468d-a942-46d9e7dc1f3c', endpoint=EndpointWrapper{endpointClass=null, endpoint=org.glassfish.tyrus.core.AnnotatedEndpoint@46e505ac, uri='/todo/todos/monitor', contextPath='/todo'}}  
情報: 0 sessions
```

Chome の Developer Tools を開いて「Open」ボタンを押下すると WebSocket のハンドシェイクとして 101 の HTTP ステータス・コードが返り、Response ヘッダーに Connection: Upgrade が含まれていることがわかります。



次に実際にメッセージを送信・受信する処理を実装します。

[5-2] CDI によるイベント操作

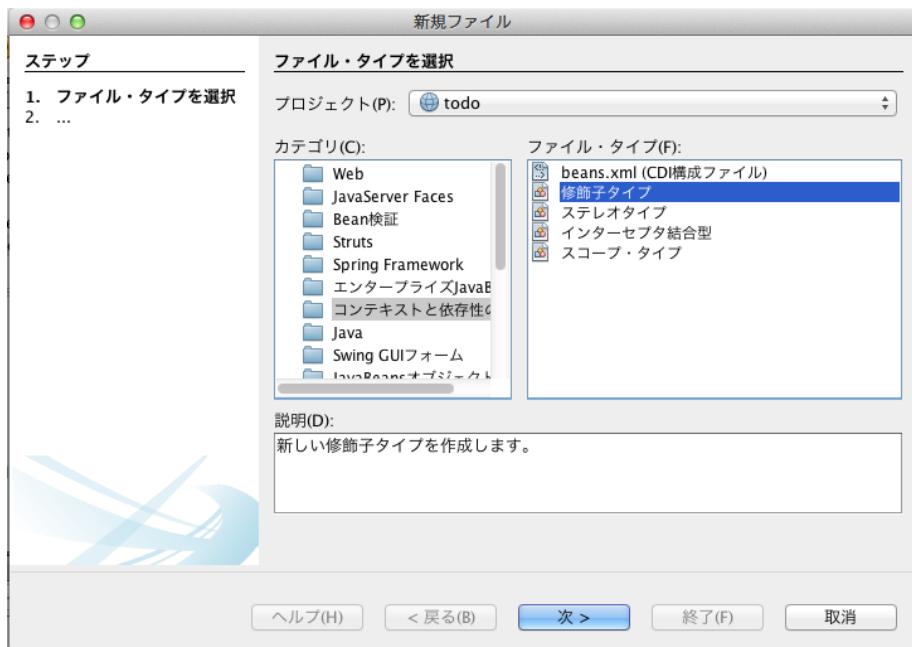
CDIはアプリケーション内のイベントの発行およびイベントの通知をサポートしており、オブザーバーパターンでイベント操作を行えます。具体的には javax.enterprise.event.Event クラスをインジェクションしてイベントを発行し、@Observes アノテーションでイベントを通知します。

■ TodoEvent 用修飾子の作成

CDI によって Todo に関するイベントをインジェクションするためにマーカーを作成します。

新規ファイルで「コンテキストと依存性の注入」→「修飾子タイプ」を選択

して「次 >」ボタンを押下します。



「New 修飾子タイプ」 ウィンドウにて下記の項目を入力し「終了(F)」ボタンを押下します。

入力項目	入力値
クラス名	TodoEvent
パッケージ	todo.domain.service.todo



ここでは、自動生成されたソース・コードに対して修正はおこないません。

```
package todo.domain.service.todo;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
```

```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface TodoEvent {
}
```

■ Event 用モデルを作成

モニタリング用のデータを保持するクラスを作成します。

「New Java クラス」 ウィンドウで下記の項目を入力し、「終了(F)」 ボタンを押下します。

入力項目	入力値
クラス名	TodoEventModel
パッケージ	todo.domain.service.todo



TodoEventModel クラスに対して下記を実装します。

```
package todo.domain.service.todo;

import todo.domain.model.Todo;

public class TodoEventModel {

    public static enum EventType {
        CREATE, UPDATE, DELETE
    }

    private final Todo todo;
    private final EventType type;

    public TodoEventModel(Todo todo, EventType type) {
        this.todo = todo;
        this.type = type;
    }

    public Todo getTodo() {
        return todo;
    }

    public EventType getType() {
        return type;
    }

    @Override
    public String toString() {
```

```
        return "TodoEventModel{" + "todo=" + todo + ", type=" + ty  
pe + "}";
    }
}
```

■ イベントの発行

TodoService に Event をインジェクションし、create, finish, delete 処理にイベントの発行処理を追加します。

```
package todo.domain.service.todo;

import java.util.Date;
import java.util.List;
import javax.ejb.Stateless;
import javax.enterprise.event.Event;
import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import todo.domain.common.exception.BusinessException;
import todo.domain.common.exception.ResourceNotFoundException;
import todo.domain.model.Todo;

@Stateless
public class TodoService {

    private static final long MAX_UNFINISHED_COUNT = 5;
    @PersistenceContext
    protected EntityManager entityManager;
    @Inject // (1)
```

```

@TodoEvent // (2)
protected Event<TodoEventModel> todoEvent;

public List<Todo> findAll() {
    TypedQuery<Todo> q = entityManager.createNamedQuery("Todo.findAll",
        Todo.class);
    return q.getResultList();
}

public Todo findOne(Integer todoId) {
    Todo todo = entityManager.find(Todo.class, todoId);
    if (todo == null) {
        throw new ResourceNotFoundException("[E404] The requested Todo is not found. (id=" + todoId + ")");
    }
    return todo;
}

public Todo create(Todo todo) {
    TypedQuery<Long> q = entityManager.createQuery("SELECT COUNT(x) FROM Todo x WHERE x.finished = :finished",
        Long.class)
        .setParameter("finished", false);
    long unfinishedCount = q.getSingleResult();
    if (unfinishedCount >= MAX_UNFINISHED_COUNT) {
        throw new BusinessException("[E001] The count of unfinished Todo must not be over "
            + MAX_UNFINISHED_COUNT + ".");
    }

    todo.setFinished(false);
    todo.setCreatedAt(new Date());
}

```

```

entityManager.persist(todo);
entityManager.flush(); // (3)

todoEvent.fire(new TodoEventModel(todo, TodoEventModel.EventType.CREATE)); // (4)

return todo;
}

public Todo finish(Integer todoId) {
    Todo todo = findOne(todoId);
    if (todo.isFinished()) {
        throw new BusinessException("[E002] The requested Todo
is already finished. (id="
        + todoId + ")");
    }
    todo.setFinished(true);
    entityManager.merge(todo);
    todoEvent.fire(new TodoEventModel(todo, TodoEventModel.EventType.UPDATE));
    return todo;
}

public void delete(Integer todoId) {
    Todo todo = findOne(todoId);
    entityManager.remove(todo);
    todoEvent.fire(new TodoEventModel(todo, TodoEventModel.EventType.DELETE));
}

```

項番	説明
(1)	@Inject アノテーションで Event オブジェクトをインジェクションする。
(2)	インジェクションする Event の種類を特定するための Qualifier アノテーションを指定する。 扱う Event が 1 種類の場合は指定しなくても良いが、複数扱う場合は明示的に指定する必要がある。アノテーションを作成しなくても @Named アノテーションで名前をつけることで区別することもできるが、文字列で区別するためタイプアンセーフである。そのため Qualifier アノテーションを作成することを推奨する。
(3)	イベントを発火する際に Todo の ID が設定された状態にする必要がある。そのため、明示的に flush メソッドを実行して DB に insert し ID を発行しておく。
(4)	イベントを発行する。

■ イベントの通知

EJB から発行されたイベントを購読するクラスを作成し、そのクラスに @Observes アノテーションを付加して Event を通知し、WebSocket クライアント・エンドポイントにメッセージを送信します。

「New Java クラス」ウィンドウで下記の項目を入力し、「終了(F)」ボタンを押下します。

入力項目	入力値
クラス名	TodoEventModel
パッケージ	todo.domain.service.todo

```

package todo.app.todo;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.websocket.Session;
import todo.domain.service.todo.TodoEvent;
import todo.domain.service.todo.TodoEventModel;

@ApplicationScoped // (1)
public class TodoEventObserver {

    private static final Logger logger = Logger.getLogger(TodoEventObserver.class.getName());

    public void onEventMessage(@Observes @TodoEvent TodoEventModel todoEventModel) { // (2)
        for (Session s : TodoWebSocketEndPoint.sessions) {
            try {
                s.getBasicRemote().sendText(todoEventModel.toString()); // (3)
            } catch (IOException ex) {
                logger.log(Level.SEVERE, null, ex);
            }
        }
    }
}

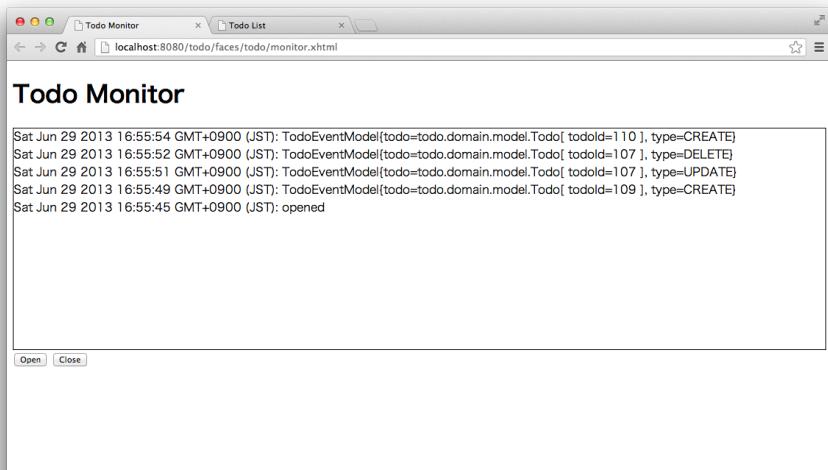
```

項目番号	説明
------	----

(1)	スコープを ApplicationScope にする。
(2)	Event を購読するために@Observes アノテーションを指定する。イベントを特定するために@TodoEvent アノテーションを指定する。イベントが発火されたタイミングでこのメソッドが実行され、fire に渡された TodoEventModel オブジェクトが引数に渡る。
(3)	全てのクライアントにイベントメッセージを同期的に送信する。非同期で送信する場合は getBasicRemote() メソッドの代わりに getAsyncRemote() を使用する。

Memo: 本クラスはクライアント同期処理や例外処理が不十分である点に注意すること。

モニタ画面で WebSocket を Open した後、JSF や JAX-RS 経由で Todo の作成、完了、削除を行うとブラウザをリロードすることなく、以下のようにモニタログが出力されます。



第6章 おわりに

本チュートリアルでは以下の内容を学習しました。

- NetBeans の基本操作
- NetBeans で JPA のエンティティ生成方法
- EJB による業務処理実装方法および EntityManager の使用方法
- JSF による web ページの作成方法
- JAX-RS による REST-API 実装方法
- WebSocket エンドポイントの作成方法
- CDI によるインジェクションおよびイベント操作

最終的なソースコードは以下で管理されています。

<https://github.com/making/javaee7-first-tutorial>

本ハンズオンを終了した後、より詳しく Java EE を学習するためには公式チュートリアルを参照していただくことを推奨します。。

<http://docs.oracle.com/javaee/7/tutorial/doc/home.htm>