



**TERASOLUNA Server/Client Framework
for .NET**

機能説明書

第 2.1.0.1 版

株式会社NTTデータ

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

- (1)本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。
- (2)本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。
- (3)本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献:TERASOLUNA Server/Client Framework for .NET 機能説明書」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
- (4)前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
- (5)NTT データの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
- (6)NTT データは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
- (7)NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTT データは一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

- Apache は、Apache Software Foundation の登録商標または商標です。
- Java は、米国 Sun Microsystems, Inc.の米国及びその他の国における登録商標または商標です。
- Microsoft、Visual Studio、Windows、.NET Framework は、米国 Microsoft Corp.の米国及びその他の国における登録商標または商標です。
- TERASOLUNA は、株式会社 NTT データの登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

本書は、以下のフレームワークに対応しています。

- TERASOLUNA Server Framework for .NET ver2.1 系
- TERASOLUNA Client Framework for .NET ver2.1 系

目次

● TERASOLUNA Server/Client Framework for .NET 共通機能

CM-01	メッセージ管理機能
CM-02	入力値検証機能
CM-03	ログ出力機能
CM-04	ビジネスロジック生成機能

● TERASOLUNA Server Framework for .NET 機能

WA-01	画面遷移管理機能
WA-02	画面遷移保証機能
WA-03	二重押下防止機能
WA-04	エラー画面遷移機能
FB-01	リクエストコントローラ機能
FB-02	ファイルアップロード機能
FB-03	ファイルダウンロード機能
WC-01	セッション管理機能
WC-02	SQL 文管理機能

● TERASOLUNA Client Framework for .NET 機能

FA-01	画面遷移機能
FA-02	拡張フォーム機能
FB-01	イベント処理機能
FB-02	データセット変換機能
FC-01	XML 通信機能
FC-02	ファイルアップロード機能
FC-03	ファイルダウンロード機能

CM-01 メッセージ管理機能

■ 概要

本機能では、アプリケーションで扱うメッセージに対し、統一的にアクセスする仕組みを提供する。

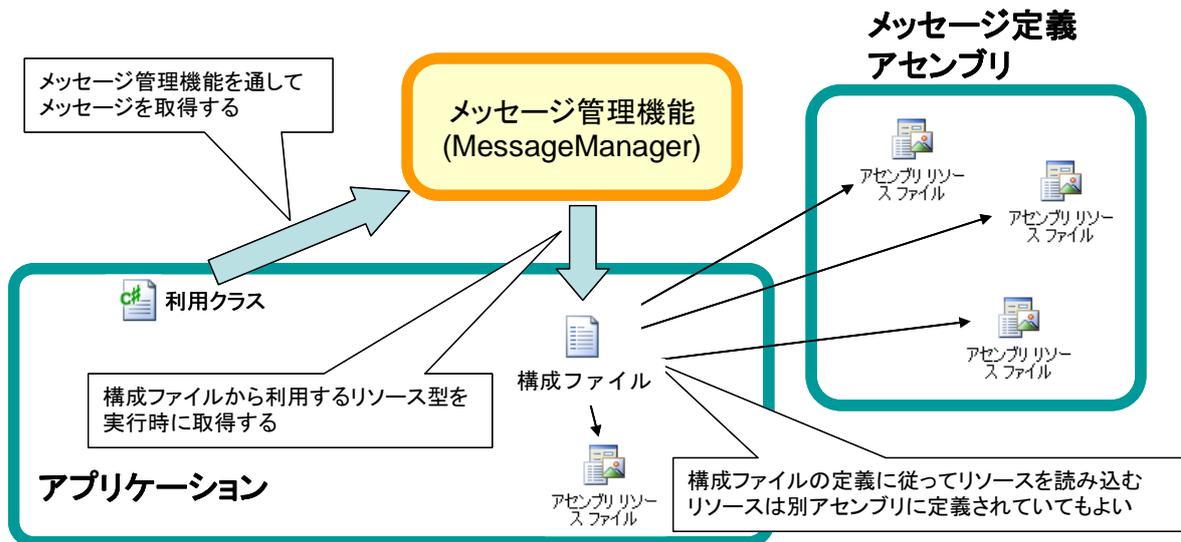


図 1 動作概念図

メッセージ管理機能を利用してメッセージを取得するには、MessageManagerクラスのメソッドを呼び出す。MessageManagerは、構成ファイル¹の定義に従ってアセンブリリソースファイルを読み込み、メッセージを返却する。メッセージを取得するクラスと、アセンブリリソースファイルは、異なるアセンブリでもよい。

¹ 構成ファイル: web.config や App.config のことを示す。

■ 使用方法

◆ 構成ファイル

本機能を有効にする場合、構成ファイルに設定を定義する。

表 1 構成ファイル

ノード	属性	必須	値
/configuration/appSettings/add			複数可
	key	○	一意の文字列 MessageResources.MyMessage01
	value	○	アセンブリ修飾名

● メッセージリソースの指定

利用するメッセージリソースの指定は構成ファイルの `appSettings` セクションに、”MessageResource.”をプレフィックスとする `key` 属性を持つ `add` 要素を追加することで行う。`add` 要素の `value` 属性には、利用するメッセージリソースの型をアセンブリ修飾名で記述する。メッセージリソースの型は、メッセージを利用するクラスと同一のアセンブリである必要はない。異なるアセンブリのメッセージを利用する場合には、実行時に読み込むことが可能な場所にアセンブリを配置する。

以下に構成ファイルの設定例を示す。

```
<appSettings>
  <add key="MessageResources.MyMessage00"
        value="CommonResources.CommonResource, CommonResources" />
  <add key="MessageResources.MyMessage01"
        value="MessageResources.Resource01, MessageResources" />
  <add key="MessageResources.MyMessage02"
        value="MessageResources.Resource02, MessageResources" />
</appSettings>
```

リスト 1 構成ファイルの例

● 複数リソースの指定

リスト 1 の例では、”MessageResources.”をプレフィックスとする `key` を持つ `add` 要素が 3 つ記述されている。MessageManager は、構成ファイルに”MessageResources.”をプレフィックスとする `key` を持つ `add` 要素が複数定義されている場合には、それらをすべて読み込み、メッセージの検索対象とする。その際、複数のリソース間のメッセージ検索順は、`key` 属性に指定された文字列をキーとした辞書順となることに注意する。リスト 1 の例では、まず `CommonResource` のメッセージが検索され、`CommonResource` に指定されたメッセージ ID が存在していない場合のみ `Resource01` のメッセージが検索される。

◆ 実装方法

● メッセージリソースの作成

本機能において利用できるリソースは、.NET Frameworkが提供するResourceManagerクラスが対応するリソース型及びリソースファイル²に準ずる。メッセージを格納するリソース型を作成する場合、Visual Studioのプルダウンメニューより、[プロジェクト - 新しい項目の追加]を選択し、「新しい項目の追加」ダイアログのテンプレートの一覧から「アセンブリ リソースファイル」を追加する。

以下に、Visual Studio からプロジェクトにリソースを追加するイメージを示す。

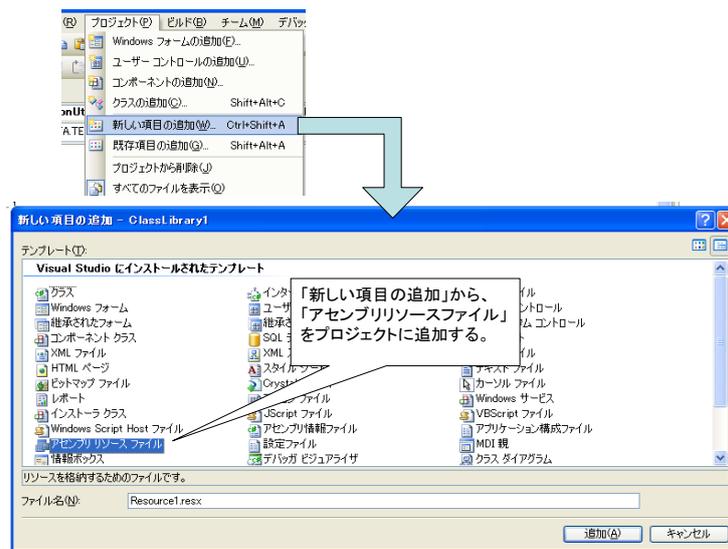


図 2 アセンブリリソースファイルの追加

リソースを作成するプロジェクトは、メッセージリソースを利用するクラスと同じアセンブリである必要はない。また、全てのメッセージリソースが一つのアセンブリに格納される必要はなく、コンパイル時に参照が解決される必要もない。実行時にはアプリケーションから参照可能な場所にメッセージリソースが定義されたアセンブリを配置する。

● メッセージリソースの編集

作成したリソース型をソリューションエクスプローラでダブルクリックするか、「ファイルを開くアプリケーションの選択」から「マネージリソースエディタ」を選択することで、リソースエディタが表示される。リソースエディタでリソースを編集すると、自動的に「[リソース名].Designers.cs」という名称でソースコードが自動生成され、リソースにアクセスするための型が作成される。リソースにアクセスするための型は、リソース名が型名となる。メッセージ管理機能では、この型を指定することで、メッセージを取得してくるリソースを追加することができる。

以下に、メッセージリソースの編集イメージを示す。

² リソースについては MSDN ドキュメント「.NET Framework 開発者ガイド - リソースのパッケージ化と配置」を参照のこと。

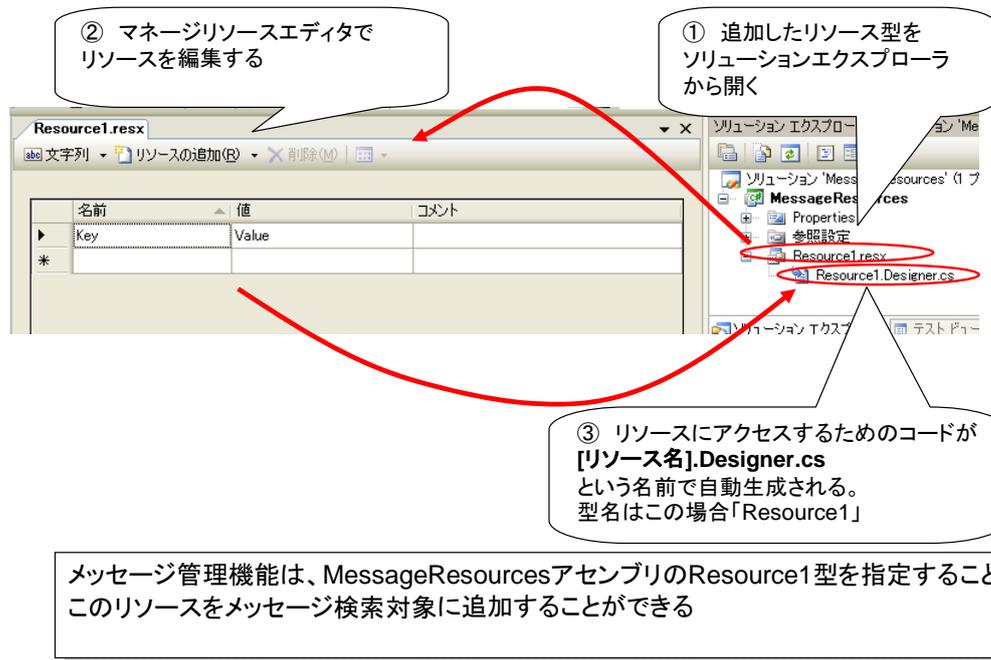


図 3 Visual Studio によるメッセージリソースの編集イメージ

なお、マネージリソースエディタでは、文字列リソースの他、アイコンや画像、ファイルなどもリソースに埋め込むことができるが、本機能によって利用できるリソースは文字列リソースのみとなり、イメージ、アイコン、オーディオ、ファイルなどには対応しない。リソースを指定する場合には、自動生成されたリソース型をアセンブリ修飾名³で指定する。

● メッセージの利用

メッセージ管理機能を用いてメッセージを取得する場合には、まずプロジェクトの参照設定に `TERASOLUNA.Fw.Common` アセンブリを追加する。ソースコードのメッセージを利用する場所では、以下のようにしてメッセージを取得するためのメッセージ ID を用いてメッセージを取得する。

```
string myMessage = ConfigurationManager.GetMessage("MessageKey");
```

リスト 2 メッセージ管理機能の利用の例

`MessageManager` クラスは静的なメソッド `GetMessage(string ,params object[])` と静的なプロパティ `Culture` を公開する。

`MessageManager` は、`GetMessage` メソッドの第一引数として渡したメッセージ ID を用いて、管理するリソースの中にメッセージ ID に対応するメッセージが定義されているか検索する。存在すればその文字列を返し、存在しなければ `null` 参照を返す。複数のリソースで同一のメッセージ

³ アセンブリ修飾名については MSDN ドキュメント「.NET Framework 開発者ガイド - 完全修飾型名の指定」を参照のこと。

ID に合致するメッセージが定義されている場合、最初に見つかったメッセージを返す。複数のリソースが存在する場合の検索順に関しては `MessageManager` が初期化される段階で決定される。

`Culture` プロパティを用いて、検索するメッセージのカルチャを指定できる。日本語のメッセージの他、利用される環境の言語環境に応じたメッセージを用意することが可能である。

- 書式項目の利用

`MessageManager` の `GetMessage` メソッドには、可変長引数として、複数のオブジェクト `args` を渡すことができる。`args` に一つ以上のオブジェクトが指定された場合、`GetMessage` メソッドはリソースから取得したメッセージを `args` で指定されたオブジェクトを用いてフォーマットする。このとき、メッセージには与えられた `args` のオブジェクトに対応したインデックス付プレースホルダ(書式項目)が適切な数定義されていないといけない。

以下に、メッセージを書式設定する例を示す。リスト 5 はリソースに定義されているメッセージであるとする。

```
ARG_NULL_EXCEPTION = 引数 "{0}" が null 参照です.
```

リスト 3 書式項目を利用したメッセージ定義例

メッセージ利用側では、たとえば書式項目 {0} に対応する文字列 "user_name" を `GetMessage` メソッドの第二引数に与えることで、「引数 "user_name" が null 参照です。」というメッセージを取得することができる。

```
string myMessage =  
    MessageManager.GetMessage("ARG_NULL_EXCEPTION", "user_name");
```

リスト 4 置換文字列の利用例

置換文字列を利用したメッセージの整形については、`.NET Framework`の提供する複合書式設定機能⁴を参照のこと。なお、書式項目を利用したフォーマットを行う場合、`args`に指定したオブジェクトの数以上のインデックスを持つプレースホルダがメッセージに存在すると例外が発生するため、注意すること。

たとえば、以下のような定義がされている場合に、`GetMessage` メソッドの `args` として 4 つ以上のオブジェクトを指定しない場合、例外となる。

```
ARG_EXCEPTION = 引数 "{3}" は不正です。 /* args が 3 つ以下だと例外! */
```

リスト 5 危険なメッセージ定義例

◆ 構成クラス

表 2 構成クラス一覧

⁴ 複合書式設定については MSDN ドキュメント「`.NET Framework` 開発者ガイド -複合書式設定」を参照のこと。

項番	クラス名	説明
1	MessageManager	メッセージ管理機能を提供するユーティリティクラス

■ 拡張ポイント

独自のメッセージ管理クラスをフレームワークに沿って利用する場合、MessageManager を継承したクラスを作成する。その際、Init メソッドをオーバーライドして ResourceManagerList の初期化ロジックを変更することで、利用するメッセージリソースの読み込み方法や、検索順序をカスタマイズすることができる。

- 利用するメッセージ管理クラス実装の変更

メッセージ管理機能を提供する MessageManager は内部に MessageManager クラスまたは MessageManager 派生クラスのインスタンスを一度だけ生成して利用する。このとき、構成ファイルの appSettings セクションにメッセージ管理クラスの型のアセンブリ修飾名を指定することで、利用する実装を差し替えることができる。キーは”MessageManagerTypeName” とする。指定を省略した場合、MessageManager クラスのインスタンスが利用される。

以下に、メッセージ管理クラスの差し替えを設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MessageManagerTypeName"
        value="MyNamespace.MessageManagerEx, MyNamespace"/>
  </appSettings>
</configuration>
```

リスト 6 メッセージ管理クラス指定の例

■ 関連機能

特になし

CM-02 入力値検証機能

■ 概要

本機能は、入力値検証設定ファイルに定義した検証ルールに従って、入力値検証を行う機能を提供する。検証対象は、データセットが保持するテーブルの各カラムであり、カラムに格納されている値が検証ルールと一致するかをチェックする。必須入力チェック、半角カナ文字列チェック、日付形式チェック、int 型範囲チェックなど、業務アプリケーションに必要な各種検証ルールを提供している。詳細は後述の『入力値検証ルール解説』を参照のこと。

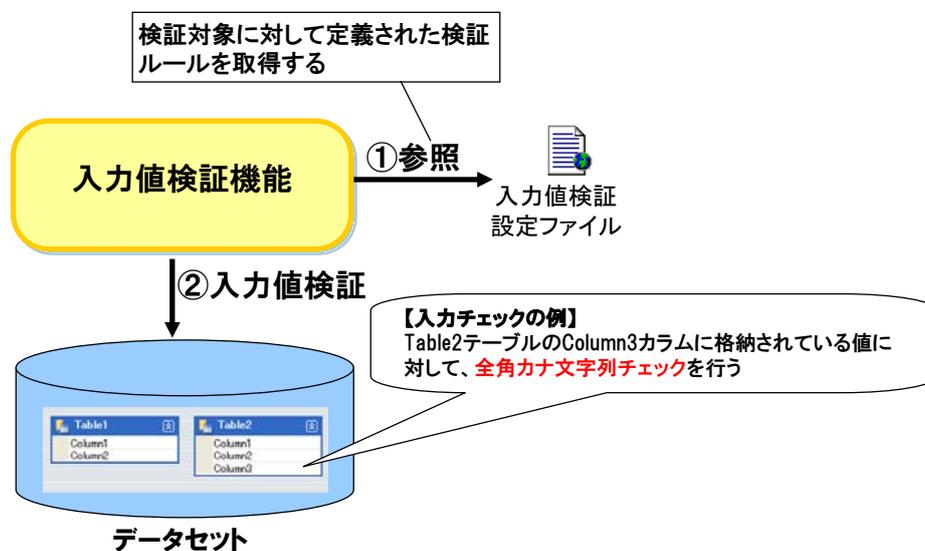


図 1 動作概念図

入力値検証機能は、入力値検証設定ファイルを参照し、検証対象（データセットが保持するテーブルの各カラム）に対して定義された検証ルールを取得する。そして、取得した検証ルールに従って、検証対象に対して入力値検証を行う。

データセットは複数のテーブルを持ち、また、各テーブルには複数のカラムが存在する。そのため、入力値検証設定ファイルには、「どのデータセットの、どのテーブルの、どのカラムに対して、どの検証ルールを適用するか」を定義する。

1つのカラムに対して、複数の検証ルールを適用することが可能である。例えば、あるカラムの入力値が必須かつ全角文字列でなければならない場合、「必須入力チェック」と「全角文字列チェック」を組み合わせることで実現できる。

なお、本機能は主にイベント処理機能とリクエストコントローラ機能から呼び出される。

■ 使用方法

◆ 入力値検証設定ファイル

検証対象に対してどの検証ルールを適用するかを、入力値検証設定ファイルに定義する。

表 1 入力値検証設定ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name	○	構成要素名。 固定値、以下を指定する。 validation
/configuration/validation/type	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値、以下を指定する。 Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
	assemblyName	○	検証対象のアセンブリ名。
/configuration/validation/type/ruleset	name	○	検証対象のデータ行クラスの型名。 データセット内のテーブルは、データセットクラスの内部クラスとして定義されており、このクラスのことを「データ行クラス」という。 データ行クラスの型名は、データセットクラス名とデータ行クラス名を”+”で連結して次のように記述する。 “データセットクラス名+テーブル行クラス名”
	name	○	ルールセット名。 /configuration/validation/type 要素で指定したテーブルに対して、検証設定をルールセットという単位で複数定義することができる。ルールセット名は ruleset 要素の中で一意でなければならない。 イベント処理機能やリクエストコントローラ機能では、ルールセット名の既定値と

			して”Default”を利用する。したがって、それらの機能でルールセット名の設定を省略して既定値を利用する場合は、入力値検証設定ファイルの ruleset 要素 name 属性に”Default”と設定すること。
/configuration/validation/ruleset/properties/property			複数可
	name	○	検証対象カラム名。 同一テーブルの複数のカラムに対して入力値検証を行う場合は、property 要素をカラムの数だけ追加する。
/configuration/validation/ruleset/properties/property/validator			複数可
	適用する検証ルールに応じて、必要な属性を設定する。 複数の検証ルールを組み合わせたい場合、validator 要素を検証ルールの数だけ追加する。		

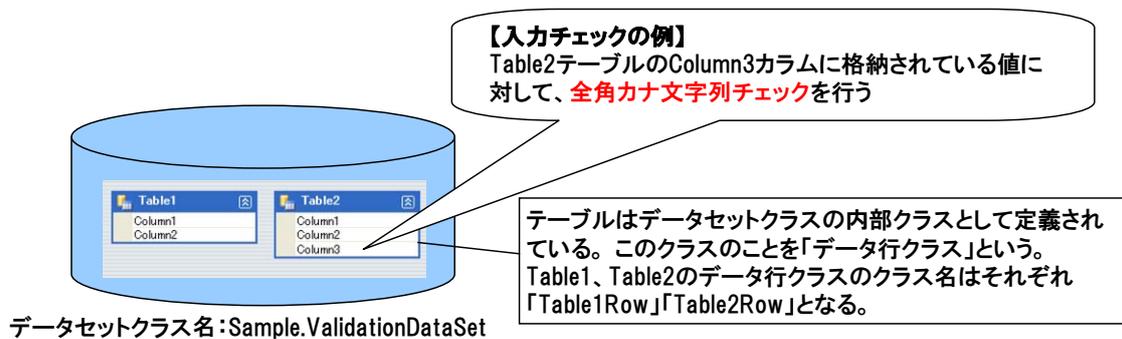


図 2 入力チェックの例

図 2に示すような入力チェック (Table2 テーブルのColumn3 カラムに格納されている値に対する全角カナ文字列チェック)を行う場合の、入力値検証設定ファイルの設定例を以下に示す。

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="validation"
      type="Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
  </configSections>
  <validation>
    <type assemblyName="Test"
      name="Sample.ValidationDataSet+Table2Row"
      <ruleset name="customRuleSet">
        <properties>
          <property name="Column3">
            <validator negated="false"
              tag="Column1"
              name="ZenkakuKanaStringValidator"
              type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuKanaStringValidator, TERASOLUNA.Fw.Common" />
          </property>
        </properties>
      </ruleset>
    </type>
  </validation>
</configuration>

```

データセットクラス名+データ行クラス名

ルールセット名

カラム名

リスト 1 入力値検証設定ファイルの設定例

validator 要素には、適用する検証ルールに応じて、必要な属性を設定する。各検証ルールに共通の属性を以下に示す。

表 2 validator 要素に設定する検証ルール共通の属性

項番	属性名	必須	説明	デフォルト値
1	negated		true を設定すると検証ルールを反転してチェックする。true もしくは false を設定する。	false
2	messageTemplate		検証エラーメッセージのテンプレートとして使用する文字列。	各検証ルールに定義されている既定のメッセージ
3	messageTemplateResourceName		検証エラーメッセージのテンプレートをメッセージリソースから読み込む際の名前。	なし
4	messageTemplateResourceType		検証エラーメッセージのテンプレートをメッセージリソースから読み込む際のリソースの型。	なし
5	tag		メッセージテンプレートのプレースホルダ{2}に渡される文字列。	なし

			検証対象項目の論理名を設定する。	
6	type	○	適用する検証ルールに対応した検証クラスの型を、完全修飾名で設定する。	なし
7	name	○	検証ルールの名前。任意の名前を設定することができる。	なし

共通の属性以外に、検証ルール固有の属性が存在する。例えば範囲チェックを行うルールならば上限値と下限値を指定するための属性がある。固有の属性の詳細については、後述の『入力値検証ルール解説』を参照のこと。

➤ 検証エラーメッセージのテンプレートの設定

`messageTemplate` 属性に設定した文字列、またはメッセージリソースから読み込んだ文字列は、検証エラーが発生した際に検証エラーメッセージを生成するのに用いられるテンプレートである。プレースホルダ{0}~{2}には固定で以下の値が格納される。

表 3 メッセージテンプレートに渡される文字列

プレースホルダ	説明
{0}	検証対象のカラムに格納されている値の文字列表現
{1}	カラム名(検証対象のカラム名)
{2}	validator 要素の tag 属性に設定した文字列

{3}以降のプレースホルダについては、検証ルールによって利用形態が異なる。詳細については後述の『入力値検証ルール解説』を参照のこと。

➤ 適用する検証ルールに対応した検証クラスの設定

`type` 属性には、適用する検証ルールに対応した検証クラスの型をアセンブリ修飾名で設定する。検証ルールは次の 3 つの提供元がある。

- Validation Application Block¹
- Validation Application Block Extensions²
- TERASOLUNAフレームワーク³

¹ Validation AB が提供する検証クラスの名前空間は“Microsoft.Practices.EnterpriseLibrary.Validation.Validators”となり、アセンブリ名は“Microsoft.Practices.EnterpriseLibrary.Validation”となる。

² Validation Application Block Extensions が提供する検証クラスの名前空間は、“EntLibContrib.Validation.Validators”となり、アセンブリは“EntLibContrib.Validation”となる。

³ TERASOLUNA フレームワークが提供する検証クラスの名前空間は、“TERASOLUNA.Fw.Common.Validation.Validators”となり、アセンブリ名は“TERASOLUNA.Fw.Common”となる。

以下に、各提供元が提供する検証ルールとその検証クラスを示す。

表 4 Validation Application Block が提供する検証ルール

項番	検証ルール	検証クラス	概要
1	byte 型チェック	TypeConversionValidator	検証対象が byte (Byte)型に変換可能か検証する。
2	short 型チェック		検証対象が short(Int16)型に変換可能か検証する。
3	int 型チェック		検証対象が int(Int32)型に変換可能か検証する。
4	long 型チェック		検証対象が long(Int64)型に変換可能か検証する。
5	decimal 型チェック		検証対象が decimal (Decimal)型に変換可能か検証する。
6	float 型チェック		検証対象が float(Single)型に変換可能か検証する。
7	double 型チェック		検証対象が double (Double)型に変換可能か検証する。

表 5 Validation Application Block Extensions が提供する検証ルール

項番	検証ルール	検証クラス	概要
1	要素数チェック	CollectionCountValidator	検証対象がコレクション(List や配列など)であった場合、指定した要素数の範囲であるかを検証する。

表 6 TERASOLUNA フレームワークが提供する検証ルール

項番	検証ルール	検証クラス	概要
1	正規表現一致チェック	RegexValidatorEx	正規表現パターンを指定して、検証対象文字列がパターンとマッチするかを検証する。
2	必須入力チェック	RequiredValidator	検証対象が null またはホワイトスペース(半角空白、全角空白、改行、タブ文字)でないか検証する。
3	数値文字列チェック	NumericStringValidator	検証対象が半角数値文字のみで構成されているか検証する。
4	半角英数大文字列チェック	CapAlphaNumericStringValidator	検証対象が大文字の半角英数文字のみで構成されているか検証する。

5	半角英数文字列チェック	AlphaNumericStringValidator	検証対象が半角英数文字のみで構成されているか検証する。
6	半角文字列チェック	HankakuStringValidator	検証対象が半角文字のみで構成されているか検証する。
7	半角カナ文字列チェック	HankakuKanaStringValidator	検証対象が半角カナ文字のみで構成されているか検証する。
8	全角カナ文字列チェック	ZenkakuKanaStringValidator	検証対象が全角カナ文字のみで構成されているか検証する。
9	全角文字列チェック	ZenkakuStringValidator	検証対象が全角文字のみで構成されているか検証する。
10	URL 形式チェック	UrlValidator	検証対象が URL 形式の文字列であるか検証する。
11	日付形式チェック	DateTimeFormatValidator	検証対象が日付形式の文字列であるか検証する。
12	int 型範囲チェック	IntRangeValidator	検証対象が int 型に変換可能であり、指定した範囲であるか検証する。
13	decimal 型範囲チェック	DecimalRangeValidator	検証対象が decimal 型に変換可能であり、指定した範囲であるか検証する。
14	float 型範囲チェック	FloatRangeValidator	検証対象が float 型に変換可能であり、指定した範囲であるか検証する。
15	double 型範囲チェック	DoubleRangeValidator	検証対象が double 型に変換可能であり、指定した範囲であるか検証する。
16	日付型範囲チェック	DateTimeRangeValidatorEx	検証対象が dateTime 型に変換可能であり、指定した範囲の日時であるかを検証する。
17	byte 列長範囲チェック	ByteRangeValidator	検証対象の文字列を指定したエンコーディングでバイト列に展開した際のバイト長が指定した範囲であるか検証する。
18	文字列長チェック	StringLengthRangeValidator	文字列が指定した文字数の範囲であるかを検証する。
19	必須文字列チェック	ContainsCharactersValidatorEx	指定した文字列を含んでいるかどうかを検証する。
20	数値チェック	NumberValidator	検証対象が数値に変換可能であり、整数部・小数部がそれぞれ指定した桁数であるか検証する。

◆ 実装方法

本機能は、主に「イベント処理機能」と「リクエストコントローラ機能」から呼び出される。本機能を利用する際に必要な共通設定についての説明と、各機能からの呼び出し例について説明する。

- 本機能を利用する際に必要な共通設定
本機能を利用するには、プロジェクトの参照設定に次の dll を追加する。
 - EntLibContrib.Validation
 - Microsoft.Practices.EnterpriseLibrary.Common
 - Microsoft.Practices.EnterpriseLibrary.Validation
 - TERASOLUNA.Fw.Common

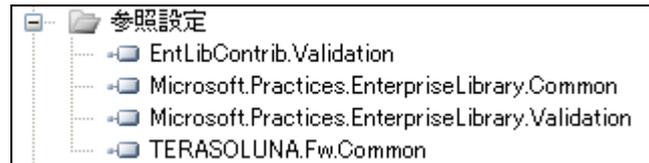


図 3 参照設定に追加した dll

- イベント処理機能からの呼び出し例
EventController コンポーネントの ValidationFilePath プロパティに入力値検証設定ファイルのパスを指定することで、入力値検証機能を利用できる。イベント処理機能から呼び出す際の実装の流れを以下に示す。

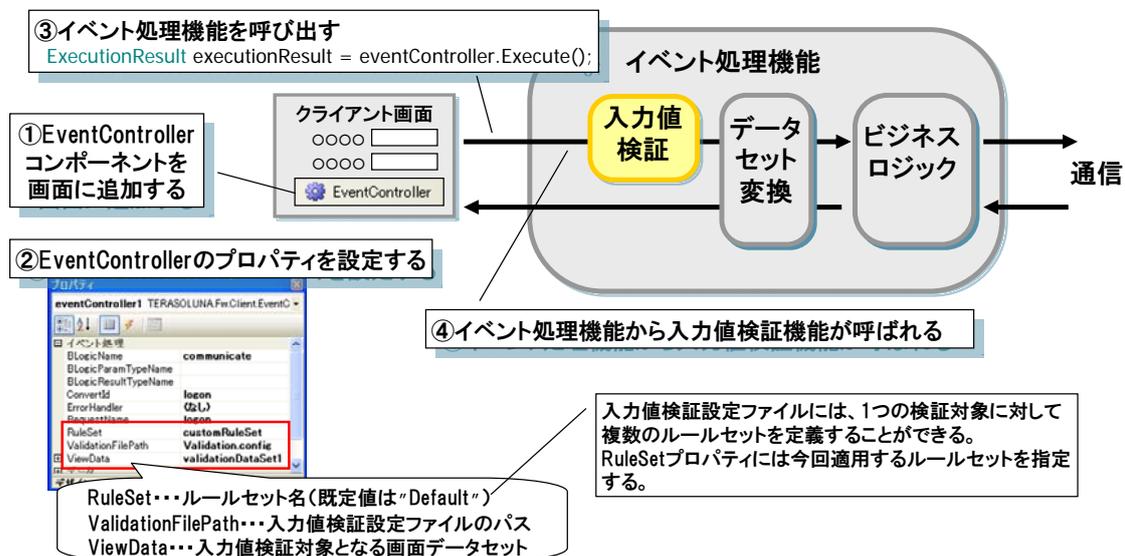
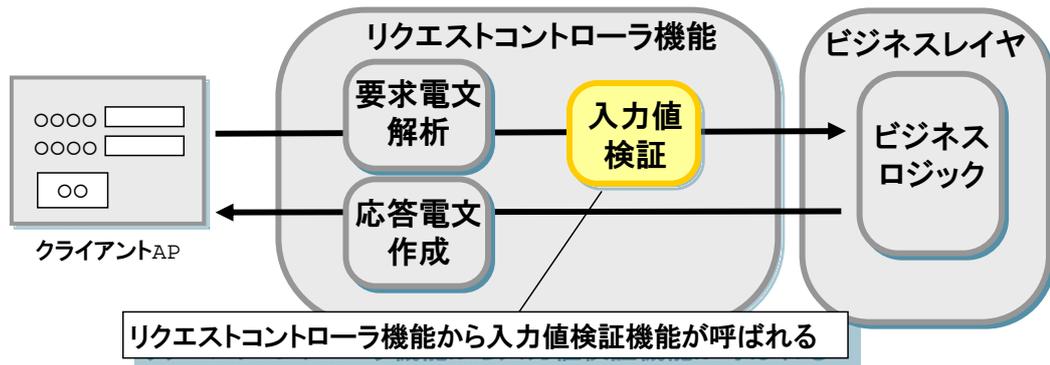


図 4 イベント処理機能から呼び出す際の実装の流れ

- リクエストコントローラ機能からの呼び出し例
ビジネスロジッククラスのクラス属性に、ValidationFilePath を定義することで、ビジネスロジックの入力値に対する検証を行うことができる。リクエストコントローラ機能から呼び出す際の実装例を以下に示す。



ビジネスロジッククラスの実装例

```
[ControllerInfo(RequestType=RequestTypeNames.NORMAL, InputDataSetType=typeof(ValidationDataSet))]
[ValidationInfo(ValidationFilePath="Validation.config", RuleSet="customRuleSet")]
public class BLogic01 : ILogic {}
```

InputDataSetType...入力値検証対象のデータセットの型

ValidationFilePath...入力値検証設定ファイルのパス
RuleSet...ルールセット名(既定値は"Default")

図 5 リクエストコントローラ機能から呼び出す際の実装例

- 各機能から呼び出す際の設定のイメージ

イベント処理機能／リクエストコントローラ機能から呼び出す際の設定と、入力値検証設定ファイル／入力値検証対象のデータセットとの関係のイメージを以下に示す。

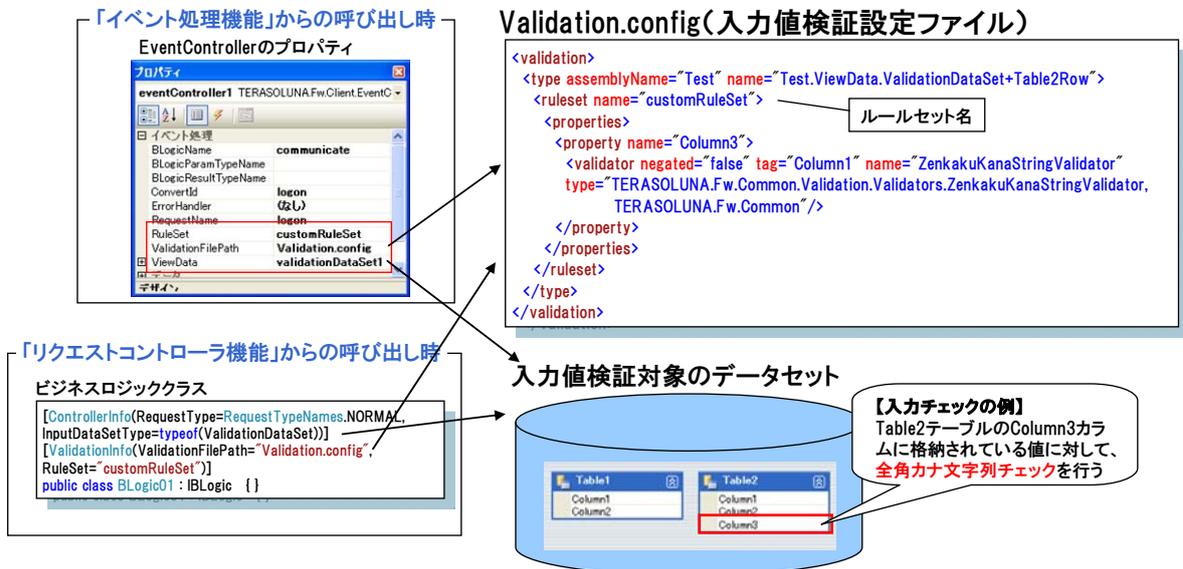


図 6 各機能から呼び出す際の設定のイメージ

■ 入力値検証ルール解説

◆ Validation Application Blockが提供する検証ルール解説

Validation Application Block(以下、Validation AB)が提供する入力値検証ルールの機能と利用方法について解説する。

Validation AB が提供する検証クラスの名前空間は“Microsoft.Practices.EnterpriseLibrary.Validation.Validators”となり、アセンブリ名は“Microsoft.Practices.EnterpriseLibrary.Validation”となる。

- byte 型チェック

検証対象値が byte 値(System.Byte)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 7 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 8 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。 byte 型チェックでは、 System.Byteを指定する	なし

表 9 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Byte)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Byte"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral,
  PublicKeyToken=b03f5f7f11d50a3a"
  name="Byte Type Conversion Validator" />
```

リスト 2 設定例

- short 型チェック

検証対象値が short 値(System.Int16)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 10 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 11 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。 short 型チェックでは、 System.Int16を指定する	なし

表 12 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int16)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int16"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Short Type Conversion Validator" />
```

リスト 3 設定例

- int 型チェック

検証対象値が int 値(System.Int32)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 13 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 14 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。int 型チェックでは、System.Int32 を指定する	なし

表 15 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int32)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int32"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Int Type Conversion Validator" />
```

リスト 4 設定例

- long 型チェック
検証対象値が long 値(System.Int64)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 16 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 17 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。 long 型チェックでは、 System.Int64 を指定する	なし

表 18 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int64)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int64"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Long Type Conversion Validator" />
```

リスト 5 設定例

- decimal 型チェック

検証対象値が decimal 値(System.Decimal)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 19 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 20 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。 decimal 型チェックでは、 System.Decimal を指定する	なし

表 21 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{}	変換対象の型名(System.Decimal)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Decimal"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Decimal Type Conversion Validator" />
```

リスト 6 設定例

- float 型チェック

検証対象値が float 値(System.Single)に変換可能かを検証する。Validation AB が提供するTypeConversionValidator を利用する。

表 22 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 23 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。 float 型チェックでは、 System.Singleを指定する	なし

表 24 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Single)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Single"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Float Type Conversion Validator" />
```

リスト 7 設定例

- double 型チェック

検証対象値が double 値(System.Double)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 25 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 26 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。 double 型チェックでは、 System.Double を指定する	なし

表 27 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Double)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Double"
  negated="false" messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  name="Double Type Conversion Validator" />
```

リスト 8 設定例

◆ Validation Application Block Extensionsが提供する入力値検証 ルールの解説

Enterprise Library Contrib の Validation Application Block Extensions (以下、Validation AB Ex) が提供する入力値検証ルールを説明する。

Validation AB Ex が提供する検証クラスの名前空間は、“EntLibContrib.Validation.Validators”となり、アセンブリは“EntLibContrib.Validation”となる。

● 要素数チェック

検証対象が配列や List などコレクション(ICollection)型であり、要素数が指定した範囲に含まれているか検証する。Validation AB Extensions で提供する CollectionCountValidator を利用する。なお、要素数は ICollection インターフェイスで定義された Count プロパティの値である。配列の場合は Length プロパティの値となる。

文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 28 構成クラス

項番	クラス名	説明
1	CollectionCountValidator	要素数チェックを行う検証クラス
2	CollectionCountValidatorData	要素数チェックを行う CollectionCountValidator を生成するクラス

表 29 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である要素数の範囲の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である要素数の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 30 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="1" lowerBoundType="Exclusive" upperBound="256"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="EntLibContrib.Validation.Validators.CollectionCountValidator,
  EntLibContrib.Validation"
  name="Collection Count Validator" />
```

リスト 9 設定例

◆ フレームワークが提供する入力値検証ルールの解説

フレームワークが提供する入力値検証ルールの機能と利用方法について解説する。

フレームワークが提供する検証クラスの名前空間は

“TERASOLUNA.Fw.Common.Validation.Validators”となり、アセンブリ名は

“TERASOLUNA.Fw.Common”となる。

- 正規表現一致チェック

検証対象値が指定した正規表現文字列にマッチするかどうかを検証する。正規表現パターンとともに、.NET Framework で提供される正規表現エンジンの各種オプションを利用することができる。フレームワークが提供する `RegexValidatorEx` を利用する。

以下に、検証成功となるパターンを示す。

- 通常時

検証対象となる文字列が指定されたパターンと正規表現オプションによって表される正規表現にマッチする場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象となる文字列が指定されたパターンと正規表現オプションによって表される正規表現にマッチしない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 31 構成クラス

項番	クラス名	説明
1	<code>RegexValidatorEx</code>	正規表現による入力値検証を行う検証クラス
2	<code>RegexValidatorExData</code>	設定情報から <code>RegexValidatorEx</code> を生成、初期化するクラス

表 32 固有設定項目

項番	属性名	必須	解説	デフォルト値
1	pattern	○	正規表現のパターン	なし
2	options	-	正規表現オプション(RegexOptions 列挙体) を指定する。列挙体の値は以下の値の一つまたは複数の組み合わせとなる <ul style="list-style-type: none"> •None •IgnoreCase •Multiline •ExplicitCapture •Compiled •Singleline •IgnorePatternWhiteSpace •RightToLeft 複数の値を指定する場合、コンマで区切って記述する	None
4	patternResourceName	-	正規表現パターンを読み込む際に利用するリソース名	なし
5	patternResourceTypeName	-	正規表現のパターンを読み込むリソースの型名	なし

表 33 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	正規表現のパターン
2	{4}	利用された正規表現オプション

以下に設定例を示す。

```
<validator pattern="^TEL [0-9][0-9][0-9][0-9]-?[0-9][0-9]-?[0-9][0-9][0-9][0-9]$"
options="Compiled, IgnorePatternWhiteSpace"
patternResourceName="" patternResourceType="" messageTemplate=""
messageTemplateResourceName="" messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.RegexValidatorEx,
TERASOLUNA.Fw.Common"
name="Regex Tel Number Validator" />
```

リスト 10 設定例

- 必須入力チェック

検証対象値が未入力でないかを検証する。フレームワークで提供する `RequiredValidator` を利用する。

以下に、検証成功となるパターンを示す。

- 通常時

検証対象が `null` ではなく、検証対象値の文字列表現からホワイトスペース(半角空白、全角空白、改行、タブ文字)を取り除いた文字列の長さが 0 より大きい場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象が `null` であるか、または検証対象値の文字列表現からホワイトスペース(半角空白、全角空白、改行、タブ文字)を取り除いた文字列の長さが 0 の場合。

フレームワークで提供する入力値検証ルールは原則として `null` または空文字列に対して検証成功を返す。`negated` が `true` である場合も同様となる。`null` または空文字列を許容しない場合には、必須入力チェックを併用する。

表 34 構成クラス

項番	クラス名	説明
1	<code>RequiredValidator</code>	必須入力チェックを行う検証クラス
2	<code>RequiredValidatorData</code>	必須入力チェックを行う <code>RequiredValidator</code> を生成するクラス

必須入力チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""  
messageTemplateResourceType="" tag=""  
type="TERASOLUNA.Fw.Common.Validation.Validators.RequiredValidator,  
TERASOLUNA.Fw.Common"  
name="Required Validator" />
```

リスト 11 設定例

- 数値文字列チェック

検証対象の文字列表現が、半角数字 (0～9)のみで構成されているか検証する。フレームワークで提供する `NumericStringValidator` を利用する。

以下に、検証成功となるパターンを示す。

- 通常時

検証対象の文字列表現が半角数字(0～9)のみで構成されている場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象の文字列表現に半角数字(0～9)以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 35 構成クラス

項番	クラス名	説明
1	<code>NumericStringValidator</code>	数値文字列チェックを行う検証クラス
2	<code>NumericStringValidatorData</code>	数値文字列チェックを行う <code>NumericStringValidator</code> を生成するクラス

数値文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""  
messageTemplateResourceType="" tag=""  
type="TERASOLUNA.Fw.Common.Validation.Validators.NumericStringValidator,  
TERASOLUNA.Fw.Common"  
name="Numeric String Validator" />
```

リスト 12 設定例

- 半角英数大文字列チェック

検証対象の文字列表現が、半角英数大文字のみで構成されているか検証する。フレームワークで提供する `CapAlphaNumericStringValidator` を利用する。半角英数大文字とは、半角数字(0～9)と半角大文字の英字(A～Z)を併せた文字集合である。

検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現が半角英数大文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現に半角英数大文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 36 構成クラス

項番	クラス名	説明
1	<code>CapAlphaNumericStringValidator</code>	半角英数大文字列チェックを行う検証クラス
2	<code>CapAlphaNumericStringValidatorData</code>	半角英数大文字列チェックを行う <code>CapAlphaNumericStringValidator</code> を生成するクラス

半角英数大文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""
messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.
    CapAlphaNumericStringValidator, TERASOLUNA.Fw.Common"
name="Cap Alpha Numeric String Validator" />
```

リスト 13 設定例

- 半角英数文字列チェック

検証対象の文字列表現が、半角英数文字のみで構成されているか検証する。フレームワークで提供する `AlphaNumericStringValidator` を利用する。半角英数文字とは、半角数字(0~9)と半角の英字(a~z、A~Z)を併せた文字集合である。

以下に、検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現が半角英数文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現に半角英数文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 37 構成クラス

項番	クラス名	説明
1	<code>AlphaNumericStringValidator</code>	半角英数文字列チェックを行う検証クラス
2	<code>AlphaNumericStringValidatorData</code>	半角英数文字列チェックを行う <code>AlphaNumericStringValidator</code> を生成するクラス

半角英数文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""  
messageTemplateResourceType="" tag=""  
type="TERASOLUNA.Fw.Common.Validation.Validators.AlphaNumericStringValidator,  
TERASOLUNA.Fw.Common"  
name="Alpha Numeric String Validator" />
```

リスト 14 設定例

- 半角文字列チェック
検証対象の文字列表現が、半角文字列のみで構成されているか検証する。フレームワ

ークで提供する `HankakuStringValidator` を利用する。半角文字とは、“\ ¢ £ §”
 ー°±‘¶×÷”を除く unicode の 0 から 255 番目まで(拡張 ASCII コードの範囲)の文字に
 半角カナを加えた文字集合である。

以下に、検証成功となるパターンを示す。

- 通常時
 検証対象の文字列表現が半角文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
 検証対象の文字列表現に半角文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は
 実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合に
 は、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 38 構成クラス

項番	クラス名	説明
1	<code>HankakuStringValidator</code>	半角文字列チェックを行う検証クラス
2	<code>HankakuStringValidatorData</code>	半角文字列チェックを行う <code>HankakuStringValidator</code> を生成するクラス

半角文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""
messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.HankakuStringValidator,
TERASOLUNA.Fw.Common"
name="Hankaku String Validator" />
```

リスト 15 設定例

- 半角カナ文字列チェック

検証対象の文字列表現が、半角カナのみで構成されているか検証する。フレームワークで提供する `HankakuKanaStringValidator` を利用する。半角カナ文字とは、以下に示す文字集合である。"アイエオアイエオカキクケコサシスセソタチツテトナニヌネノヒフヘホマミメモヤユヨヤョラリルロワヅンパー、。」「"

以下に、検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現が半角カナ文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現に半角カナ文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 39 構成クラス

項番	クラス名	説明
1	<code>HankakuKanaStringValidator</code>	半角カナ文字列チェックを行う検証クラス
2	<code>HankakuKanaStringValidatorData</code>	半角カナ文字列チェックを行う <code>HankakuKanaStringValidator</code> を生成するクラス

半角カナ文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""
messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.HankakuKanaStringValidator,
TERASOLUNA.Fw.Common"
name="HankakuKana String Validator" />
```

リスト 16 設定例

- 全角カナ文字列チェック

検証対象の文字列表現が、全角カナ文字列のみで構成されているか検証する。フレームワークで提供する `ZenkakuKanaStringValidator` を利用する。全角カナ文字とは、以下に示す文字集合である。"アイウヴエオアイウエオカキクケコカケガギグゲゴサシスセソザジズゼゾタチツテトダヂヅデドナニヌネノハヒフヘホバビブベボパピプペポマミムメモヤユヨャョラリルレロワウヰエヲッシー"

以下に、検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現が全角カナ文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現に全角カナ文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 40 構成クラス

項番	クラス名	説明
1	<code>ZenkakuKanaStringValidator</code>	全角カナ文字列チェックを行う <code>Validator</code>
2	<code>ZenkakuKanaStringValidatorData</code>	全角カナ文字列チェックを行う <code>ZenkakuKanaStringValidator</code> を生成するクラス

全角カナ文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""
messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuKanaStringValidator,
TERASOLUNA.Fw.Common"
name="Zenkaku Kana String Validator" />
```

リスト 17 設定例

- 全角文字列チェック

検証対象の文字列表現が、全角文字のみで構成されているか検証する。フレームワークで提供する `ZenkakuStringValidator` を利用する。全角文字列とは、半角文字(半角文字列チェックの定義に準じる)ではない文字のみで構成された文字列のことである。

以下に、検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現が全角文字のみで構成されている場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現に全角文字以外の文字が含まれている場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 41 構成クラス

項番	クラス名	説明
1	<code>ZenkakuStringValidator</code>	全角文字列チェックを行う検証クラス
2	<code>ZenkakuStringValidatorData</code>	全角文字列チェックを行う <code>ZenkakuStringValidator</code> を生成するクラス

全角文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""  
messageTemplateResourceType="" tag=""  
type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuStringValidator,  
TERASOLUNA.Fw.Common"  
name="ZenkakuString String Validator" />
```

リスト 18 設定例

- URL 形式文字列チェック

検証対象の文字列表現が、URL 形式の文字列であるかを検証する。フレームワークで提供する `UrlValidator` を利用する。URL の形式は、以下に示す。

<プロトコル>://<ホスト名>:<ポート番号>/<パス文字列>

表 42 URL 形式文字列の構成要素

項番	名称	説明
1	プロトコル	http or https のみ対応。小文字と大文字は区別しない
2	ホスト名	“/”及び”：“以外の任意の文字列である
3	ポート番号	任意桁数の数値。省略可能であり、省略時は”:"を記述しないこと
4	パス	任意の文字列

検証成功となるパターンを示す。

- 通常時

検証対象の文字列表現が URL 形式である場合。

- 反転時(negated 属性が true の場合)

検証対象の文字列表現が URL 形式ではない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 43 構成クラス

項番	クラス名	説明
1	<code>UrlValidator</code>	URL 形式文字列チェックを行う検証クラス
2	<code>UrlValidatorData</code>	URL 形式文字列チェックを行う <code>UrlValidator</code> を生成するクラス

URL 形式文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateName=""
messageTemplateResourceType="" tag=""
type="TERASOLUNA.Fw.Common.Validation.Validators.UrlValidator,
TERASOLUNA.Fw.Common"
name="Url Format String Validator" />
```

リスト 19 設定例

- 日付形式チェック

検証対象の文字列表現が、指定した形式の日付文字列であるかを検証する。フレームワークで提供する `DateTimeFormatValidator` を利用する。日付文字列の形式は、`DateTimeFormatInfo` で利用可能な形式指定文字列の組み合わせで指定する。たとえば、“yyyy/MM/dd”を形式として指定する場合、“2005/08/12”などの日付文字列には検証に成功するが、“2005-08-12”や“2005 年 08 月 12 日”などの文字列には検証に失敗する。

検証成功となるパターンを示す。

- 通常時

検証対象の文字列表現が、指定した日付書式文字列に従って `DateTime` 型に変換可能である場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象の文字列表現が、指定した日付書式文字列に従って `DateTime` 型に変換可能でない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 44 構成クラス

項番	クラス名	説明
1	<code>DateTimeFormatValidator</code>	日付形式チェックを行う検証クラス
2	<code>DateTimeFormatValidatorData</code>	日付形式チェックを行う <code>DateTimeFormatValidator</code> を生成するクラス

表 45 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	<code>dateTimeFormat</code>	○	検証対象である日付形式文字列を指定する	なし

表 46 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	日付形式文字列。(<code>dateTimeFormat</code>)

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
messageTemplateResourceType="" tag=""
dateTimeFormat="yyyy/MM/dd"
type="TERASOLUNA.Fw.Common.Validation.Validators.DateTimeFormatValidator,
TERASOLUNA.Fw"
name="Date Time Format String Validator" />
```

リスト 20 設定

● int 型範囲チェック

検証対象が `int(System.Int32)`型の数値または `int` 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する `IntRangeValidator` を利用する。

検証成功となるパターンを示す。

➤ 通常時

検証対象が `int` 型の値または `int` 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。

➤ 反転時(`negated` 属性が `true` の場合)

検証対象が `int` 型の値または `int` 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 47 構成クラス

項番	クラス名	説明
1	<code>IntRangeValidator</code>	<code>int</code> 型範囲チェックを行う検証クラス
2	<code>IntRangeValidatorData</code>	<code>int</code> 型範囲チェックを行う <code>IntRangeValidator</code> を生成するクラス

表 48 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である int 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である int 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 49 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

なお、IntRangeValidator に限らず、Validation AB を利用する範囲チェックの仕組みでは、lowerBound(最小値)、lowerBoundType(指定した最小値の扱い)、upperBound(最大値)、upperBoundType(指定した最大値の扱い)の四つの属性を利用して範囲を指定する。共通して以下の制約があるので注意すること。

- lowerBound <= upperBound である必要がある
- lowerBoundType が Inclusive または Exclusive と指定されているとき、lowerBound は必ず指定する必要がある(lowerBound のデフォルト値がある場合を除く)
- upperBoundType が Inclusive または Exclusive と指定されているとき、upperBound は必ず指定する必要がある(upperBound のデフォルト値がある場合を除く)
- lowerBoundType と upperBoundType の両方が Ignore であってはならない
- lowerBound、upperBound に指定する文字列は検証しようとするデータの型に変換できなければならない

検証しようとするデータとは検証対象の値とは異なる。IntRangeValidator であれば、数値であるから、int 型であるが、後述する StringLengthRangeValidator では文字列を対象とするものの検証される値は文字列長であるから int 型である。negated を true と指定した場合、有効となる範囲が反転する。このとき、Inclusive /Exclusive の意味も逆転することに注意する。

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Ignore" upperBound="20"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.IntRangeValidator,
    TERASOLUNA.Fw.Common"
  name="Int Range Validator" />
```

リスト 21 設定例

- decimal 型範囲チェック

検証対象が decimal(System.Decimal)型の数値または decimal 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する DecimalRangeValidator を利用する。

検証成功となるパターンを示す。

- 通常時

検証対象が decimal 型の値または decimal 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。

- 反転時(negated 属性が true の場合)

検証対象が decimal 型の値または decimal 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 50 構成クラス

項番	クラス名	説明
1	DecimalRangeValidator	decimal 型範囲チェックを行う検証クラス
2	DecimalRangeValidatorData	decimal 型範囲チェックを行う DecimalRangeValidator を生成するクラス

表 51 固有設定項目

- float 型範囲チェック

検証対象が float(System.Single)型の数値または float 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する FloatRangeValidator を利用する。

検証成功となるパターンを示す。

- 通常時

検証対象が float 型の値または float 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。

- 反転時(negated 属性が true の場合)

検証対象が float 型の値または float 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 53 構成クラス

項番	クラス名	説明
1	FloatRangeValidator	float 型範囲チェックを行う検証クラス
2	FloatRangeValidatorData	float 型範囲チェックを行う FloatRangeValidator を生成するクラス

表 54 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である float 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である float 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 55 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="-0.1" lowerBoundType="Exclusive" upperBound="0.01"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.FloatRangeValidator,
  TERASOLUNA.Fw.Common"
  name="Float Range Validator" />
```

リスト 23 設定例

- double 型範囲チェック

検証対象が `double(System.Double)`型の数値または `double` 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する `DoubleRangeValidator` を利用する。

検証成功となるパターンを示す。

- 通常時

検証対象が `double` 型の値または `double` 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象が `double` 型の値または `double` 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 56 構成クラス

項番	クラス名	説明
1	<code>DoubleRangeValidator</code>	<code>double</code> 型範囲チェックを行う検証クラス
2	<code>DoubleRangeValidatorData</code>	<code>double</code> 型範囲チェックを行う <code>DoubleRangeValidator</code> を生成するクラス

表 57 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	<code>lowerBound</code>	-	検証対象である <code>double</code> 型の値の最小値を指定する	0
2	<code>lowerBoundType</code>	-	<code>lowerBound</code> で指定した最小値の値の扱いを指定する <code>RangeBoundaryType</code> 列挙体	Ignore
3	<code>upperBound</code>	-	検証対象である <code>double</code> 型の値の最大値を指定する	0
4	<code>upperBoundType</code>	-	<code>upperBound</code> で指定した最小値の値の扱いを指定する <code>RangeBoundaryType</code> 列挙体	Inclusive

表 58 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Exclusive" upperBound="20"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.DoubleRangeValidator,
  TERASOLUNA.Fw.Common"
  name="Double Range Validator" />
```

リスト 24 設定例

- 日付型範囲チェック

検証対象が日付型(`System.DateTime`)型の数値または日付型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する `DateTimeRangeValidatorEx` を利用する。`Validation AB` で提供される `DateTimeRangeValidator` とは継承関係はなく、名前空間も異なるが、記述上の可読性を考慮してクラス名には"Ex"サフィックスを付与している。

日付型範囲チェックでは、日付型に変換可能な文字列に対する範囲チェックも可能であるが、記述形式を指定することはできない。`DateTimeConverter` が対応した形式であれば検証対象となる。日付文字列の形式を指定するには、フレームワークで提供する日付文字列フォーマットチェック(`DateTimeFormatValidator`)と併用する。

検証成功となるパターンを示す。

- 通常時

検証対象が日付型の値または日付型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象が日付型の値または日付型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 59 構成クラス

項番	クラス名	説明
1	<code>DateTimeRangeValidator</code>	日付型範囲チェックを行う検証クラス
2	<code>DateTimeRangeValidatorData</code>	日付型範囲チェックを行う <code>DateTimeRangeValidator</code> を生成するクラス

表 60 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である DateTime 型の値の最小値を指定する	0 (1年1月1日 00:00)
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である DateTime 型の値の最大値を指定する	0 (1年1月1日 00:00)
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 61 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="2007/04/01" lowerBoundType="Exclusive"
  upperBound="2008/04/01" upperBoundType="Exclusive"
  negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.DateTimeRangeValidator,
  TERASOLUNA.Fw.Common"
  name="Float Range Validator" />
```

リスト 25 設定例

- byte 列長範囲チェック

検証対象の文字列を指定したエンコード形式でバイト列にデコードした際、バイト列長が指定した範囲であるかどうかを検証する。フレームワークで提供する `ByteRangeValidator` を利用する。

検証成功となるパターンを示す。

- 通常時

検証対象が文字列であり、かつ、指定したエンコーディングでデコードしたバイト列の長さが指定した最小値と最大値の間に含まれる場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象が文字列であり、かつ、指定したエンコーディングでデコードしたバイト列の長さが指定した最小値と最大値の間に含まれない場合。

検証対象が `null` である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 62 構成クラス

項番	クラス名	説明
1	<code>ByteRangeValidator</code>	byte 列長範囲チェックを行う検証クラス
2	<code>ByteRangeValidatorData</code>	byte 列長範囲チェックを行う <code>ByteRangeValidator</code> を生成するクラス

表 63 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	<code>maxByteLength</code>	-	検証対象であるバイト長の最大値を <code>int</code> 型の値で指定する。範囲には指定した値自体が含まれる。	2,147,483,647
2	<code>minByteLength</code>	-	検証対象であるバイト長の最小値を <code>int</code> 型の値で指定する。範囲には指定した値自体が含まれる	0

3	encodingName	-	検証対象文字列をバイト列にデコードするために利用するエンコーディング名。コードページでの指定はできません。利用可能なエンコーディング名は System.Text.Encoding クラスでサポートされているエンコーディングを参照のこと。	utf-8
---	--------------	---	---	-------

表 64 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	バイト列長の最大値。(maxLength)
2	{4}	バイト列長の最小値。(minLength)
3	{5}	エンコーディング名

以下に設定ファイルへの設定例を示す。

```
<validator minLength="5" maxLength="10" encodingName="iso-2022-jp"
  negated="false" messageTemplate=""
  messageTemplateName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ByteRangeValidator,
  TERASOLUNA.Fw.Common"
  name="Byte Range Validator" />
```

リスト 26 設定例

- 文字列長チェック

検証対象の文字列表現の文字列長が指定した範囲であるかどうかを検証する。フレームワークで提供する `StringLengthRangeValidator` を利用する。

最小値を `Ignore`(無視)指定することで最大文字数制限として、最大値を `Ignore` 指定することで最小文字数制限として機能する。また、最大値と最小値として同じ値を指定し、いずれも `Inclusive`(値を含む)として指定すれば、固定文字数チェックとして機能する。

検証成功となるパターンを示す。

- 通常時
検証対象の文字列表現の文字列長が最小値と最大値の間に含まれる場合。
- 反転時(`negated` 属性が `true` の場合)
検証対象の文字列表現の文字列長が最小値と最大値の間に含まれない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 65 構成クラス

項番	クラス名	説明
1	<code>StringLengthRangeValidator</code>	文字列の長さが指定した範囲内であるかどうかを検証する検証クラス
2	<code>StringLengthRanheValidatorData</code>	設定情報から <code>StringLengthRangeValidator</code> を生成、初期化するクラス

表 66 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	<code>lowerBound</code>	-	検証対象である文字列長範囲の最小値を指定する	0
2	<code>lowerBoundType</code>	-	<code>lowerBound</code> で指定した最小値の値の扱いを指定する <code>RangeBoundaryType</code> 列挙体。以下の値のいずれかとなる。 <ul style="list-style-type: none"> • <code>Ignore</code> (値を無視する) • <code>Inclusive</code> (値を含む) • <code>Exclusive</code> (値を含まない) 	<code>Ignore</code>
3	<code>upperBound</code>	-	検証対象である文字列長範囲の最大値を指定する	0

4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive
---	----------------	---	---	-----------

表 67 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Inclusive" upperBound="20"
  upperBoundType="Inclusive" negated="false"
  messageTemplate="値は10文字以上、20文字以下で指定してください。"
  messageTemplateName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.StringLengthRangeValidator,
  TERASOLUNA.Fw.Common"
  name="String Length Range Validator" />
```

リスト 27 設定例

- 必須文字列チェック(禁止文字列チェック)

検証対象文字列が、指定した文字を含んでいるかどうかの検証を行う。フレームワークで提供する `ContainsCharactersValidatorEx` を利用する。

`negated` 属性を省略または `false` と指定した場合、`characterSet` 属性で指定した文字が含まれていない場合に検証エラーとなる。`negated` 属性を `true` と指定することで、指定した文字が含まれる場合に検証エラーとする禁止文字列チェックとして利用することができる。

検証成功となるパターンを示す。

- 通常時

検証対象が文字列であり、かつ、`containsCharacter` で「Any」が指定されている場合には、`characterSet` で指定された文字のうちいずれかを含んでいれば検証成功となる。`containsCharacter` で「All」が指定されている場合には、`characterSet` で指定された文字を全て含んでいれば検証成功となる。

- 反転時(`negated` 属性が `true` の場合)

検証対象が文字列であり、かつ、`containsCharacter` で「Any」が指定されている場合には、`characterSet` で指定された文字が一つも含まれていなければ検証成功となる。`containsCharacter` で「All」が指定されている場合には、`characterSet` で指定された文字を全て含んでいる場合以外に検証成功となる。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 68 構成クラス

項番	クラス名	説明
1	<code>ContainsCharactersValidatorEx</code>	指定した文字が含まれているかどうかを検証する検証クラス
2	<code>ContainsCharactersValidatorExData</code>	設定情報から <code>ContainsCharactersValidator</code> を生成、初期化するクラス

表 69 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	characterSet	○	検証対象の文字列中で存在をチェックする文字。(複数指定する場合、コンマなどで区切らず、"AB."と続けて記述する。)	なし
2	containsCharacter	○	characterSet で指定した文字の存在をチェックする方式を指定する ContainsCharacters 列挙体。 以下の値のいずれかとなる。 ・Any (いずれか一つを含む) ・All (全ての文字を含む)	Any

表 70 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	チェック対象の文字。(characterSet)
2	{4}	文字のチェック方式。(containsCharacter)

以下に設定ファイルへの設定例を示す。

```
<validator characterSet="qwerty" containsCharacter="All" negated="true"
  messageTemplate="" messageTemplateName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ContainsCharactersValidatorEx,
  TERASOLUNA.Fw.Common"
  name="Contains Characters Ex Validator" />
```

リスト 28 設定例

- 数値チェック

検証対象の文字列表現が、整数部と小数部がそれぞれ指定した桁数である数値の形式であるかを検証する。フレームワークで提供する `NumberValidator` を利用する。桁数のチェックには、指定した値と等しいか調べる一致チェックと、指定した以内の範囲に含まれるか調べる範囲チェックを利用することができる。

検証成功となるパターンを示す。

- 通常時

検証対象の文字列表現が整数または小数の数値形式であり、桁数がそれぞれ指定した値・範囲に含まれる場合。

- 反転時(`negated` 属性が `true` の場合)

検証対象の文字列表現が整数または小数の数値形式でない場合、または、桁数がそれぞれ指定した値・範囲に含まれない場合。

検証対象が `null` または空文字列である場合、`negated` の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。`null` または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(`RequiredValidator`)と併用する。

表 71 構成クラス

項番	クラス名	説明
1	<code>NumberValidator</code>	数値チェックを行う検証クラス
2	<code>NumberValidator</code>	数値チェックを行う <code>NumberValidator</code> を生成するクラス

表 72 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	<code>integerLength</code>	○	数値文字列の整数部の桁数を <code>int</code> 型の自然数で指定する。最小値は 1	なし
2	<code>scale</code>	-	数値文字列の小数部の桁数を <code>int</code> 型の値で指定する。最小値は 0 であり、0 を指定した際には整数チェックとなる	0

3	isAccordedInteger	-	整数部の桁数一致チェックを行うかどうかを指定する。 True であれば、integerLength で指定した桁数と一致しているかどうかを検証する。false であれば、指定した桁数以下であるか検証する	false
4	isAccordedScale	-	isAccordedInteger と同様に小数部の桁数一致チェックを行うかどうかを指定する	false

表 73 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	指定した整数部の桁数。(integerLength)
2	{4}	指定した小数部の桁数(scale)

以下に設定ファイルへの設定例を示す。

```
<validator integerLength="5" scale="4"
  isAccordedInteger="false" isAccordedScale="true"
  negated="false" messageTemplate=""
  messageTemplateName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.NumberValidator,
  TERASOLUNA.Fw.Common"
  name="Number Validator" />
```

リスト 29 設定例

■ 内部構成

◆ 構成クラス

表 74 構成クラス一覧

項番	クラス名	説明
1	AlphaNumericStringValidator	半角英数文字列チェックを行う検証クラス
2	AlphaNumericStringValidatorData	AlphaNumericStringValidator の生成に利用するクラス
3	ByteRangeValidator	エンコーディングを指定して文字列のバイト長範囲チェックを行う検証クラス
4	ByteRangeValidatorData	ByteRangeValidator の生成に利用するクラス
5	CapAlphaNumericStringValidator	半角英数大文字列チェックを行う検証クラス
6	CapAlphaNumericStringValidatorData	CapAlphaNumericStringValidator の生成に利用するクラス
7	CaseCheckUtil	文字種・パターンチェックを行う Validator で利用するユーティリティクラス
8	ContainsCharactersValidatorEx	指定した文字列を含んでいるかどうかを検証する
9	ContainsCharactersValidatorExData	ContainsCharactersValidatorEx の生成に利用するクラス
10	DateTimeFormatValidator	日付形式文字列チェックを行う検証クラス
11	DateTimeFormatValidatorData	DateTimeFormatValidator の生成に利用するクラス
12	DateTimeRangeValidatorEx	日付の範囲チェックを行う検証クラス
13	DateTimeRangeValidatorExData	DateTimeRangeValidatorEx の生成に利用するクラス
14	DecimalRangeValidator	Decimal 値の範囲チェックを行う検証クラス
15	DecimalRangeValidatorData	DecimalRangeValidator の生成に利用するクラス
16	DoubleRangeValidator	Double 値の範囲チェックを行う検証クラス
17	DoubleRangeValidatorData	DoubleRangeValidator の生成に利用するクラス
18	FloatRangeValidator	Float 値の範囲チェックを行う検証クラス

19	FloatRangeValidatorData	FloatRangeValidator の生成に利用するクラス
20	HankakuKanaStringValidator	半角カナ文字列チェックを行う検証クラス
21	HankakuKanaStringValidatorData	HankakuKanaStringValidator の生成に利用するクラス
22	HankakuStringValidator	半角文字列チェックを行う検証クラス
23	HankakuStringValidatorData	HankakuStringValidator の生成に利用するクラス
24	IntRangeValidator	Int 値の範囲チェックを行う検証クラス
25	IntRangeValidatorData	IntRangeValidator の生成に利用するクラス
26	NumberRangeValidator	数値型の範囲チェックを行う Validator の基底クラス
27	NumberValidator	数値の桁数チェックを行う検証クラス
28	NumberValidatorData	NumberValidator の生成に利用するクラス
29	NumericStringValidator	数値文字列チェックを行う検証クラス
30	NumericStringValidatorData	NumericStringValidator の生成に利用するクラス
31	RequiredValidator	必須チェックを行う検証クラス
32	RequiredValidatorData	RequiredValidator の生成に利用するクラス
33	RegexValidatorEx	正規表現チェックを行う検証クラス
34	RegexValidatorExDaya	RegexValidatorEx の生成に利用するクラス
35	StringLengthRangeValidator	文字列長チェックを行う検証クラス
36	StringLengthRangeValidatorData	StringLengthRangeValidator の生成に利用するクラス
37	TypeRangeValidator	型指定した範囲チェックを行う Validator の基底クラス
38	UrlValidator	URL 形式文字列チェックを行う検証クラス
39	UrlValidatorData	UrlValidator の生成に利用するクラス
40	VabValidator	Validation Application Block の仕組みを利用して入力値検証を行う IValidator 実装クラス。
41	ZenkakuKanaStringValidator	全角カナ文字列チェックを行う検証クラス

42	ZenkakuKanaStringValidatorData	ZenkakuKanaStringValidator の生成に利用するクラス
43	ZenkakuStringValidator	全角文字列チェックを行う検証クラス
44	ZenkakuStringValidatorData	ZenkakuStringValidator の生成に利用するクラス

■ 拡張ポイント

◆ 検証ルールを拡張する方法

一つの検証ルールは役割の異なる二種類のクラスによって実現される。実際に入力値検証を実施する `Validator` 検証クラスと、設定情報の解析と `Validator` インスタンスの初期化を行う `ValidatorData` クラスである。下表にそれぞれの違いを示す。

表 75 入力値検証クラスの構成

項番	名称	説明
1	<code>ValidatorData</code> クラス	<code>Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidatorData</code> を継承した、 <code>Validator</code> を生成・初期化するためのクラス。設定情報から <code>Validator</code> を生成するための <code>DoCreateValidator</code> メソッドを実装する。
2	<code>Validator</code> クラス	<code>Microsoft.Practices.EnterpriseLibrary.Validation.Validator</code> を継承した入力値検証の実装クラス 検証を行う <code>DoValidate</code> メソッドを実装する。 <code>ConfigurationElementType</code> 属性をクラスに付与し、使用する <code>ValidatorData</code> クラスの型を指定する。

`Validation AB` は、設定情報から `Validator` の型を取得するが、その型のインスタンスを直接生成しない。利用される `Validator` が生成される過程を以下に示す。

- (1) 設定情報から、利用する `Validator` の型 (`TypeA`) を取得する。
- (2) 取得した型 `TypeA` の `ConfigurationElementType` 属性より、`Validator` の生成・初期化を行う `ValidatorData` クラスの型 (`TypeB`) を取得し、型 `TypeB` のインスタンスを作成する。
- (3) 型 `TypeB` の `CreateValidator` メソッドを用いて、実際に利用する `Validator` のインスタンスを取得する。

◆ 実装例

数字文字列のみを許可するシンプルな `Validator` の実装例を示す。`Validation AB` の提供する正規表現チェックを拡張して実装し、対応する `ValidatorData` クラスを作成する。

```
namespace SampleValidator
{
    [ConfigurationElementType(typeof(NumberStringValidatorData))]
    public class NumberStringValidator : RegexValidator
    {
        /// <summary>
        /// 正規表現を利用するValidatorのコンストラクタでパターンを固定
        /// </summary>
        public NumberStringValidator(string messageTemplate, bool negated)
            : base(@"[0-9]*", messageTemplate, negated)
        {
        }
    }
}
```

リスト 30 Validator クラス

```
namespace SampleValidatorData
{
    public class NumberStringValidatorData : ValueValidatorData
    {
        public NumberStringValidatorData()
        {
        }
        protected override Validator DoCreateValidator(Type targetType)
        {
            // NumberStringValidatorを生成
            return new NumberStringValidator(MessageTemplate, Negated);
        }
    }
}
```

リスト 31 ValidatorData クラス

```
<validator negated="false"
  messageTemplate="{2}には数字のみを入力してください。{0}は数字以外を含んでいるか、
  文字列ではありません。"
  messageTemplateName="" messageTemplateResourceType=""
  tag="数値項目"
  type="SampleValidatorData.NumberStringValidator, SampleValidatorData"
  name="number validator test" />
```

リスト 32 設定ファイル

例では固有の設定項目を持たないため、`SampleValidatorData` は単に `SampleValidator` のインスタンスを生成する処理のみを行う。生成される `SampleValidator` は数字文字列を表す固定の正規表現を用いて入力値検証を行う、`RegexValidator` のサブクラスである。`RegexValidator` はコンストラクタの第一引数で正規表現パターン文字列を受け取るため、`SampleValidator` では、親クラスのコンストラクタに固定の正規表現パターンを渡すことで機能を実現している。

固有の設定項目を利用して `Validator` を初期化するには、`validator` 要素に属性として指定された設定情報を取得し、`Validator` に設定する。`ValidatorData` クラスのインデクサに属性名を指定することで、`validator` 要素に設定された属性の値を取得することができる。

入力値検証のロジックを独自に定義するには、`Validator` を実装する際、`DoValidate` メソッドをオーバーライドする。その際、`ValueValidator` を親クラスに持つ場合には、`negated` が `true` と指定されている場合の検証条件の反転を考慮すること。

◆ ファクトリクラスの変更

`Validator` のインスタンスを生成するために利用するファクトリクラスは、必要に応じて変更することができる。特に指定がないとき、ファクトリクラスは `ValidatorFactory` が用いられる。

ファクトリクラスを変更する場合は、構成ファイル⁴の `appSettings` 要素に、`"ValidatorFactoryTypeName"`をキーとして、利用するファクトリクラスのアセンブリ修飾名を記述する。なお、本機能説明書においては標準のファクトリクラスである `ValidatorFactory`を用いた場合の例のみを示す。

以下に構成ファイルの例を示す。

⁴ 構成ファイル: `App.config` や `WebConfig` のことを示す

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ValidatorFactoryTypeName"
          value="MyNamespace.MyValidatorFactory, MyNamespace"/>
  </appSettings>
</configuration>
```

リスト 33 ファクトリ型指定の例(構成ファイル)

◆ 入力値検証実行クラスの変更

標準のファクトリを利用する場合、特に指定がないとき入力値検証実行クラス (IValidator 実装クラス)は VabValidator が用いられる。

利用する IValidator 実装クラスを変更する場合は、構成ファイルの appSettings 要素に、"ValidatorTypeName"をキーとしてアセンブリ修飾名を記述する。この設定をもとに、入力値検証生成機能(ValidatorFactory クラス)によって入力値検証実行クラスのインスタンスが生成される。

以下に構成ファイルの例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ValidatorTypeName"
          value="TERASOLUNA.Fw.Common.Validation.VabValidator,
                TERASOLUNA.Fw.Common" />
  </appSettings>
</configuration>
```

リスト 34 入力値検証実行クラス指定の例(構成ファイル)

■ 関連機能

- FB-01 イベント処理機能
- WB-01 リクエストコントローラ機能

CM-03 ログ出力機能

■ 概要

本機能では、アプリケーションで統一的にログを出力する仕組みを提供する。個別のロギングパッケージ¹をラップするため、異なるロギングパッケージへ移行した場合も、コードの修正を最小限に抑えることができる。また、Jakarta Commons Logging²と同等のログレベルを備えているため、サーバ側がJava、クライアント側が.NETという組み合わせの開発においても、アプリケーション全体で統一的なログポリシーを定めることができる。

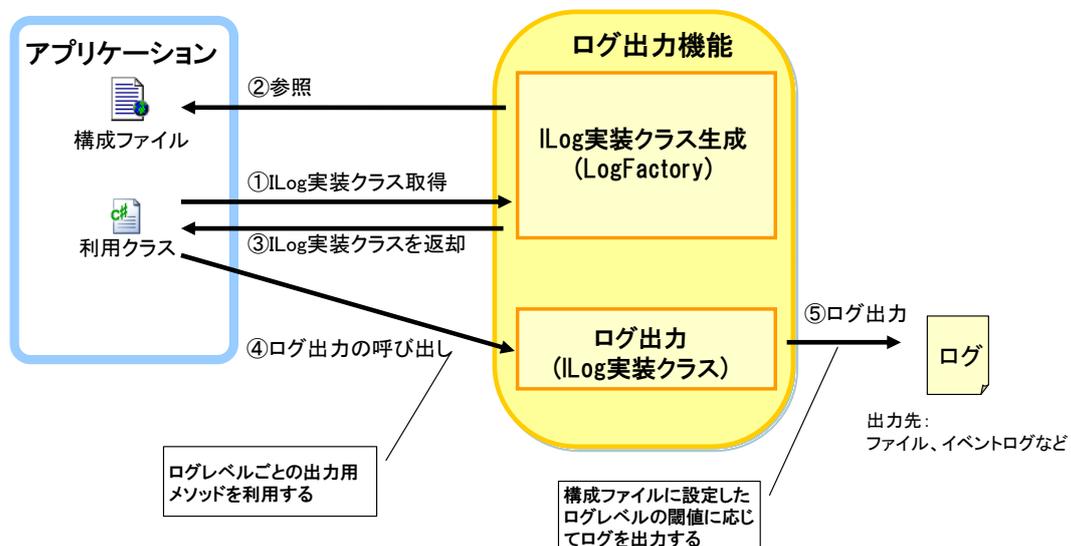


図 1 ログ出力機能

ログ出力機能を利用するクラスは、はじめにロガー (ILog 実装クラスのインスタンス) を取得する。ログ出力機能の内部では、構成ファイル³を参照して生成するロガーの種類や設定を決定する。ログを出力する際は、ロガーに用意されているログレベルごとのログ出力用メソッドを呼び出す。

本機能では、ロガーの標準実装として **TraceSourceLogger** を提供している。

¹ Logging AB : マイクロソフトが推進するオープンソース・ライブラリ「Enterprise Library」に含まれる、ログ出力用の Application Block [http://www.codeplex.com/entlib]

log4net : Apache プロジェクトでオープンソースとして開発されているロギングライブラリ。「log4j」を .NET に移植したもの [http://logging.apache.org/log4net/index.html]

² Jakarta Commons Logging : Apache プロジェクトでオープンソースとして開発されているロギングパッケージ間の互換性問題解決するためのコンポーネント [http://commons.apache.org/logging/]

³ 構成ファイル: web.config や App.config のことを示す。

■ 使用方法

◆ 構成ファイル

TERASOLUNA が標準で提供する TraceSourceLogger を利用する場合、構成ファイルに TraceSource の設定をする。

表 1 構成ファイル

ノード	属性	必須	値
/configuration/system.diagnostics/			複数可
		○	詳細な設定方法は MSDN を参照

● TraceSource の設定

TraceSourceLogger を使用する場合、TraceSource の設定を構成ファイルに記述する。TraceSourceLogger は、構成ファイルで定義した「カテゴリ」、カテゴリごとの「ログレベル」「出力先」に基づいて、ログ出力する機能を提供する。

構成ファイルに設定する TraceSource のログレベルに関して、本機能で定義するログレベルと TraceSource のログレベルに違いがあるため、注意が必要である。構成ファイルに設定するログレベルの閾値と、有効なログレベルの対応を表 2 に示す。

表 2 ログレベルの閾値と有効なログレベルの対応表

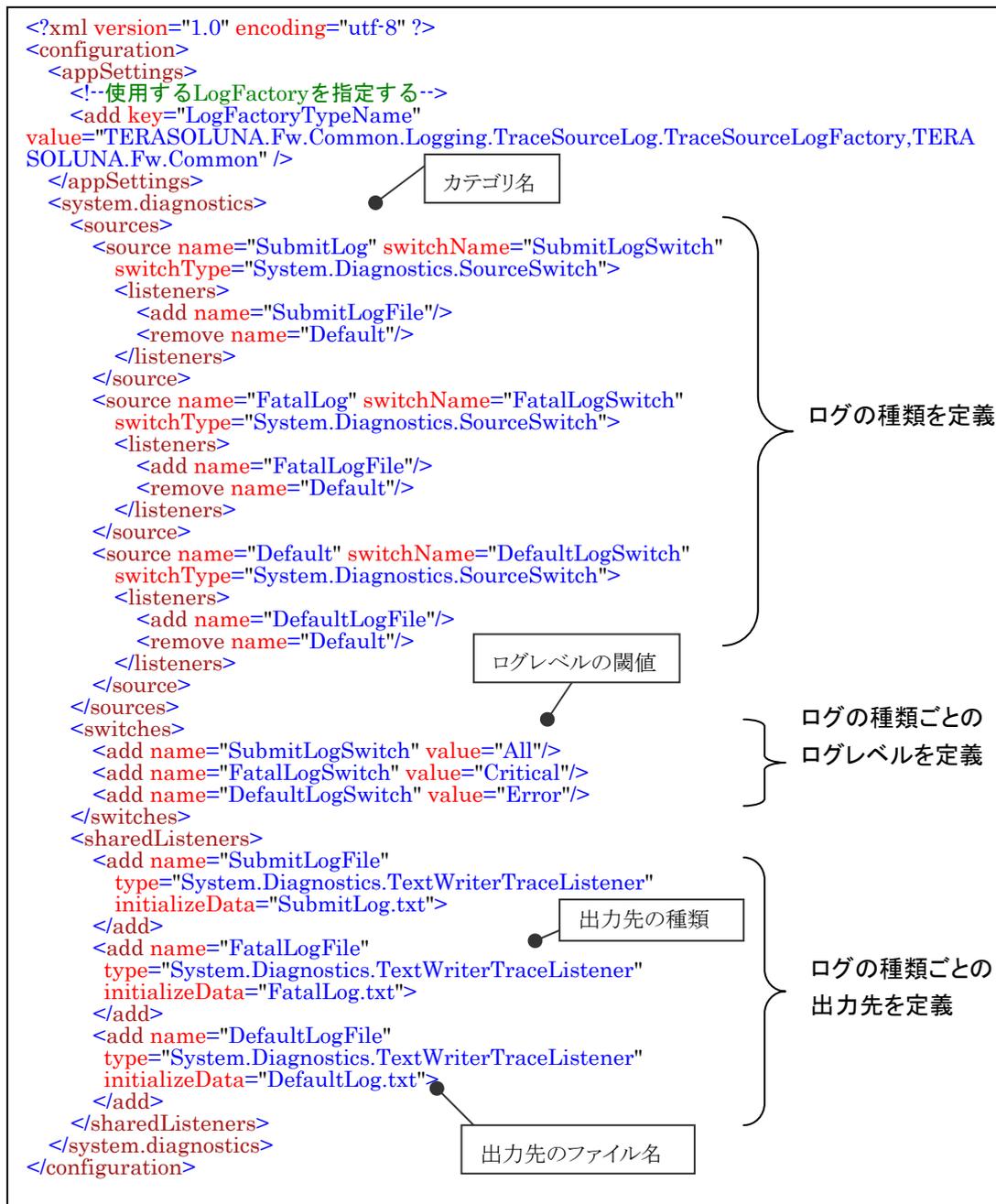
項番	ログレベル	構成ファイルに 設定するログレベルの閾値						
		Off	Critical	Error	Warning	Information	Verbose	All
1	FATAL		○	○	○	○	○	○
2	ERROR			○	○	○	○	○
3	WARN				○	○	○	○
4	INFO					○	○	○
5	DEBUG						○	○
6	TRACE							○

表 3 に示すような関心の異なるログを、別々のファイルに出力する場合の構成ファイルの設定例をリスト 1 に示す。

表 3 出力するログの種類

項番	ログの種類	カテゴリ名	ログレベル の閾値	出力先の ファイル名
1	統計用のユーザによるボタン押下を単位としたログ	SubmitLog	TRACE	SubmitLog.txt
2	運用管理ソフトウェアの監視対象として利用するログ	FatalLog	FATAL	FatalLog.txt

3	その他のログ	Default	ERROR	DefaultLog.txt
---	--------	---------	-------	----------------



リスト 1 構成ファイル記述例

TraceSource の詳細な設定方法は MSDN を参照のこと。

◆ 実装方法

- ロガーの取得方法
ロガーを取得するにはLogFacotryクラスのGetLoggerメソッドを呼び出す。GetLoggerメソッドの一覧を表 4に示す。

表 4 LogFactory クラスの GetLogger メソッド一覧

項番	GetLogger メソッドの種類	第 1 引数	第 2 引数	戻り値の型
1	public static ILog GetLogger(string className, params object[] args)	クラス名を表す文字列	object 型の可変長引数	ILog
2	public static ILog GetLogger(Type classType, params object[] args)	クラスの Type	object 型の可変長引数	ILog
3	public static T GetLogger<T>(string className, params object[] args) where T : class, ILog	クラス名を表す文字列	object 型の可変長引数	T(ジェネリック型)
4	public static T GetLogger<T>(Type classType, params object[] args) where T : class, ILog	クラスの Type	object 型の可変長引数	T(ジェネリック型)

TraceSourceLogger を使用する場合、GetLogger メソッドの第 2 引数には構成ファイルに定義したカテゴリ名を指定する。省略した場合は "Default" になる。指定したカテゴリ名が構成ファイルに定義されていない場合、エラーにはならないが、ログレベルの閾値が Off となるためログは出力されない。また、第 3 引数以降の入力は無視される。

カテゴリ名を指定してロガーを取得する場合と、指定せずに取得する場合の実装例をリスト 2に示す。

```
class Program
{
    // カテゴリ名 ("CategoryName") を指定してロガーを取得する
    private static ILog _loggerA = LogFactory.GetLogger(typeof(Program), "CategoryName");
    // カテゴリ名を指定せずにロガーを取得する
    private static ILog _loggerB = LogFactory.GetLogger(typeof(Program));
    static void Main(string[] args)
    {
        // 処理
    }
}
```

第 2 引数にカテゴリ名を指定する。
省略した場合は "Default" になる。

リスト 2 ロガーを取得する実装例

- ログの出力方法

ログを出力するには、ロガーのログ出力用メソッドを呼び出す。メソッドはログレベルごとに用意されており、引数にはメッセージや例外をとる。ログレベルと対応するメソッドの一覧を表 5に示す。

表 5 ログレベルと対応するメソッド

項番	種類	ログレベル	ログ出力用メソッド	出力ログレベル確認用プロパティ
1	致命的なエラー	FATAL	Fatal(object message) Fatal(object message, Exception ex)	IsFatalEnabled
2	エラー	ERROR	Error(object message) Error(object message, Exception ex)	IsErrorEnabled
3	警告	WARN	Warn(object message) Warn(object message, Exception ex)	IsWarnEnabled
4	情報	INFO	Info(object message) Info(object message, Exception ex)	IsInfoEnabled
5	デバッグ	DEBUG	Debug(object message) Debug(object message, Exception ex)	IsDebugEnabled
6	トレース	TRACE	Trace(object message) Trace(object message, Exception ex)	IsTraceEnabled

引数が 1 つのメソッドでは、“message”に指定された引数をログメッセージとして出力する。引数が 2 つのメソッドでは、“message”に加えて“ex”に指定された例外を共に出力する。

出力ログレベル確認用のプロパティは、そのログレベルが現在有効かどうかをチェックし、有効なら true、無効なら false を返す。ログ出力用メソッドの引数に与えるメッセージを作成するために、文字列の連結のような重い処理を行う場合、このプロパティをチェックすることで、ログ出力のオーバーヘッドを減らすことができる。

※イベント ID を指定してログを出力する方法

イベントIDはデフォルトで 0 に設定されているが、TraceSourceLoggerに実装されているログ出力用メソッドを利用することで、イベントIDを指定してログを出力することができる。このメソッドを呼び出す場合、ロガーを取得するGetLoggerメソッドにジェネリックの型 (ITraceSourceLog) を指定し、取得したロガーをITraceSourceLog型で受け取る。実装例をリスト 3に示す。

```
class Program
{
    // ジェネリックの型を指定してロガーを取得する
    private static ITraceSourceLog logger =
        LogFactory.GetLogger<ITraceSourceLog>(typeof(Program));

    static void Main(string[] args)
    {
        if (logger.IsDebugEnabled)
        {
            // イベントID
            int eventId = 1;

            logger.Debug("出力するログメッセージ", eventId);
        }
    }
}
```

ジェネリックの型に ITraceSourceLog を指定

第 2 引数にイベント ID を指定するメソッドを呼び出す

リスト 3 ジェネリックの型を指定してロガーを取得する実装例

- 実装例

表 3に示した種類のログを、カテゴリごとに出力する場合の実装例をリスト 4 に示す。なお、カテゴリの定義、カテゴリごとのログレベルや出力先の定義はリスト 1 のように構成ファイルに設定されているものとする。

```

class Program
{
    // 統計用のユーザによるボタン押下を単位としたログ
    private static ILog _submitLog = LogFactory.GetLogger(typeof(Program), "SubmitLog");

    // 運用管理ソフトウェアの監視対象として利用するログ
    private static ILog _fatalLog = LogFactory.GetLogger(typeof(Program), "FatalLog");

    // その他のログ(カテゴリはDefault)
    private static ILog _defaultLog = LogFactory.GetLogger(typeof(Program));

    static void Main(string[] args)
    {
        // ログの出力
        if (_submitLog.IsTraceEnabled)
        {
            _submitLog.Trace("統計用のログメッセージ");
        }

        if (_fatalLog.IsFatalEnabled)
        {
            _fatalLog.Fatal("運用管理ソフトウェアの監視対象用のログメッセージ");
        }

        if (_defaultLog.IsFatalEnabled)
        {
            _defaultLog.Fatal("致命的エラー");
        }

        try
        {
            throw new Exception();
        }
        catch (Exception ex)
        {
            if (_defaultLog.IsErrorEnabled)
            {
                _defaultLog.Error("エラー", ex);
            }
        }

        if (_defaultLog.IsWarnEnabled)
        {
            _defaultLog.Warn("警告");
        }
    }
}

```

第 2 引数にカテゴリ名を指定する。
省略した場合は"Default" になる。

Defaultカテゴリはログレベルの
閾値が"ERROR"に設定されている
ため、FATALとERRORが出力される

WARNは出力されない

リスト 4 TraceSourceLog を使用する場合の実装例

- 出力結果例

リスト 4 の実装例を実行した場合、ログの種類ごとにテキストファイルが作成される。それぞれの出力結果を以下に示す。

SubmitLog.txt

SubmitLog Verbose: 0 : TRACE 2008-06-27 20:14:58,058 [1] (ConsoleApplication1.Program) - 統計用のログ
メッセージ

FatalLog.txt

FatalLog Critical: 0 : FATAL 2008-06-27 20:14:58,058 [1] (ConsoleApplication1.Program) - 運用管理ソフトウェアの監視対象用のログメッセージ

DefaultLog.txt

Default Critical: 0 : FATAL 2008-06-27 20:14:58,073 [1] (ConsoleApplication1.Program) - 致命的エラー
 Default Error: 0 : ERROR 2008-06-27 20:14:58,073 [1] (ConsoleApplication1.Program) - エラー
 <System.Exception - 種類 'System.Exception' の例外がスローされました。>
 System.Exception : 種類 'System.Exception' の例外がスローされました。
 場所 ConsoleApplication1.Program.Main(String[] args) 場所 C:%Documents and Settings%ユーザー名%My Documents%Visual Studio 2005%Projects%ConsoleApplication1%ConsoleApplication1%Program.cs:行 39

出力結果のフォーマット

{Category} {TraceEventType};{EventID};{LogLevel} {Date} [{ThreadID}] (ClassName) - {Message} <{Exception}> ↩
 Exception Detail

表 6 出力結果のフォーマットの要素

項番	要素名	説明
1	Category	カテゴリ名。
2	TraceEventType	TraceSourceLogger が内部で TraceSource を利用してログ出力する際に指定したイベントの種類。
3	EventID	イベント ID。デフォルトは 0。
4	LogLevel	ログレベル。
5	Date	現在の日付 (yyyy-MM-dd HH:mm:ss,fff)
6	ThreadID	現在実行中のスレッドの ID。
7	ClassName	ログを出力したクラス名。ロガーを取得する GetLogger メソッドの第 1 引数に与えたクラス情報 (クラスの Type もしくはクラス名を表す文字列) が使用される。
8	Message	ログ出力用メソッドの引数に与えられたメッセージ。
9	Exception	ログ出力用メソッドの引数に与えられた例外の型とメッセージ。
10	Exception Detail	例外の型とメッセージ。InnerException を含むスタックトレース。

■ 内部構成

本機能の内部構成について説明する。

- ログレベル

ログを出力する時、その優先度や重要度を考慮して「ログレベル」を設定する。本機能で

は、表 7 ログレベルの種類に示す 6 つのログレベルを用意している。

表 7 ログレベルの種類

項番	種類	ログレベル	説明
1	致命的なエラー	FATAL	システムに対して致命的な障害が発生した場合に利用する。
2	エラー	ERROR	予期せぬ動作などにより、正しく処理できない場合に利用する。通常は例外発生時に出力する。
3	警告	WARN	エラーの原因やバグの原因となりそうなときに利用する。例外をやむを得ず捕捉して、処理を業務フローに戻す場合などに出力する。
4	情報	INFO	システムの動作を通知する際に利用する。起動時や処理の開始など、重要な動作(外部仕様にかかわるもの)をトレースしたい場合に出力する。
5	デバッグ	DEBUG	起動時・メソッド呼び出し時・インスタンス生成時など、内部の動作をトレースしたい場合に利用する。
6	トレース	TRACE	詳細なデバッグ情報を出力する際に利用する。モジュール内部の情報、ループの繰り返して大量に出力される情報など。

表 7のログレベルは、Jakarta Commons Loggingに合わせて定義したものである。TraceSourceやLogging AB、log4netのログレベルとは表 8のように対応している。コーディング時に特に意識する必要はないが、個別のロギングパッケージの設定をする場合に注意が必要である。

表 8 TERASOLUNA Logging とロギングパッケージのログレベルの対応

項番	TERASOLUNA Logging	ロギングパッケージ		
		TraceSource	Logging AB	Log4net
1	FATAL	Critical	Critical	FATAL
2	ERROR	Error	Error	ERROR
3	WARN	Warning	Warning	WARN
4	INFO	Information	Information	INFO
5	DEBUG	Verbose	Verbose	DEBUG
6	TRACE	※	※	※

※ロギングパッケージには TRACE にあたるログレベルが定義されていないため、個別の Logger で定義する必要がある。標準実装の TraceSourceLogger における Trace レベルの定義は、「TraceSource の設定」を参照のこと。

◆ 構成クラス

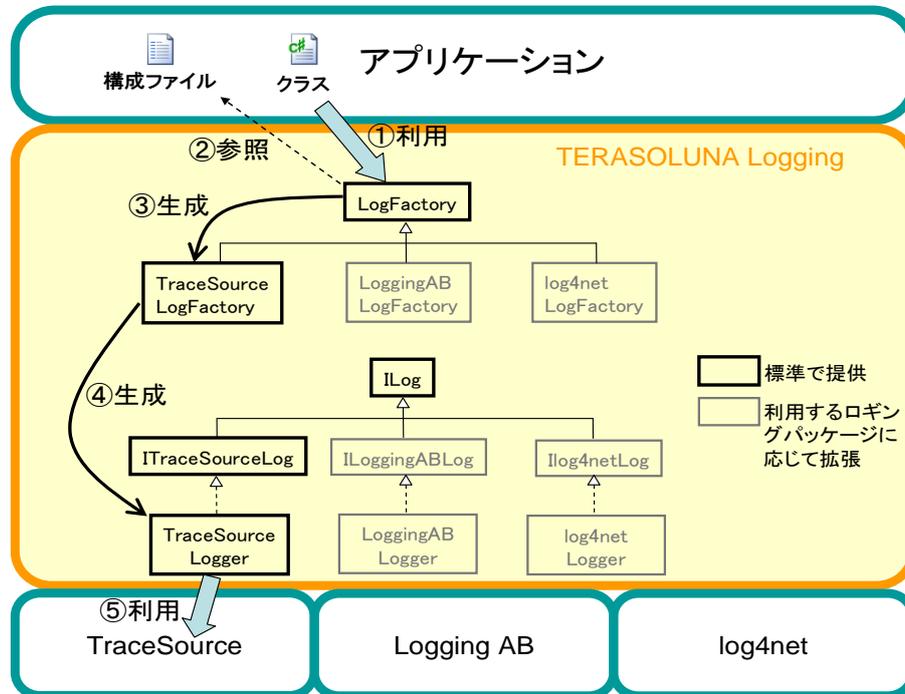


図 2 構成クラス図

表 9 構成クラス一覧

項番	クラス名	説明
1	ILog	ログ出力クラスに関わらず、統一的な方法でログを出力するためのインターフェイス。
2	LogFactory	ILog 実装クラスを生成するための抽象メソッドを持つ抽象クラス。構成ファイルに基づいて、ファクトリクラスを生成する。
3	ITraceSourceLog	ILog を継承し、TraceSource 用の抽象メソッドを追加したインターフェイス。
4	TraceSourceLogger	ITraceSourceLog を実装したログ出力クラス。TraceSource を使ってログを出力する。
5	TraceSourceLogFactory	TraceSourceLogger を生成するファクトリクラス。LogFactory を継承しており、TraceSourceLogger を生成するためのメソッドを実装している。
6	LogLevel	ログレベル列挙体を定義するクラス。

■ 拡張ポイント

Logging AB や log4net などの利用するロギングパッケージに応じて本機能を拡張する場合、以下のクラスを作成する。

- ILog 実装クラス

ILog に定義されているログ出力用メソッドと、出力ログレベル確認用プロパティを実装する。個別のロギングパッケージの機能を実現するために、必要に応じてメソッドやプロパティを追加する。

- LogFactory 継承クラス

ILog 実装クラスのインスタンスを取得するための抽象メソッドを実装する。

```
protected abstract ILog GetInstance(string className, params object[] args);
protected abstract ILog GetInstance(Type classType, params object[] args);
```

- LogFactory 継承クラスの指定

表 10 構成ファイル

ノード	属性	必須	値
/configuration/appSettings /add	key	○	構成要素名。 固定値。以下の値とする。 LogFactoryTypeName
			複数可
	value		LogFactory 継承クラスの完全修飾型名

LogFactory クラスは GetLogger メソッドでロガー(ILog 実装クラスのインターフェイス)を取得する時に、ログ出力に使用するロギングパッケージを決定する。使用するロギングパッケージに対応する LogFactory 継承クラスを、構成ファイルに記述する。

appSettings 要素内に add 要素を追加し、key 属性に”LogFactoryTypeName”、value 属性に LogFactory 継承クラスの完全修飾型名を設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!--使用するLogFactory継承クラスを指定する-->
    <add key="LogFactoryTypeName"
         value="SampleAP.Logging.SampleLog.SampleLogFactory, SampleAP" />
  </appSettings>
</configuration>
```

LogFactory 継承クラスの完全修飾型名

リスト 5 構成ファイルの設定例

CM-04 ビジネスロジック生成機能

■ 概要

ビジネスロジック設定ファイルの定義をもとに、ビジネスロジック名に対応するビジネスロジッククラスのインスタンスを生成する機能を提供する。

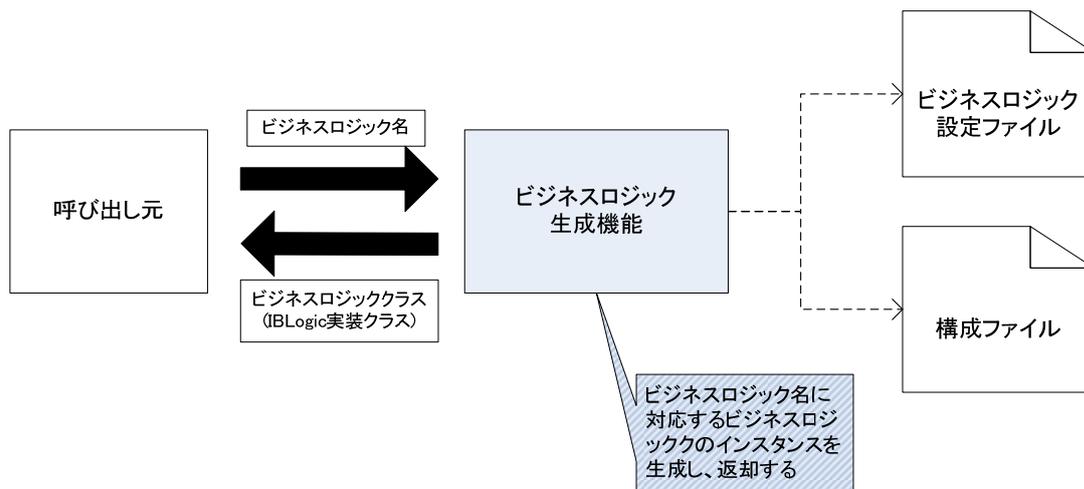


図 1 ビジネスロジック生成機能の概念図

全てのビジネスロジックは、IBLogic インタフェースを実装する必要がある。

ビジネスロジック設定ファイルを変更することで、再ビルドを行わずに利用するビジネスロジッククラスを変更することができる。

■ 使用方法

◆ 構成ファイル

構成ファイルに、blogicConfiguration 要素を有効化するためのクラスの設定と、ビジネスロジック設定ファイルのパスを指定する。ビジネスロジック設定ファイルは複数指定できる。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
      type="TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection,
        TERASOLUNA.Fw.Common"/>
  </configSections>

  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="Config¥BLogicConfiguration01.config"/>
      <file path="Config¥BLogicConfiguration02.config"/>
    </files>
  </blogicConfiguration>

```

構成ファイルの blogicConfiguration 要素を有効化するためのクラスを定義する。

ビジネスロジック定義ファイルは複数設定が可能。

リスト 1 構成ファイルの記述例

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルにビジネスロジックのビジネスロジック名とアセンブリ修飾名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。

以下に、ビジネスロジック設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <blogic name="sample01" type="Sample.BLogic.SampleBLogic01, Sample" />
  <blogic name="sample02" type="Sample.BLogic.SampleBLogic02`1, Sample" />
</blogicConfiguration>
```

name 属性にビジネスロジック名を指定する。

type 属性にビジネスロジッククラスのアセンブリ修飾名を指定する。

ジェネリックパラメータを持つビジネスロジックを生成する場合は、アセンブリ修飾名の末尾に「+ ジェネリックパラメータの個数」を付与する。

リスト 2 ビジネスロジック設定ファイルの記述例

◆ 実装方法

(1) ビジネスロジックの作成

ジェネリックパラメータを持たないビジネスロジックと、ジェネリックパラメータを持つビジネスロジックの実装方法を以下に記す。

- ジェネリックパラメータを持たないビジネスロジックの実装

```
/// <summary>
/// サンプルBLogic。
/// </summary>
public class SampleBLogic01 : ILogic
{
    public BLogicResult Execute(BLogicParam param)
    {
        BLogicResult blogicResult = new BLogicResult();
        blogicResult.ResultString = BLogicResult.SUCCESS;

        return blogicResult;
    }
}
```

リスト 3 ジェネリックパラメータを持たないビジネスロジックの実装例

- ジェネリックパラメータを持つビジネスロジックの実装

```
/// <summary>
/// サンプルBLogic (ジェネリックパラメータ付き)。
/// </summary>
public class SampleBLogic02<T> : IBLogic where T : DataSet, new()
{
    public BLogicResult Execute(BLogicParam param)
    {
        BLogicResult blogicResult = new BLogicResult();

        T resultDataSet = new T();
        resultDataSet.Tables[0].Rows.Add("VALUE01", "VALUE02");

        blogicResult.ResultString = BLogicResult.SUCCESS;
        blogicResult.ResultData = resultDataSet;

        return blogicResult;
    }
}
```

リスト 4 ジェネリックパラメータを持つビジネスロジックの実装例

- BLogicParam のプロパティ
BLogicParam のプロパティ一覧を示す。

表 1 BLogicParam のプロパティ

項番	プロパティ名	説明
1	ParamData	ビジネスロジックの入力データセット。
2	Items	ビジネスロジックの入力パラメータを格納する IDictionary。

- BLogicResult のプロパティ
BLogicResult のプロパティ一覧を示す。

表 2 BLogicResult のプロパティ

項番	プロパティ名	説明
1	ResultString	ビジネスロジックの実行結果を表す文字列が格納される。ビジネスロジックが正常に実行された場合は、”success” (BLogicResult.SUCCESS) を設定すること。
2	ResultData	ビジネスロジックの実行結果を格納するデータセット。
3	Errors	ビジネスロジック内部で発生したエラーを格納するリスト。
4	Items	ビジネスロジックの実行結果を格納する IDictionary。

(2) ビジネスロジック生成クラスの実行

BLogicFactory の CreateBLogic メソッドを実行することで、ビジネスロジック設定ファイルに定義したビジネスロジッククラスのインスタンスを生成することができる。CreateBLogic メソッドの引数には、ビジネスロジック設定ファイルに定義したビジネスロジック名を指定すること。ジェネリックパラメータを持つビジネスロジックを生成する場合は、CreateBLogic メソッドの第二引数にジェネリックパラメータの型を設定する必要がある。

ジェネリックパラメータを持たないビジネスロジックを生成する場合と、ジェネリックパラメータを持つビジネスロジックを生成する場合の実装方法を以下に記す。

- ジェネリックパラメータを持たないビジネスロジックを生成する

```
// ビジネスロジックの作成
BLogic blogic = BLogicFactory.CreateBLogic("sample01");

// ビジネスロジック入力クラスの作成
BLogicParam param = new BLogicParam();

// ビジネスロジック入力クラスへのパラメータ設定

// ビジネスロジックの実行
BLogicResult result = blogic.Execute(param);

// ビジネスロジック実行結果の確認
if (BLogicResult.SUCCESS.Equals(result.ResultString))
{
    Console.WriteLine("ビジネスロジック実行に成功しました。");
}
else
{
    Console.WriteLine("ビジネスロジック実行に失敗しました。");
}
```

ビジネスロジック設定ファイルに定義されたビジネスロジック名を指定する。

リスト 5 ジェネリックパラメータを持たないビジネスロジックの生成例

- ジェネリックパラメータを持つビジネスロジックを生成する

```
// ビジネスロジックの作成
BLogic blogic = BLogicFactory.CreateBLogic("sample02",
typeof(SampleDataSet).AssemblyQualifiedName);

// ビジネスロジック入力クラスの作成
BLogicParam param = new BLogicParam();

// ビジネスロジック入力クラスへのパラメータ設定

// ビジネスロジックの実行
BLogicResult result = blogic.Execute(param);

// ビジネスロジック実行結果の確認
if (BLogicResult.SUCCESS.Equals(result.ResultString))
{
    Console.WriteLine("ビジネスロジック実行に成功しました。");
}
else
{
    Console.WriteLine("ビジネスロジック実行に失敗しました。");
}
```

ビジネスロジック設定ファイルに定義されたビジネスロジック名を指定する。

ジェネリックパラメータのアセンブリ修飾名を指定する。

リスト 6 ジェネリックパラメータを持つビジネスロジックの生成例

- NopBLogic

BLogicFactory の CreateBLogic メソッドの引数に、空文字列もしくは null を設定した場合、BLogicFactory は NopBLogic のインスタンスを返却する。NopBLogic は BLogicResult の ResultString に"success"を設定する処理のみを行う。

■ 拡張ポイント

◆ ビジネスロジック生成クラスの差し替え

構成ファイルの `AppSettings` セクションに `BLogicFactory` を拡張したクラスのアセンブリ修飾名を指定することで、利用するビジネスロジック生成クラスの実装を差し替えることができる。アセンブリ修飾名の指定を省略した場合は、TERASOLUNA が提供する `BLogicFactory` クラスが使用される。なお、アセンブリ修飾名を指定する `AppSettings` セクションのキーは「`BLogicFactoryTypeName`」である。

以下に、ビジネスロジック生成クラスの差し替えを設定する例を示す。

```
<configuration>
  <appSettings>
    <add key="BLogicFactoryTypeName"
        value="Sample.BLogic.BLogicFactoryEx, Sample" />
  </appSettings>
</configuration>
```

リスト 7 ビジネスロジック生成クラスを差し替える設定例

WA-01 画面遷移管理機能

■ 概要

本機能は、ページ設定ファイルに定義した画面遷移情報に基づいて画面遷移を行う機能である。

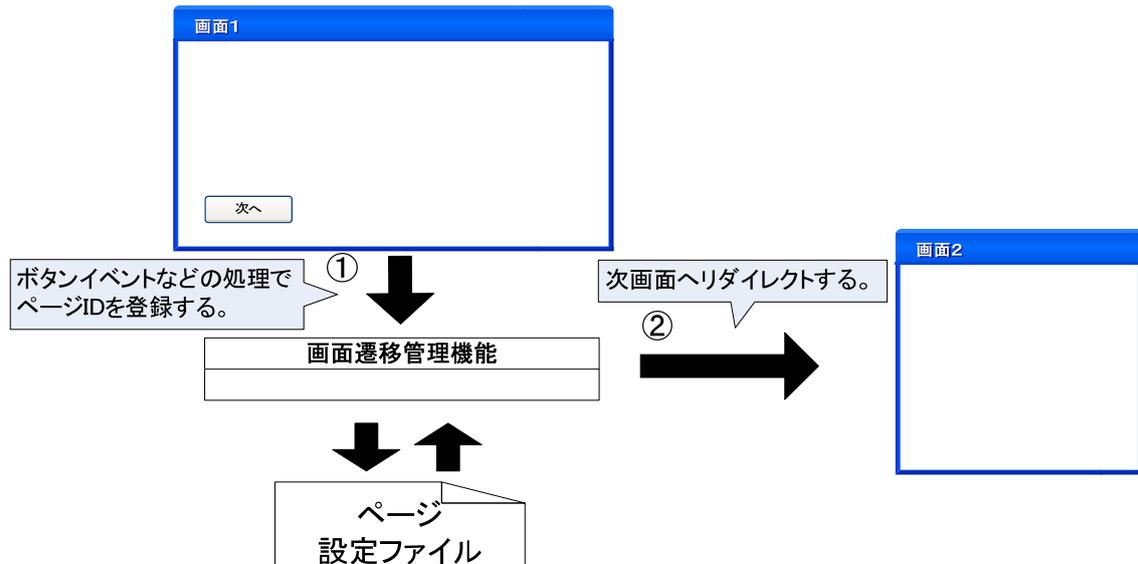


図 1 画面遷移管理機能

ページ設定ファイルには、画面に対して一意なページ ID と画面の URL を定義する。ボタンクリックイベントなどで遷移先ページ ID を指定することで、ページ設定ファイルから遷移先ページの URL を取得し、次画面へリダイレクトする。遷移先 URL をページ設定ファイルから取得するため、再ビルドを行わずに画面遷移先を変更することができる。また、ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule およびページ設定ファイルのパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	name	○	構成要素名。 固定値、以下を指定する。 pageConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.Configuration.Page.ConfigurationSection, TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file			複数可
	path	○	ページ設定ファイルの保存先のパスを指定する。
/System.web/httpModules	name	○	モジュール登録名。 固定値、以下を指定する。 TransitionListenerImpl
	type	○	クラス名と、クラスが含まれるアセンブリ名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.HttpModule.TransitionListenerImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <configSections>
    <section name="pageConfiguration"
              type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                  TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration01.config" />
      <file path="Config¥PageConfiguration02.config" />
    </files>
  </pageConfiguration>
  <system.web>
    <httpModules>
      <add name="TransitionListenerImpl"
           type="TERASOLUNA.Fw.Web.HttpModule.TransitionListenerImpl,
               TERASOLUNA.Fw.Web" />
    </httpModules >
  </system.web>
  ...
</configuration>
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルには、ページ毎のページ ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移保証機能、二重押下防止機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値						
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd						
/pageConfiguration/page			複数可。						
	name	○	ページ ID。重複不可。						
	path	○	ページパス。重複不可。						
	preventDoubleSubmit		二重押下防止フラグ。 デフォルト値は、on。on、または off を指定可能。						
			<table border="1"> <thead> <tr> <th>設定値</th> <th>説明</th> </tr> </thead> <tbody> <tr> <td>on</td> <td>ページに対して二重押下防止機能を有効にする。</td> </tr> <tr> <td>off</td> <td>ページに対して二重押下防止機能を無効にする。</td> </tr> </tbody> </table>	設定値	説明	on	ページに対して二重押下防止機能を有効にする。	off	ページに対して二重押下防止機能を無効にする。
	設定値	説明							
	on	ページに対して二重押下防止機能を有効にする。							
	off	ページに対して二重押下防止機能を無効にする。							
	checkToken		トークンチェックフラグ。 デフォルト値は、on。on、または off を指定可能。						
			<table border="1"> <thead> <tr> <th>設定値</th> <th>説明</th> </tr> </thead> <tbody> <tr> <td>on</td> <td>ページに対してトークンチェック機能を有効にする。</td> </tr> <tr> <td>off</td> <td>ページに対してトークンチェック機能を無効にする。</td> </tr> </tbody> </table>	設定値	説明	on	ページに対してトークンチェック機能を有効にする。	off	ページに対してトークンチェック機能を無効にする。
設定値	説明								
on	ページに対してトークンチェック機能を有効にする。								
off	ページに対してトークンチェック機能を無効にする。								
updateToken		トークン更新フラグ。 デフォルト値は、on。on、または off を指定可能。							
		<table border="1"> <thead> <tr> <th>設定値</th> <th>説明</th> </tr> </thead> <tbody> <tr> <td>on</td> <td>ページ遷移時にトークンを更新する。</td> </tr> <tr> <td>off</td> <td>ページ遷移時にトークンを更新しない。</td> </tr> </tbody> </table>	設定値	説明	on	ページ遷移時にトークンを更新する。	off	ページ遷移時にトークンを更新しない。	
設定値	説明								
on	ページ遷移時にトークンを更新する。								
off	ページ遷移時にトークンを更新しない。								

ページ ID とパスは、すべてのページ設定ファイルにおいて一意である必要がある。異なるページ設定ファイルに重複するページ ID やパスがある場合、設定ファイルの読み込み時に TERASOLUNA フレームワークから ConfigurationErrorsException がスローされる。

preventDoubleSubmit 属性は「二重押下防止機能」、checkToken 属性、updateToken 属

性については「画面遷移保証機能」を参照のこと。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="n01" path="/UI/n01.aspx"/>
  <page name="n02" path="/UI/n02.aspx"/>
  <page name="m01" path="/UI/t1/n01.aspx"/>
  <page name="m02" path="/UI/t1/n02.aspx"/>
  <page name="m03" path="/UI/t1/n03.aspx"/>
  <page name="m04" path="/UI/t2/n01.aspx"/>
  <page name="m05" path="/UI/t2/n02.aspx"/>
  <page name="m06" path="/UI/t2/n03.aspx"/>
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

◆ 実装方法

WebUtils クラスの Transit メソッドにページ ID やクエリ文字列を指定する。

```
/// "http://<ホスト名>/<アプリケーションルート名>/UI/n01.aspx"
/// へ遷移するクリックイベント実装例
protected void Button1_Click(object sender, EventArgs e)
{
    WebUtils.Transit("n01");
}

/// "http://<ホスト名>/<アプリケーションルート名>/UI/n01.aspx?k01=v01&k02=v02"
/// へ遷移するクリックイベント実装例
protected void Button2_Click(object sender, EventArgs e)
{
    IDictionary<string, string> querys = new Dictionary<string, string>();
    querys.Add("k01", "v01");
    querys.Add("k02", "v02");
    WebUtils.Transit("n01", querys);
}

/// "http://<ホスト名>/<アプリケーションルート名>/UI/t1/n01.aspx?k03=v03"
/// へ遷移するクリックイベント実装例
protected void Button3_Click(object sender, EventArgs e)
{
    WebUtils.Transit("m01", "k03", "v03");
}
```

リスト 3 実装例

■ 関連機能

- WA-02 画面遷移保証機能
- WA-03 二重押下防止機能
- WA-04 エラー画面遷移機能

WA-02 画面遷移保証機能

■ 概要

本機能は、ページ設定ファイルに定義したトークン情報に基づいて画面への不正アクセスを防止する機能である。

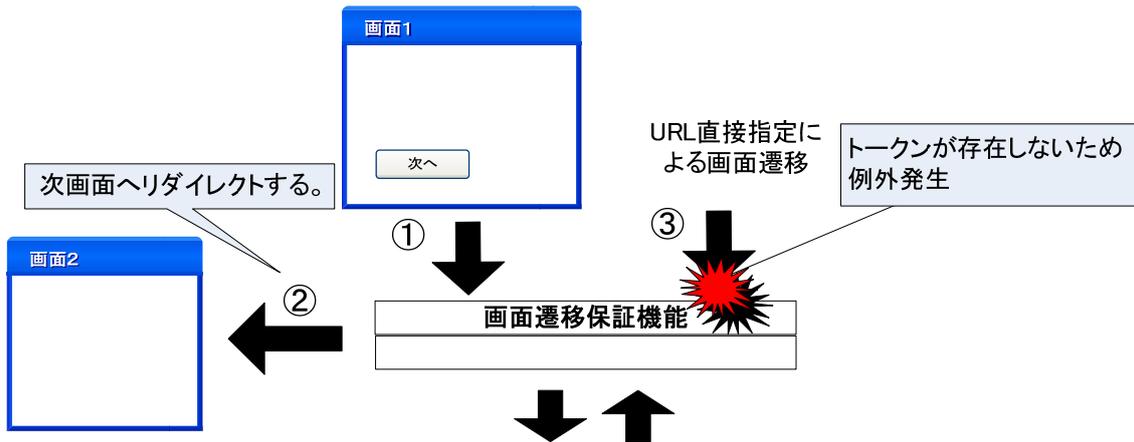


図 1 画面遷移保証機能

ページ設定ファイルには、画面に対して一意なページ ID、画面の URL、トークンチェックフラグ、トークン更新フラグを定義する。遷移先ページへのリダイレクトや URL 直接指定による画面遷移時に、遷移先ページのトークンチェックフラグが有効であれば、不正な画面遷移ではないか確認する。もし、不正な画面遷移である場合は、例外が発生する。また、ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule とページ設定ファイルを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	name	○	構成要素名。 固定値。以下の値を指定する。 pageConfiguration
	type	○	構成設定の処理を行う構成セクション ハンドラクラス名。 固定値。以下の値を指定する。 TERASOLUNA.Fw.Web.Configuration. Page.PageConfigurationSection, TE RASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file			複数可
	path	○	ページ設定ファイルのアプリケーション ルートからのパス。
/System.web/httpModules/add			複数可
	name	○	Http モジュール登録名。 固定値。以下の値を指定する。 TokenProcessorImpl
	type	○	クラス名とクラスが含まれる、アセンブリ名。 固定値。以下の値を指定する。 TERASOLUNA.Fw.Web.HttpModule.T okenProcessorImpl, TERASOLUNA.F w.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <configSections>
    <section name="pageConfiguration"
              type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                  TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration01.config" />
      <file path="Config¥PageConfiguration02.config" />
    </files>
  </pageConfiguration>
  <system.web>
    <httpModules>
      <add name="TokenProcessorImpl"
           type="TERASOLUNA.Fw.Web.HttpModule.TokenProcessorImpl,
               TERASOLUNA.Fw.Web"/>
    </httpModules>
  </system.web>
  ...

```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページのページ ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移管理機能、二重押下防止機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page			複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit		二重押下防止フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対して二重押下防止機能を有効にする。 off ページに対して二重押下防止機能を無効にする。
	checkToken		トークンチェックフラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対してトークンチェック機能を有効にする。 off ページに対してトークンチェック機能を無効にする。
	updateToken		トークン更新フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページ遷移時にトークンを更新する。 off ページ遷移時にトークンを更新しない。

【checkToken 属性について】

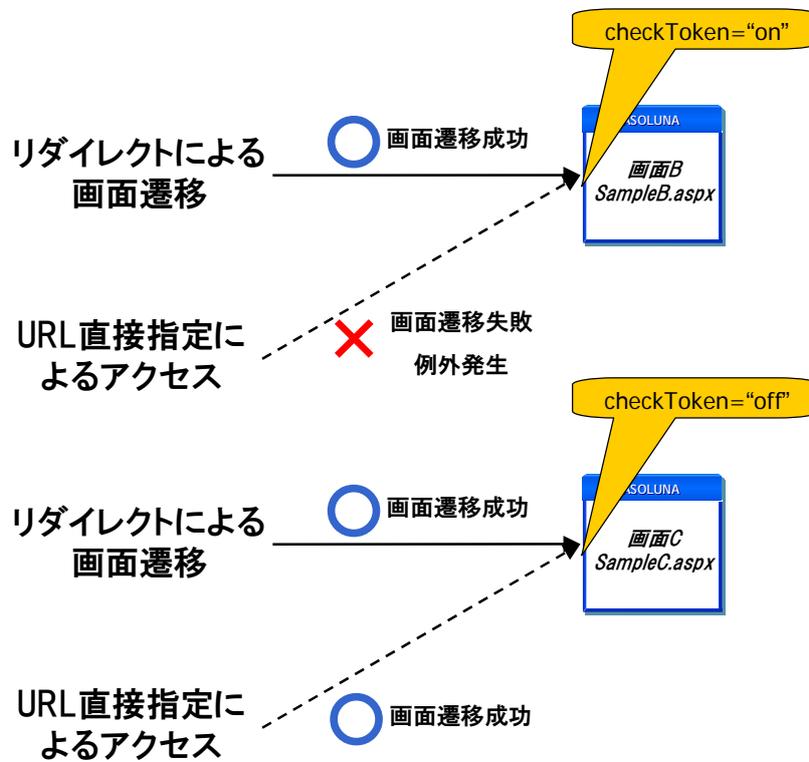


図 2 checkToken の動作例

`checkToken` 属性は、指定されたページに対してトークンをチェックするかどうかを示すフラグである。二重送信を防止したいページや、URL を直接指定してのアクセスを防止したいページにはトークンチェックを有効(`checkToken="on"`)にする。遷移先画面のトークンチェックを有効にすることで、Web アプリケーション内から遷移先画面へリダイレクトする時に、画面遷移保証機能が自動でトークンの設定と確認をする。

【updateToken 属性について】

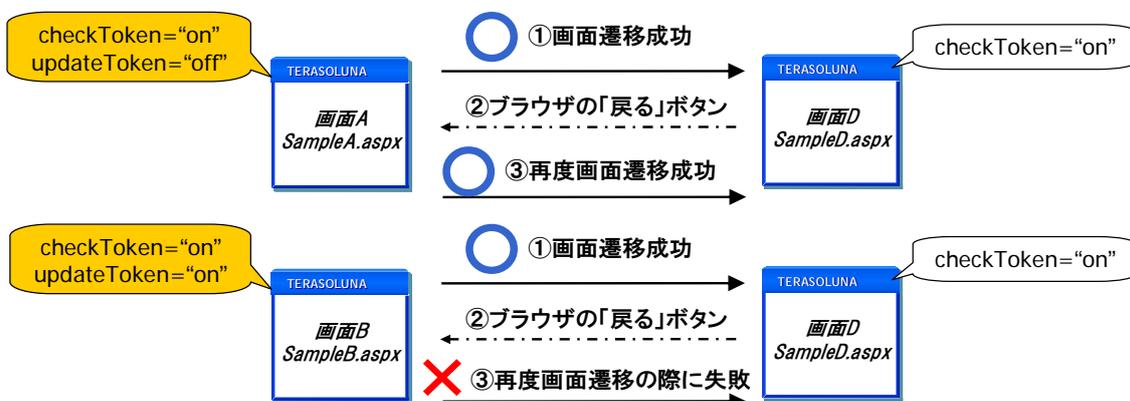


図 3 updateToken の動作例

updateToken 属性は、トークンを更新するかどうかを示すフラグである。ブラウザの「戻る」ボタンで戻ったページから再度 checkToken="on" の画面への画面遷移を許可するためには、トークン更新を無効(updateToken="off")にする。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="n01" path="/SampleA.aspx"
        checkToken="on" updateToken="off"/>
  <page name="n02" path="/SampleB.aspx" checkToken="on" updateToken="on"/>
  <page name="n03" path="/SampleC.aspx" checkToken="off"/>
  <page name="n04" path="/SampleD.aspx" checkToken="on"/>
</pageConfiguration>
```

リスト 2 ページ設定ファイル設定例

■ 関連機能

- WA-01 画面遷移管理機能
- WA-03 二重押下防止機能
- WA-04 エラー画面遷移機能

WA-03 二重押下防止機能

■ 概要

本機能は、ボタンが押下された際に画面内の全てのボタンを無効(disable)にして、リクエストの二重送信を防止する機能である。

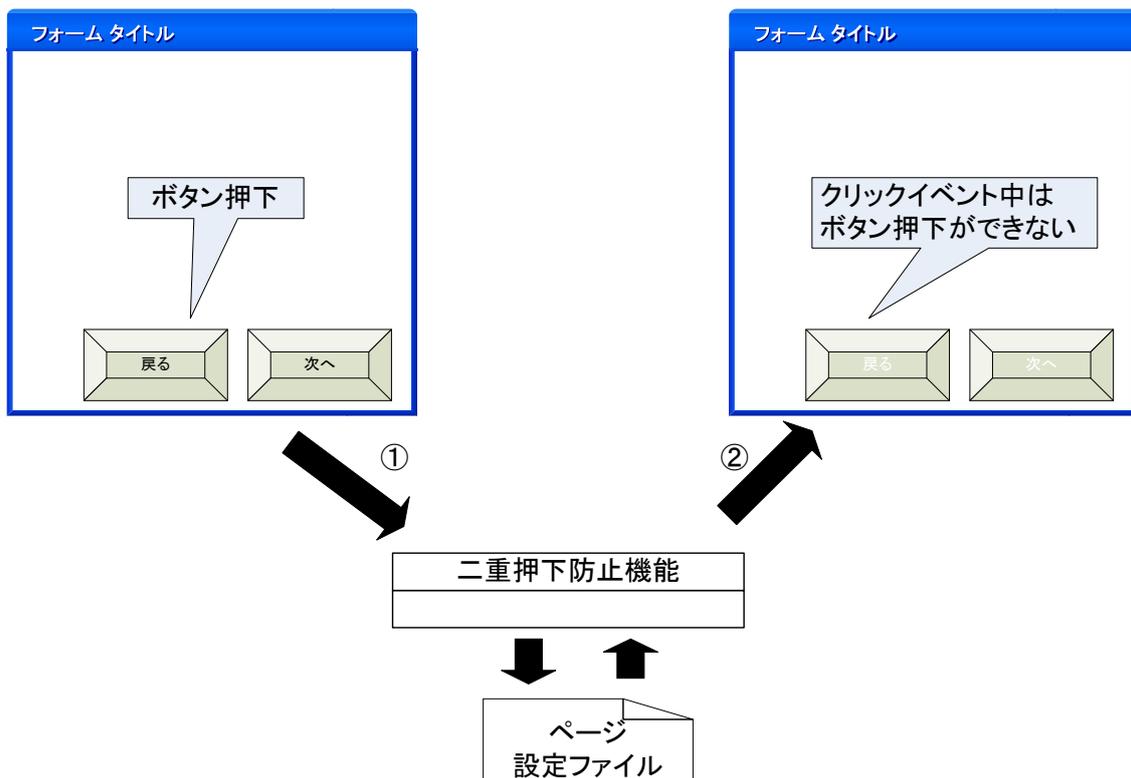


図 1 二重押下防止機能

ページ設定ファイルには、画面に対して一意なページ ID、画面の URL、二重押下防止フラグを定義する。二重押下防止フラグが有効になっている画面のボタンをクリックした場合、画面内の全てのボタンを無効(disable)にする。ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)にページ設定ファイルパスおよびHttpModule を定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	-	-	複数可
	name	-	構成要素名。 固定値、以下の値とする pageConfiguration
	type	-	構成設定の処理を行う構成セクション ハンドラクラス名。 固定値、以下の値とする。 TERASOLUNA.Fw.Web.Configuration. Page.PageConfigurationSection, TE RASOLUNA.Fw.Web
/configuration/pageConfiguration /files/file	-	-	複数可
	path	○	ページ設定ファイルの保存先のパスを指 定する。
/configuration/System.web/http Modules	name	○	モジュール登録名。 固定値、以下の値とする。 PreventDoubleSubmitImpl
	type	○	クラス名と、クラスが含まれるアセンブ リ名。 固定値、以下の値とする。 TERASOLUNA.Fw.Web.HttpModule.P reventDoubleSubmitImpl, TERASOL UNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <configSections>
    <section name="pageConfiguration"
      type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
      TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration.config" />
    </files>
  </pageConfiguration>
  ...
  <system.web>
    <httpModules>
      <add name="PreventDoubleSubmitImpl"
        type="TERASOLUNA.Fw.Web.HttpModule.PreventDoubleSubmitImpl, TERASOLUNA.Fw.Web" />
    </ httpModules >
  </ system.web>
  ...
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページのページ ID、仮想アプリケーションルートパス以降のパス、二重押下防止フラグを関連付ける。ページ設定ファイルについては、画面遷移管理機能、画面遷移保証機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page	-	-	複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit	-	二重押下防止フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対して二重押下防止機能を有効にする。 off ページに対して二重押下防止機能を無効にする。
	checkToken	-	トークンチェックフラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対してトークンチェック機能を有効にする。 off ページに対してトークンチェック機能を無効にする。
	updateToken	-	トークン更新フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページ遷移時にトークンを更新する。 off ページ遷移時にトークンを更新しない。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="menu" path="/UI/menu.aspx" preventDoubleSubmit="on" />
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

■ 関連機能

- WA-01 画面遷移管理機能
- WA-02 画面遷移保証機能
- WA-04 エラー画面遷移機能

WA-04 エラー画面遷移機能

■ 概要

本機能は、Web アプリケーションで例外が発生した時に、エラー画面遷移設定ファイル、ページ設定ファイルに定義した情報に基づいて画面遷移を行う機能である。

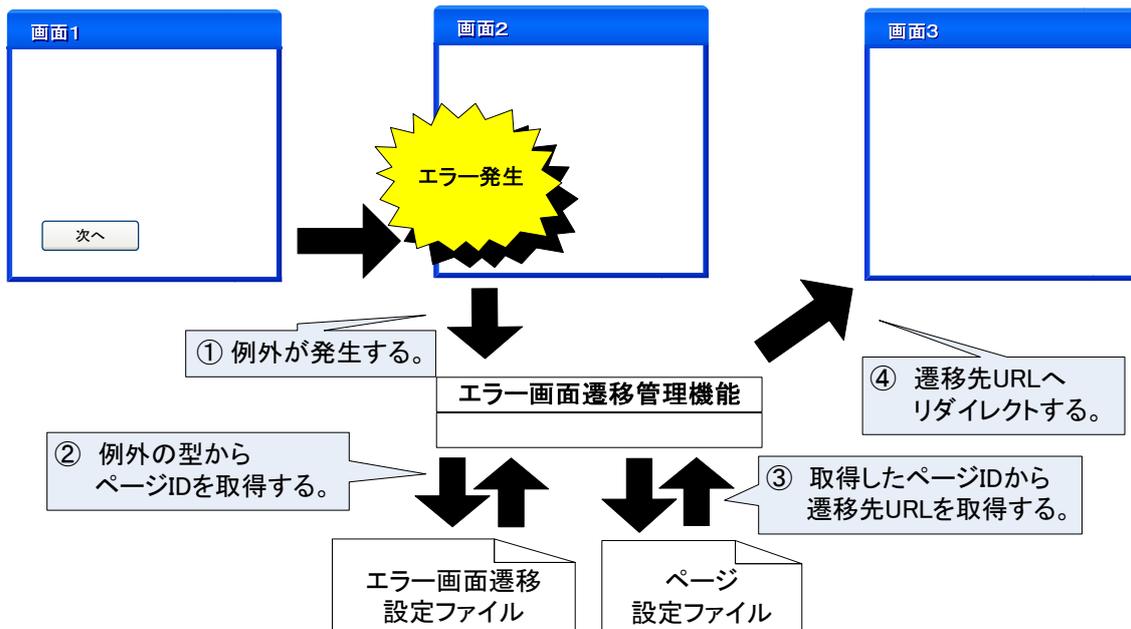


図 1 エラー画面遷移管理機能

エラー画面遷移設定ファイルには、例外クラスの完全修飾名とページ ID を定義する。ページ設定ファイルには、ページ ID と画面の URL を定義する。画面遷移やポストバック実行時に例外が発生した場合、発生した例外クラスに対応したページ ID をエラー画面遷移設定ファイルから取得する。取得したページ ID に対応した遷移先の URL をページ設定ファイルから取得し、エラー画面へリダイレクトする。エラー画面遷移設定ファイル及びページ設定ファイルは、Web 構成ファイル (web.config) に複数定義できるため、ユースケースごとに分割できる。

■ 使用方法

エラー画面遷移機能を使う上で、必要な Web 構成ファイル、ページ設定ファイル、エラー画面遷移設定ファイル、IIS の設定について解説する。

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule、ページ設定ファイルパス、エラー画面遷移設定ファイルパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	-	-	複数可
	name	-	構成要素名。 固定値、以下の値とする。 ・ pageConfiguration ・ exceptionTransitionConfiguration
	type	-	構成設定の処理を行う構成セクション ハンドラ クラス名。 固定値、以下の値とする。 ・ TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection, TERASOLUNA.Fw.Web ・ TERASOLUNA.Fw.Web.Configuration.ExceptionTransition.ExceptionTransitionConfigurationSection, TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file	-	-	複数可
	path	○	ページ設定ファイル。
/configuration/ExceptionTransitionConfiguration	mode	-	エラー画面遷移フラグ。 デフォルト値は、on。 設定値 説明 on エラー画面遷移機能を有効にする。 off エラー画面遷移機能を無効にする。
	logging	-	エラー画面遷移ログ出力フラグ。 デフォルト値は、off。 IIS ログに例外情報を出力するためには、エラー画面遷移ログ出力フラグを on にする以外にも、IIS でログを出力するための設定が必要である。本機能説明書の「IIS の設定」を参照すること。 設定値 説明 on IIS ログを出力する。 off IIS ログを出力しない
/configuration/ExceptionTransitionConfiguration/files/file	-	-	複数可
	path	○	エラー画面遷移設定ファイル。
/System.web/httpModules	name	○	モジュール登録名。 固定値、以下を指定する。 ExceptionTransitionListenerImpl
	type	○	クラス名と、クラスが含まれるアセンブリ名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.HttpModule.ExceptionTransitionListenerImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  ...
  <configSections>
    <section name="pageConfiguration"
      type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
      TERASOLUNA.Fw.Web"/>
    <section name="exceptionTransitionConfiguration"
      type="TERASOLUNA.Fw.Web.Configuration.ExceptionTransition.ExceptionTransitionConfigurationSection, TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration.config" />
    </files>
  </pageConfiguration>
  <exceptionTransitionConfiguration mode="on" logging="on">
    <files>
      <file path="Config¥ExceptionTransition.config"/>
    </files>
  </exceptionTransitionConfiguration>
  <customErrors mode="On" defaultRedirect="DefaultErrorPage.aspx">
  </customErrors>
  <system.web>
    <httpModules>
      <add name="ExceptionTransitionListenerImpl"
        type="TERASOLUNA.Fw.Web.HttpModule.ExceptionTransitionListenerImpl,
        TERASOLUNA.Fw.Web" />
    </httpModules >
  </system.web>
  ...
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページの ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移管理機能、画面遷移保証機能、二重押下防止機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page	-	-	複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit	-	二重押下防止フラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページに対して二重押下防止機能を有効にする。 off ページに対して二重押下防止機能を無効にする。
	checkToken	-	トークンチェックフラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページに対してトークンチェック機能を有効にする。 off ページに対してトークンチェック機能を無効にする。
updateToken	-	トークン更新フラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページ遷移時にトークンを更新する。 off ページ遷移時にトークンを更新しない。	

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="ExTranMain" path="/ExceptionTransitionListerImpl/Main.aspx"/>
  <page name="ExTranError"
        path="/ExceptionTransitionListerImpl/ExTranErrorPage.aspx" />
  <page name="CustomError"
        path="/ExceptionTransitionListerImpl/CustomErrorPage.aspx" />
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

◆ エラー画面遷移設定ファイル

エラー画面遷移設定ファイルにより、例外クラスの完全修飾名とページ ID を関連付ける。

表 3 エラー画面遷移設定ファイル

ノード	属性	必須	値
/ exceptionTransitionConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/ExceptionTransitionSchema.xsd
/exceptionTransitionConfiguration			複数可
/exceptionTransition	exceptionType	○	例外クラスの完全修飾名。
	nextPage	○	遷移先ページ ID。ページ設定ファイルで指定したページ ID を指定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<exceptionTransitionConfiguration
xmlns="http://www.terasoluna.jp/schema/ExceptionTransitionSchema.xsd">
  <exceptionTransition
    exceptionType="System.ArgumentException" nextPage="ExTranError" />
  <exceptionTransition
    exceptionType="TERASOLUNA.Fw.Web.Controller.InvalidRequestException"
    nextPage="CustomError" />
</exceptionTransitionConfiguration>
```

リスト 3 エラー画面遷移管理ファイル記述例

◆ IISの設定

IIS にログを出力するためには、Web 構成ファイルでのエラー画面遷移ログ出力フラグを有効 (logging="on")にする以外に、IIS の設定が必要である。以下の設定を行うことで、IIS で例外ログを出力することができる。

- Web サーバの管理ツールで、拡張ログオプション”URI クエリ”にチェックを付ける。

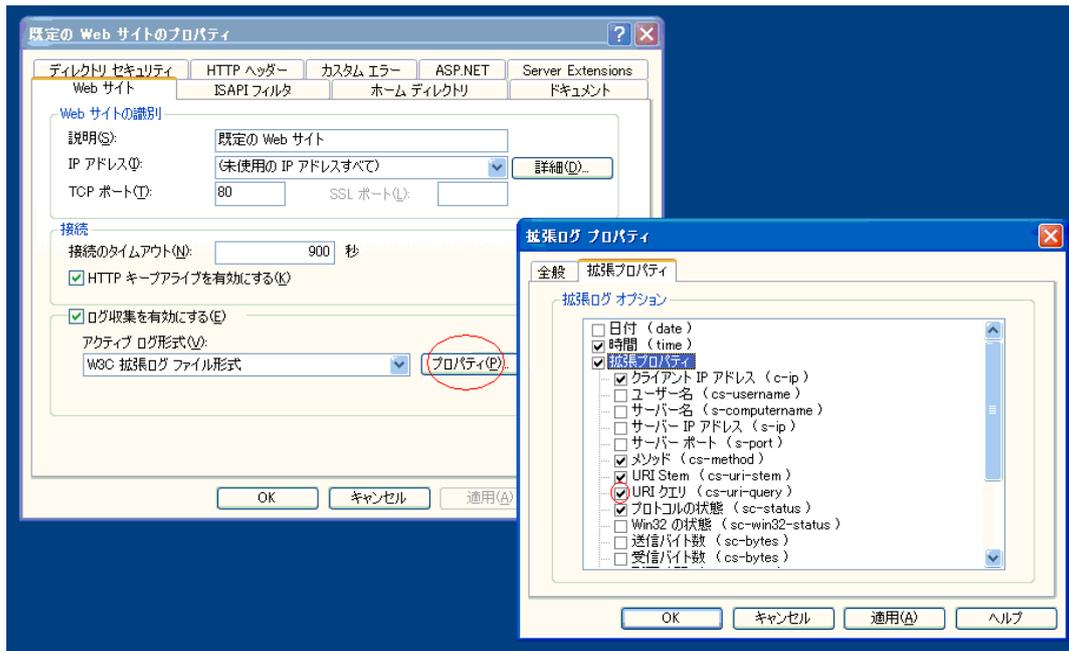


図 2 IIS 設定画面

```
#Software: Microsoft Internet Information Services 5.0
#Version: 1.0
#Date: 2006-03-29 04:40:52
#Fields: date time c-ip cs-method cs-uri-stem cs-uri-query sc-status sc-bytes cs-bytes time-taken
2006-03-29 04:40:52 127.0.0.1 GET
  /WebSI_AP01/ExceptionTransitionListerImpl/Main.aspx - 200 1334 330 5187
2006-03-29 04:40:54 127.0.0.1 POST
  /WebSI_AP01/ExceptionTransitionListerImpl/Main.aspx
+Source+:App_Web_mfblpop;
+Message+:+値が有効な範囲にありません。;
+StackTrace+:+++
+場所+Main.ButtonThrowArgumentEx_Click(Object+sender,+EventArgs+e)++++
+場所+System.Web.UI.WebControls.Button.OnClick(EventArgs+e)++++
(中略)
2006-03-29 04:40:54 127.0.0.1 GET
  /WebSI_AP01/ExceptionTransitionListerImpl/ExTranErrorPage.aspx
  __Token=9673d8c671e644a4a28fd732ef637ac5 200 1523 646 120
```

リスト 4 ログ出力例

■ 関連機能

- WA-01 画面遷移管理機能

WB-01 リクエストコントローラ機能

■ 概要

本機能は、クライアントからのリクエスト要求に対応する入力値検証、ビジネスロジックを実行し、処理結果をクライアントへ送信する機能である。

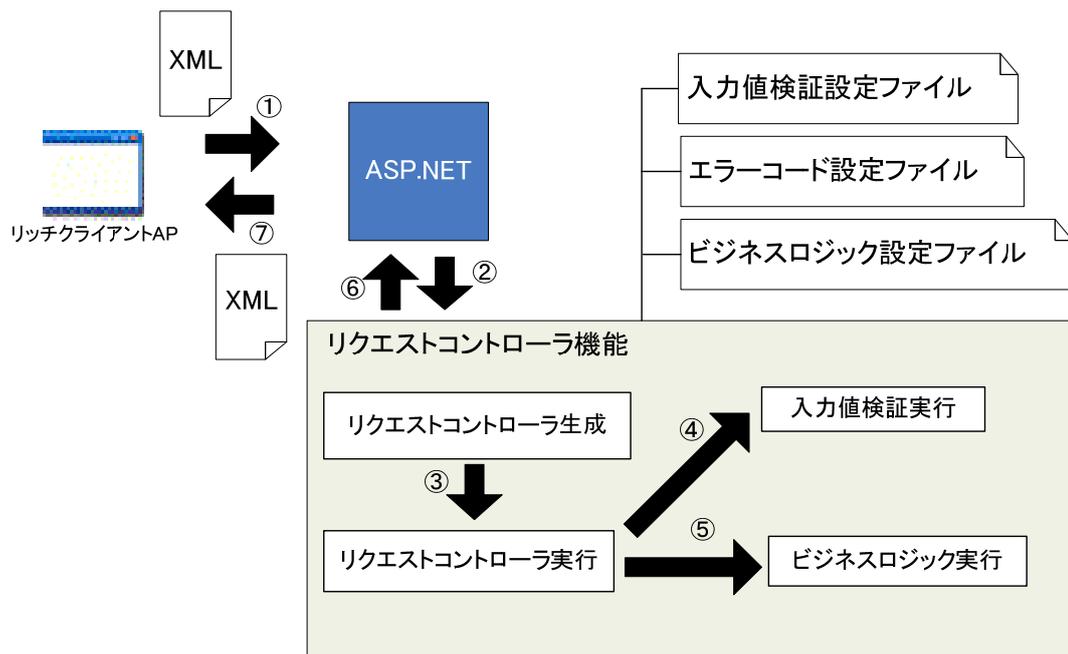


図 1 リクエストコントローラ機能

クライアントから送信されるリクエスト情報に基づいて、実行するリクエストコントローラが生成される。生成されたリクエストコントローラが入力値検証、ビジネスロジックを実行する。リクエストコントローラで例外が発生した場合、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、入力値検証は、TERASOLUNA で提供している入力値検証機能、ビジネスロジックは TERASOLUNA で提供しているビジネスロジック生成機能を利用する。入力値検証機能、ビジネスロジック機能の詳細な内容は、「CM-02 入力値検証機能」、「CM-04 ビジネスロジック生成機能」を参照すること。本機能で説明を行うリクエストコントローラ機能は、XML を送受信するリクエストコントローラである。クライアントがファイル送信する処理に対応するリクエストコントローラは、「WB-02 ファイルアップロード機能」、ファイル受信する処理に対応するリクエストコントローラは、「WB-03 ファイルダウンロード機能」を参照すること。

■ 使用方法

リクエストコントローラ機能を使う上で、必要な Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイル、入力値検証設定ファイル、ビジネスロジックの実装方法について解説する。

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に IHttpHandlerFactory およびビジネスロジック設定ファイルのパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name	○	構成要素名。 固定値、以下を指定する。 ・ blogicConfiguration ・ exceptionCodeConfiguration
	type	○	構成設定の処理を行う構成セクション ハンドラクラス名。以下の固定値を指定。 ・ TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection, TERASOLUNA.Fw.Common ・ TERASOLUNA.Fw.Web.Configuration.ExceptionCode.ExceptionCodeConfigurationSection, TERASOLUNA.Fw.Web
/configuration/blogicConfiguration/files/file			複数可
	path	○	ビジネスロジック設定ファイルパス。
/configuration/exceptionCodeConfiguration/files/file			複数可
	path	○	エラーコード設定ファイルパス。
/configuration/system.web/httpHandlers/add			複数可
	verb	○	受け付けるリクエストの HTTP メソッド。以下を指定する。 POST
	path	○	リクエストを受け付ける URL となるパス。アプリケーションルートからのパスを設定する。
	type	○	リクエストを受け付ける HTTP ハンドラクラスのアセンブリ修飾名。TERASOLUNA で提供しているリクエストコントローラをそのまま利用する場合は、以下を指定。 TERASOLUNA.Fw.Web.Controller.RequestControllerFactory, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="blogicConfiguration"
      type="TERASOLUNA. Fw. Common. Configuration. BLogic. BLogicConfigurationSection,
        TERASOLUNA. Fw. Common"/>
    <section name="exceptionCodeConfiguration"
      type="TERASOLUNA. Fw. Web. Configuration. ExceptionCode. ExceptionCodeConfigurati
        onSection, TERASOLUNA. Fw. Web"/>
  </configSections>
  <blogicConfiguration>
    <files>
      <file path="Config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>
  <exceptionCodeConfiguration>
    <files>
      <file path="Config¥ErrorConfiguration.config"/>
    </files>
  </exceptionCodeConfiguration>
  <system.web>
    <httpHandlers>
      <add verb="POST" path="RequestController.aspx"
        type="TERASOLUNA. Fw. Web. Controller. RequestControllerFactory,
          TERASOLUNA. Fw. Web"/>
    </httpHandlers>
  </system.web>
</configuration>
```

受け付けるリクエストの HTTP メソッドとして、POST を指定。

リクエストを受け付ける URL を指定。
(アプリケーションルートからの相対パス)

リスト 1 Web 構成ファイル記述例

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルに、IBLogic インタフェースを実装するクラスを設定する。

表 2 ビジネスロジック設定ファイル

ノード	属性	必須	値
/blogicConfiguration	xmlns	○	XML スキーマの名前空間。 以下の固定値を指定。 http://www.terasoluna.jp/schema/BL ogicSchema.xsd
/blogicConfiguration/blogic			複数可
	name	○	ビジネスロジック名。重複不可。
	type	○	IBLogic インタフェースを実装したク ラス名と、クラスが含まれるアセンブ リ名。

```
<?xml version="1.0" encoding="utf-8" ?>  
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">  
  <blogic name="sampleA" type="SampleProject.Sample.SampleA, SampleProject" />  
  <blogic name="sampleB" type="SampleProject.Sample.SampleB, SampleProject" />  
</blogicConfiguration>
```

リスト 2 ビジネスロジック設定ファイル記述例

◆ エラーコード設定ファイル

エラーコード設定ファイルに、例外とエラーコードの対応を設定する。エラーコード設定ファイルに設定した例外クラスまたは例外クラスのサブクラスがリクエストコントローラで発生した場合、例外に対応したエラーコードがクライアントへ XML 形式で送信される。クライアントへ送信される内容は本機能説明書の内部構成を参照すること。

表 3 エラーコード設定ファイル

ノード	属性	必須	値
/exceptionConfiguration/exceptionCode	-	-	例外とエラーコードのマッピングを保持する要素 ※複数可
	exceptionType	○	エラーコードをマッピングする例外の完全クラス名。
	code	○	設定するエラーコード。

```
<?xml version="1.0" encoding="utf-8" ?>
<exceptionCodeConfiguration
  xmlns="http://www.terasoluna.jp/schema/ExceptionCodeSchema.xsd">
  <exceptionCode exceptionType="System.Exception" code="E000001" />
</exceptionCodeConfiguration>
```

リスト 3 エラーコード設定ファイル記述例

◆ 入力値検証設定ファイル

入力値検証設定ファイルの設定方法や入力値検証の動作内容については、機能説明書「CM-02 入力値検証機能」を参照すること。

◆ 実装方法

IBLogic インタフェースを実装し、実装したクラスのクラス属性にリクエストタイプ名、入力データセットタイプ、入力値検証設定ファイルパス、ルールセット名を指定する。

表 3 ビジネスロジックのクラス属性一覧

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	-	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。デフォルト値は、"Normal"となる。 *リクエストタイプ名の種類は本機能説明書の内部構成を参照すること。
	InputDataSetType	○	ビジネスロジックの入力データセットの型。
ValidationInfo	ValidationFilePath	-	入力値検証設定ファイルパス。
	RuleSet	-	ルールセット名。

```
[ControllerInfo (RequestType=RequestTypeNames.NORMAL,
                 InputDataSetType=typeof(SampleDs))]
[ValidationInfo (ValidationFilePath=@"Config¥validation.config",
                RuleSet="Default")]
public class SampleA : ILogic
{
    public BLogicResult Execute (BLogicParam param)
    {
        SampleDs sample = param.ParamData as SampleDs;
        return new BLogicResult (BLogicResult.SUCCESS);
    }
}
```

リスト 4 ILogic インタフェース実装例

■ 内部構成

◆ リクエストコントローラの生成

(1) リクエスト名の取得

クライアントからのリクエストの HTTP ヘッダのキー RequestName の値をビジネスロジック名として取得する。

(2) ビジネスロジッククラスのクラス属性の取得

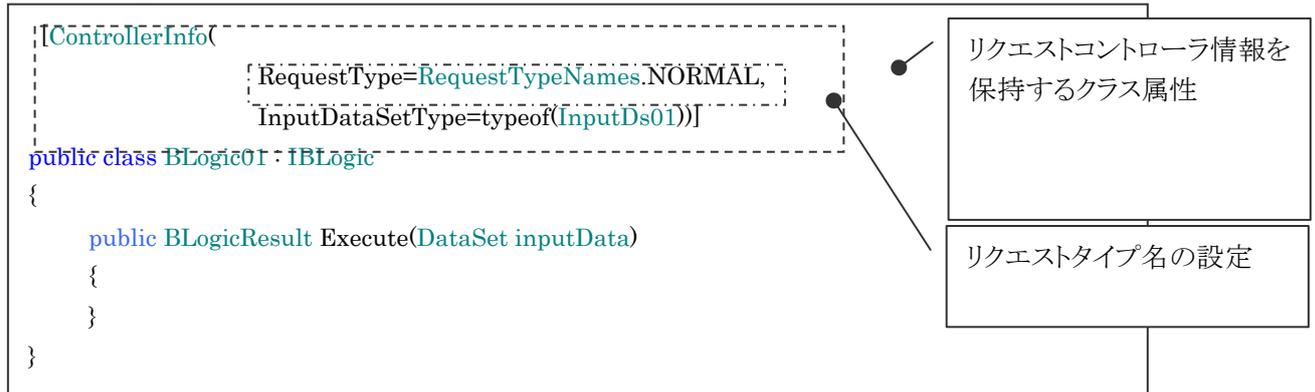
ビジネスロジック生成機能を利用して、ビジネスロジック名からビジネスロジック実装クラスのクラス属性を取得する。

以下の場合、リクエスト不正例外(InvalidRequestException)が発生する。

- HTTP ヘッダに RequestName キーが存在しない場合
- RequestName キーに対応するビジネスロジック名が存在しない場合

(3) リクエストタイプ名の取得

取得したクラス属性からコントローラ情報(ControllerInfo 属性)に設定されたリクエストタイプ名(RequestType)を取得する。



リスト 5 リクエストタイプ名の定義例

(4) リクエストコントローラの生成

取得したリクエストタイプ名に応じたリクエストコントローラを生成する。リクエストコントローラの生成に失敗した場合、TERASOLUNA.Fw.Web.Controller.UnknownRequestController を生成する。リクエストタイプ名とリクエストコントローラの既定での対応を下表に示す。

表 4 リクエストタイプ名とリクエストコントローラの対応

リクエストタイプ名	リクエストコントローラクラス
なし	TERASOLUNA.Fw.Web.Controller.BLogicRequestController,
Normal	TERASOLUNA.Fw.Web
Download	TERASOLUNA.Fw.Web.Controller.FileDownloadRequestController, TERASOLUNA.Fw.Web
Upload	TERASOLUNA.Fw.Web.Controller.FileUploadRequestController, TERASOLUNA.Fw.Web
Multipart Upload	TERASOLUNA.Fw.Web.Controller.MultipartUploadRequestController, TERASOLUNA.Fw.Web
Unknown	TERASOLUNA.Fw.Web.Controller.UnknownRequestController, TERASOLUNA.Fw.Web

◆ リクエストコントローラの実行

リクエストコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、入力値検証、ビジネスロジックの生成と実行、レスポンスの設定を行う。

(1) HTTPヘッダの検証

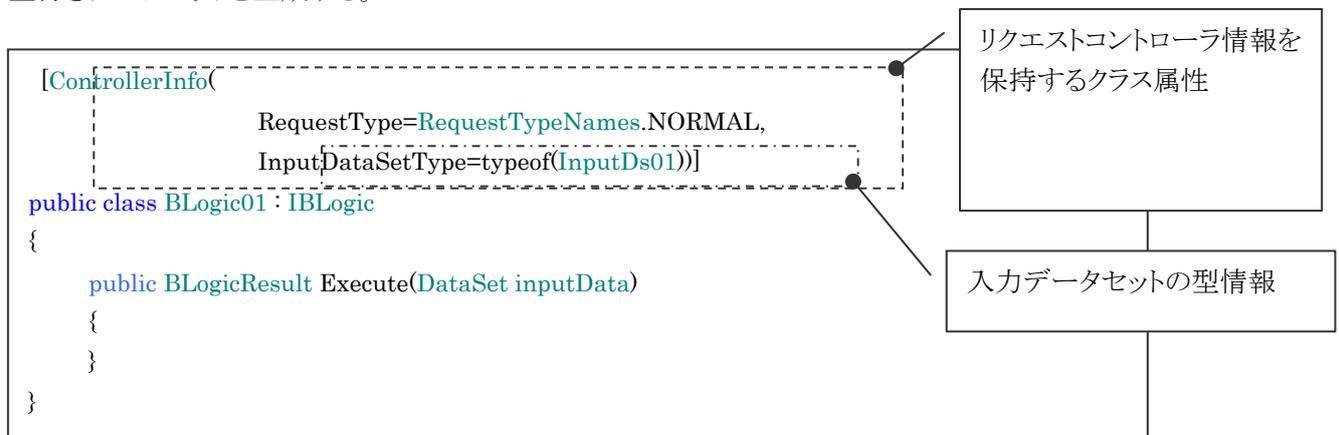
クライアントから送信された Content-Type ヘッダが "text/xml;charset=utf-8" と設定されているか検証する。

以下の場合、リクエスト不正例外(InvalidRequestException)が発生する。

- Content-Type ヘッダがない場合
- Content-Type ヘッダに空文字が設定されている場合
- Content-Type ヘッダのフォーマットが不正の場合
- MediaType が "text/xml" でない場合
- MediaType が "text/xml"であるが、Charset が "utf-8" でない場合

(2) HTTPボディの解析

ビジネスロジッククラスのクラス属性(ControllerInfo)の InputDataSetType に設定された入力データセットの型情報と HTTP ボディの XML データにより、ビジネスロジックに入力値として渡す型付きデータセットを生成する。



リスト 6 入力データセット型の定義例

以下の場合、フレームワーク例外(TerasolunaException)をスローする。

- ビジネスロジックのクラス属性に入力データセットの型情報が設定されていない場合
- 型付きデータセットを生成できなかった場合

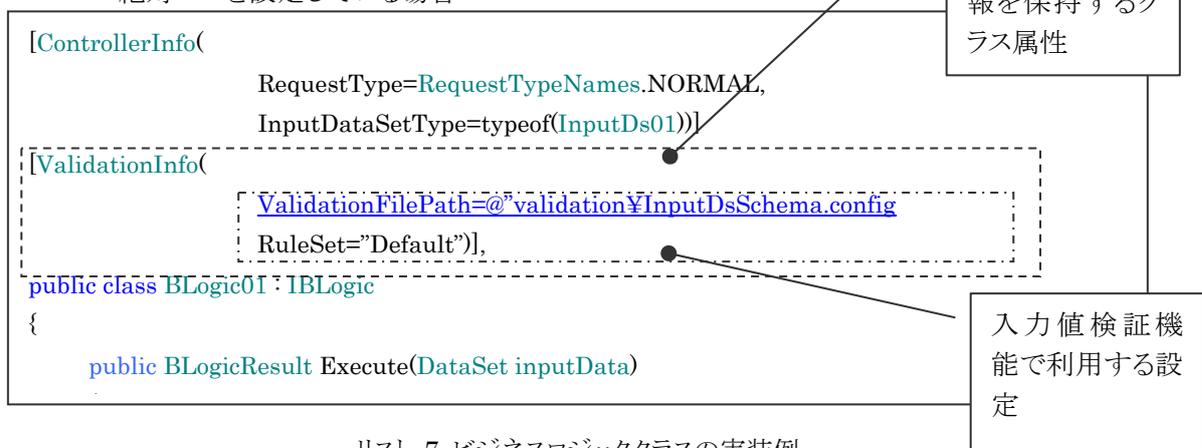
以下の場合、リクエスト不正例外(InvalidRequestException)をスローする。

- HTTP ボディに格納されているデータが XML 形式でない場合
- HTTP ボディに格納されているデータが生成対象データセットのスキーマと合致しない場合

(3) 入力値検証

ビジネスロジッククラスのクラス属性(ValidationInfo)の ValidationFilePath に設定された入力値検証設定ファイルのファイルパスを取得する。RuleSet に設定されたルールセット名を取得する。取得した入力値検証設定ファイルパスとルールセット名を利用して、入力値検証機能を実行する。ビジネスロジッククラスのクラス属性に設定されたファイルパスが以下の場合、フレームワーク例外(TerasolunaException)をスローする。

- 入力値検証設定ファイルが存在しない場合
- ファイル名称としての上限文字数を超過する場合
- ファイル名称として利用できない文字を含む場合
- アプリケーションルート配下のパスでない場合
- 絶対パスを設定している場合



リスト 7 ビジネスロジッククラスの実装例

(4) ビジネスロジックの生成と実行

入力値検証でエラーがない場合、ビジネスロジックを生成して実行する。以下の場合、TerasolunaException をスローする。

- ビジネスロジックをインスタンス化できない場合
- ビジネスロジックが ILogic インタフェースを実装していない場合
- ビジネスロジックの返却値であるビジネスロジック結果オブジェクトが null である場合

(5) レスポンスの設定

クライアントへ返却するレスポンスを書き込む。HTTP ヘッダ、HTTP ボディに格納される情報は、以下の 4 パターンで異なる。

- 実行したビジネスロジックが正常終了した場合
- ビジネスロジックにおいて業務エラーが発生した場合
- システムエラーが発生した場合
- 入力値検証エラーが発生した場合

以下、各パターンにおける HTTP ヘッダ、ならびに HTTP ボディの一覧とサンプルのレスポンス XML を示す。

表 5 サーバ処理終了パターンと返却されるレスポンスの対応表

パターンID	終了パターン	HTTP ヘッダ		HTTP ボディ			
		キー	値	XPath		説明	
①	ビジネスロジック正常終了	content-type	text/xml; charset=utf-8	(省略)			
		status-code	200				
		status-description	OK				
②	業務エラー発生時 ¹	content-type	text/xml; charset=utf-8	<errors			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		exception	ビジネスロジック結果オブジェクトに設定された結果文字列(ResultString)			<error-message>	ビジネスロジック結果オブジェクトに設定されたエラーメッセージ
						<error-code>	ビジネスロジック結果オブジェクトに設定されたエラーメッセージのキー

¹ 業務エラーは、ResultString と Errors を設定したビジネスロジック結果オブジェクトを返却してビジネスロジックを終了した場合を指す。

パターンID	終了パターン	HTTP ヘッダ		HTTP ボディ			
		キー	値	XPath		説明	
③	システムエラー発生時	content-type	text/xml; charset=utf-8	<errors >			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		status-description	OK			<error-message >	発生した例外のメッセージ
		exception	exception			<error-code>	後述するエラーコードマッピング処理で取得したエラーコード ²
④	入力値検証エラー発生時	content-type	text/xml; charset=utf-8	<errors >			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		status-description	OK			<error-message >	入力値検証エラーメッセージ
		exception	validateException			<error-code>	入力値検証エラーエラーコード
						<error-field>	入力値検証エラー発生箇所

² エラーコードマッピング機能を利用しない、エラーコードを取得できない場合には、"E_UNKNOWN_EXCEPTION" が設定される

```
<?xml version="1.0" encoding="utf-8" ?>
<OutputDS01>
  <User>
    <Name>山田太郎</Name>
    <Age>1</Age>
  </User>
</OutputDS01>
```

リスト 8 ビジネスロジック正常終了時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>既に登録された会員です。名前：山田太郎
  </error-message>
    <error-code>MULTIPLE_REGISTER</error-code>
  </error>
</errors>
```

リスト 9 業務エラー時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>DBに接続できません。</error-message>
    <error-code>DB_DISCONNECTED</error-code>
  </error>
</errors>
```

リスト 10 システムエラー時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>文字列カラムの形式が正しくありません。
  </error-message>
    <error-code>PATTERN</error-code>
    <error-field>User[1]/Name</error-field>
  </error>
</errors>
```

リスト 11 入力値検証エラー時のサンプルレスポンス電文

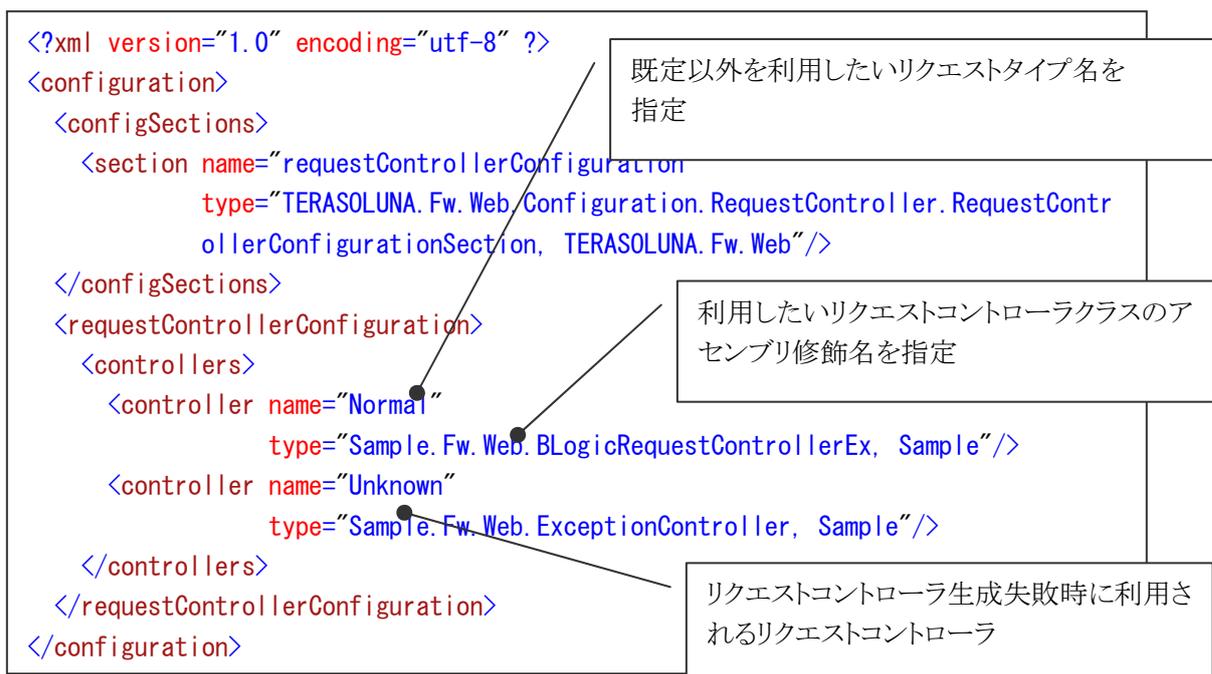
■ 拡張ポイント

◆ 既定リクエストタイプ名に対応するリクエストコントローラを変更する

既定リクエストタイプ名(Normal, Upload, Download, Unknown)のリクエストコントローラを変更する場合、Web 構成ファイル(web.config)に既定のリクエストタイプ名とリクエストコントローラのアセンブリ修飾名を定義する。

表 6 リクエストコントローラの設定

ノード	属性	必須	値
/configuration/configSections/section	name	-	構成要素名。 固定値、以下を指定する。 requestControllerConfiguration
	type	-	構成設定の処理を行う構成セクションハンドラクラス名。以下の固定値を指定。 TERASOLUNA.Fw.Web.Configuration.RequestController, TERASOLUNA.Fw.Web
/configuration/requestControllerConfiguration/controllers/controller		○	複数可
	name	○	リクエストタイプ名。
	type	○	リクエストタイプ名に紐付くリクエストコントローラのアセンブリ修飾名。



リスト 12 Web 構成ファイル(リクエストタイプ名の設定例)

◆ 新規リクエストタイプ名に対応するリクエストコントローラを追加する

前述の「既定リクエストタイプ名のリクエストコントローラを変更する場合」と同様に、Web 構成ファイルに新規追加を行いたいリクエストタイプ名とリクエストコントローラのアセンブリ修飾名を設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <requestControllerConfiguration>
    <controllers>
      <controller name="Rich"
        type="Sample.Fw.Web.RichController, Sample"/>
    </controllers>
  </requestControllerConfiguration>
</configuration>
```

リスト 13 Web 構成ファイルの設定例(独自リクエストタイプ名)

新規に追加したリクエストタイプ名を利用するビジネスロジックの実装例を以下に示す。

```
[ControllerInfo(  
    RequestType="Rich",  
    InputDataSetType=typeof(InputDs01))]  
public class BLogic01 : ILogic  
{  
    public BLogicResult Execute(DataSet inputData)  
    {  
    }  
}
```

新規追加した
リクエストタイプ名を設定

リスト 14 ビジネスロジック実装例(独自リクエストタイプ名)

◆ エラー時にクライアントに返却する電文フォーマットを変更する

(1) ルートノードを変更する場合

ルートノード(既定では、“error”)を変更する場合、フレームワークが提供する BLogicRequestController の CreateInitializeErrorDocument メソッドをオーバーライドし、任意に実装を行う。

(2) ルートノード配下のノード名を変更する場合

ルートノード配下のノード名(既定では、“error”, “error-message”, “error-code”)を変更する場合、BLogicRequestController の以下のメソッドをオーバーライドし、任意に実装を行う。

表 7 エラー電文を作成するメソッド一覧

メソッド	呼び出される契機
WriteBLogicErrorResponseBody	業務エラー
WriteValidationErrorResponseBody	入力値検証エラー時
WriteSystemErrorResponseBody	システムエラー時

◆ クライアントに返却するHTTPヘッダに任意の値を追加する

BLogicRequestController の以下のメソッドをオーバーライドし、任意に実装を行う。

表 8 ヘッダを作成するメソッド一覧

メソッド	呼び出される契機
WriteSuccessXmlResponseHeader	業務正常終了時
WriteBLogicErrorResponseHeader	業務エラー時
WriteValidationErrorResponseHeader	入力値検証エラー時
WriteSystemErrorResponseHeader	システムエラー時

■ 関連機能

- CM-02 入力値検証機能
- CM-04 ビジネスロジック生成機能
- WB-02 ファイルアップロード機能
- WB-03 ファイルダウンロード機能

WB-02 ファイルアップロード機能

■ 概要

本機能は、クライアントからのファイルアップロード要求に対応するファイルの受信処理及びビジネスロジックを実行し、クライアントへ XML でファイルアップロード結果を通知する機能である。クライアントからのファイルアップロード要求に対応できる形式は、マルチパート形式とバイナリ形式の2種類である。

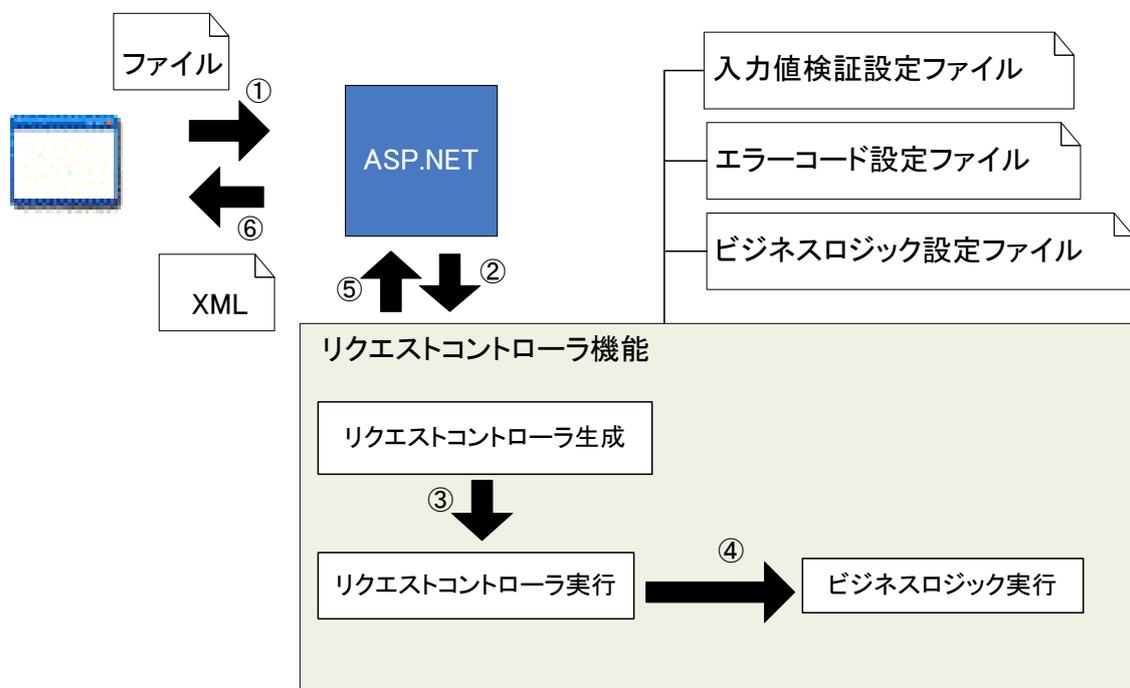


図 1 ファイルアップロード機能

クライアントから送信するファイルアップロード要求に基づいて、実行するリクエストコントローラを生成する。生成したリクエストコントローラがファイルの受信処理およびビジネスロジックを実行する。リクエストコントローラの処理の途中で例外が発生した場合、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、ビジネスロジックは Terasoluna で提供しているビジネスロジック生成機能を利用する。ビジネスロジック機能の詳細な内容は、「CM-4 ビジネスロジック生成機能」を参照すること。本機能で説明するリクエストコントローラ機能は、クライアントからのファイルアップロード要求に対してファイルを受信するリクエストコントローラである。クライアントとサーバで XML を送受信する処理に対応するリクエストコントローラは、「WB-01 リクエストコントローラ機能」、クライアントからのファイルダウンロード要求に対して、ファイルを送信するリクエストコントローラは、「WB-03 ファイルダウンロード機能」を参照すること。

■ 使用方法(マルチパート形式)

Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルアップロード機能のマルチパート形式を利用する場合に必要なビジネスロジックの実装方法について説明する。

◆ 実装方法

ビジネスロジックは必ず `IUploadBLogic` インタフェースを実装する。`IUploadBLogic` 実装クラスのクラス属性に必ずリクエストタイプ名”`MultipartUplaod`”を指定する(表 1)。クライアントからマルチパート形式で送信したデータは、`IUploadBLogic` で実装する `MultipartItemList` プロパティで取得することができる。

表 1 ビジネスロジックのクラス属性一覧(マルチパート形式)

クラス属性	パラメータ名	必須	内容
<code>ControllerInfo</code>	<code>RequestTypeName</code>	○	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。 以下の固定地を指定。 <code>MultipartUpload</code>

IUploadBLogic を実装する。

```
[ControllerInfo(RequestType = RequestTypeNames.MULTIPART_UPLOAD)]
public class BLogic01 : IUploadBLogic
{
    private IDictionary<string, IMultipartItem> _multipartItemList =
        new Dictionary<string, IMultipartItem>();

    public IDictionary<string, IMultipartItem> MultipartItemList
    {
        get
        {
            return _multipartItemList;
        }
        set
        {
            _multipartItemList = value;
        }
    }

    public BLogicResult Execute(BLogicParam param)
    {
        foreach (IMultipartItem item in _multipartItemList)
        {
            if (!item.IsText)
            {
                MultipartFileItem fileItem = item as MultipartFileItem;
                using (FileStream stream = new FileStream(
                    @"C:\temp¥" + fileItem.FileName, FileMode.Create))
                {
                    byte[] buffer = new byte[256];
                    int i = 0;
                    while (fileItem.OutputStream.Read(buffer, 0, 256) != 0)
                    {
                        stream.Write(buffer, 0, 256);
                    }
                    stream.Flush();
                }
            }
        }
        return new BLogicResult(BLogicResult.SUCCESS, new DataSet());
    }
}
```

呼び出し元のリクエストコントローラに公開する設定情報 RequestType には RequestTypeName.MULTIPART_UPLOAD を指定する。

マルチパートアップローデータを設定・取得するプロパティを実装する。

マルチパートアップロードリクエストコントローラが、リクエストを受け付けるたびに新しいインスタンスを設定する。

マルチパートデータがテキストデータかどうかチェック。

リスト 1 ビジネスロジックの実装例(マルチパート形式)

■ 使用方法(バイナリ形式)

ビジネスロジック設定ファイル、エラーコード設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルアップロード機能のバイナリ形式を利用する場合に必要な Web 構成ファイル、ビジネスロジックの実装方法について説明する。

◆ Web構成ファイル

本機能のバイナリ形式アップロードを利用する場合、Web 構成ファイル(web.config)にバイナリアップロード時の一時保存ルートディレクトリの指定及びアップロードファイルの最大サイズを定義することができる。一時保存ルートディレクトリを定義しない場合、システムの一部フォルダを利用する。また、アップロードファイルの最大サイズを定義しない場合、アップロードファイルの最大サイズは 4MB となる。

表 2 Web 構成ファイル

ノード	属性	必須	値
/configuration/appSettings /add	key	-	キー名。
	value	-	固定値、以下を指定する。 “RootTmpDirectory”
/configuration/system.web/ httpRuntime	maxRequestLength	-	一時保存ルートディレクトリのディレクトリパスを絶対パスで指定する。省略した場合は、現在のシステムの一部フォルダのパスを利用する。
			アップロードファイルの最大サイズ(KB yte)を数値型で指定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
...
<appSettings>
  <!-- バイナリアップロード時の一時フォルダの指定 -->
  <add key="RootTmpDirectory" value="C:¥MyFolder¥CustomTemp"/>
</appSettings>
...
<system.web>
  <httpRuntime maxRequestLength="32768"/>
</system.web>
...
```

リスト 2 Web 構成ファイルの記述例

◆ 実装方法

ビジネスロジックは必ず **IBLogic** インタフェースを実装する。**IBLogic** 実装クラスのクラス属性に必ずリクエストタイプ名”Upload”を指定する(表 3)。**IBLogic** 実装クラスの入力データセット(**BLogicParam** の **ParamData** プロパティ)は必ず **TERASOLUNA** で提供しているファイルアップロード用データセット(**UploadFileInfo** クラス)となる(表 4)。バイナリ形式でアップロードしたデータのファイルパスは、ファイルアップロード用データセット(**UploadFileInfo** クラス)の **TempFiles** テーブルの **AbsolutePath** データカラムから取得することができる(表 4)。

表 3 ビジネスロジックのクラス属性一覧(バイナリ形式)

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	○	ビジネスロジックを実行するリクエストコントローラ情報。 以下の固定地を指定。 Upload を指定。

表 4 ファイルアップロード用データセットのテーブル一覧

データテーブル名	データカラム名	データ型	説明
ApplicationRoot	AbsolutePath	System.String	アプリケーションルートの絶対パス。
TempFiles	AbsolutePath	System.String	一時保存ディレクトリに保存したアップロードファイルの絶対パス。

```
[ControllerInfo(RequestType = RequestTypeNames.UPLOAD)]
public class BLogic01 : ILogic
{
    public BLogicResult Execute(BLogicParam blogicParam)
    {
        // 入力データセットのキャスト
        UploadFileInfo inputDs = (UploadFileInfo)blogicParam.ParamDs;
        // アプリケーションルートの絶対パスを取得
        string appRootPath = inputDs.ApplicationRoot[0].AbsolutePath;
        // 一時保存されているアップロードファイルのパスを取得
        string tempFilePath = inputDs.TempFiles[0].AbsolutePath;
        // 一時保存されているアップロードファイルのファイル情報を作成
        FileInfo tempFileInfo = new FileInfo(tempFilePath);

        // 一時保存されているアップロードファイルをアップロードファイル用
        // ディレクトリに移動
        File.Move(tempFilePath, appRootPath + @"¥Upload¥"
                                                         + tempFileInfo.Name);

        // ビジネスロジック結果オブジェクトの生成
        return new BLogicResult(BLogicResult.SUCCESS, new DataSet());
    }
}
```

呼び出し元のリクエストコントローラに公開する設定情報
RequestType には RequestTypeName.UPLOAD を指定する。

入力データセットは、UploadFileInfo となる。

リスト 3 ビジネスロジックの実装例(バイナリ形式)

■ 内部構成(マルチパート形式)

◆ リクエストコントローラの生成

マルチパートアップロードコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

マルチパートアップロードコントローラは、HTTP ヘッダの解析、HTTP ボディの解析、ビジネスロジックの生成と実行、レスポンスの生成を行う。

(1) HTTPヘッダの解析

Content-Type ヘッダから boundary 属性の値と、charset 属性の値を取得する (表 5)。

表 5 HTTP ヘッダから取得する情報一覧

取得する情報	内容
boundary	マルチパートアップロードデータのパートごとの区切りを表す文字列。
charset	マルチパートアップロードデータをエンコードした際に使用する文字コード。

以下の場合、マルチパートアップロードコントローラは `InvalidRequestException` をスローする。

- boundary 属性がない場合
- charset 属性の値が不正な場合

(2) HTTPボディの解析

複数のアップロードデータが存在する場合、それぞれのアップロードデータに対して以下の処理を行う。

各パートのヘッダから、`Content-Disposition`、`Content-Type` を取得する(表 6)。

表 6 取得するヘッダ情報一覧

取得するヘッダ情報		内容
Content- Disposition	name	マルチパート要素名。
	filename (アップロードデータがファイルである場合)	クライアント側で送信するファイルの名前。
Content-Type		パートごとのアップロードデータの形式。

以下の場合、マルチパートアップロードコントローラは `InvalidRequestException` をスローする。

- マルチパート要素名が重複している場合
- `Content-Disposition` ヘッダがない場合
- `Content-Type` ヘッダがない場合
- アップロードデータがファイルで、`filename` 属性がない場合

各パートの `Content-Type` ヘッダが、`"application/x-www-form-urlencoded"` の場合はアップロードデータがテキストであるため、`MultipartTextItem` クラス(表 8)のインスタンスを生成する。`"application/octet-stream"` の場合はアップロードデータがファイルであるため、`MultipartFileItem` クラス(表 9)のインスタンスを生成し、アップロードデータの一時ファイルを生成する。一時ファイルは、ガーベジコレクションが `MultipartFileItem` インスタンスを破棄するタイミングで削除する。

表 7 Content-Type によって作成される MultipartItem クラス一覧

Content-Type の値	格納する MultipartItem クラス
application/x-www-form-urlencoded	MultipartFormItem クラス。
application/octet-stream	MultipartFileItem クラス。

(3) ビジネスロジックの生成と実行

呼び出し対象ビジネスロジックの型を取得し、インスタンス化する。

以下の場合、マルチパートアップロードコントローラは `TerasolunaException` をスローする。

- 呼び出し対象ビジネスロジックをインスタンス化できない場合
- 呼び出し対象ビジネスロジックが `IUploadBLogic` インタフェースを実装していない場合

生成した `IUpload` 実装インスタンスの `MultipartItemList` プロパティに生成した `MultipartItem` のリストを設定し、`IUpload` 実装インスタンスを実行する。

(4) レスポンスの設定

マルチパートアップロードコントローラが実行するレスポンスの設定は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

■ 内部構成(バイナリ形式)

◆ リクエストコントローラの生成

ファイルアップロードコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

ファイルアップロードコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、ビジネスロジックの生成と実行、レスポンスの生成を行う。

(1) HTTPヘッダの検証

Content-Type ヘッダに”application/octet-stream”が適切に設定しているか検証する。

以下の場合、ファイルアップロードコントローラは `InvalidRequestException` をスローする。

- Content-Type ヘッダがない場合
- Content-Type ヘッダに空文字列を設定している場合
- Content-Type ヘッダのフォーマットが不正な場合
 - MediaType が”application/octet-stream”でない場合

Content-Disposition ヘッダに入っている文字列をデコードし、ファイル名称として取得する。取得したファイル名称を HTTP コンテキストに設定する。

以下の場合、ファイルアップロードコントローラは `InvalidRequestException` をスローする。

- Content-Disposition ヘッダがない場合
- Content-Disposition ヘッダに空文字列を設定している場合

- Content-Disposition ヘッダのフォーマットが不正の場合
 - Base64 でエンコードしていない場合
 - Base64 でエンコードしたファイル名称の長さが 4 の倍数でない場合
- Content-Disposition ヘッダの値のフォーマットが不正等の理由でファイル名称を取得できなかった場合

(2) HTTPボディの解析

① アップロードファイルの一時保存ルートディレクトリの生成

Web 構成ファイル(web.config)に一時保存ルートディレクトリのパスを指定する。Web 構成ファイルに一時保存ルートディレクトリのパスを指定しない場合は、システムの一時的フォルダを利用する。

Web 構成ファイルの /configuration/appSettings/add 要素の key 属性に RootTmpDirectory、value 属性に一時保存ルートディレクトリの絶対パスを設定する。

```
<configuration>
  <appSettings>
    <add key="RootTmpDirectory" value="C:¥MyFolder¥CustomTemp"/>
  </appSettings>
</configuration>
```

リスト 4 一時保存ルートディレクトリの設定例

Web 構成ファイルに設定している一時保存ルートディレクトリパスが以下の場合、ファイルアップロードコントローラが ConfigurationErrorsException をスローする。

- 空文字列を設定している場合
- ディレクトリパスとして設定できるサイズを超過する場合
- ディレクトリ名称として指定できない文字を含む場合

② アップロードファイルの一時保存ディレクトリの生成

アップロードファイルの一時保存ルートディレクトリ直下に乱数で生成した文字列をディレクトリ名としてディレクトリを生成する。

以下の場合、ファイルアップロードコントローラは TerasolunaException をスローする。

- 一時保存ルートディレクトリにディレクトリ作成権限がない場合

③ アップロードファイルの一時保存

アップロードファイルの一時保存ディレクトリにクライアントからバイナリ形式で送信してきたアップロードファイルを保存する。

以下の場合、ファイルアップロードコントローラは TerasolunaException をスローする。

- 一時保存ディレクトリにファイル作成権限がない場合



図 2 Web 構成ファイルに一時保存ルートディレクトリを利用する場合のフォルダ構成

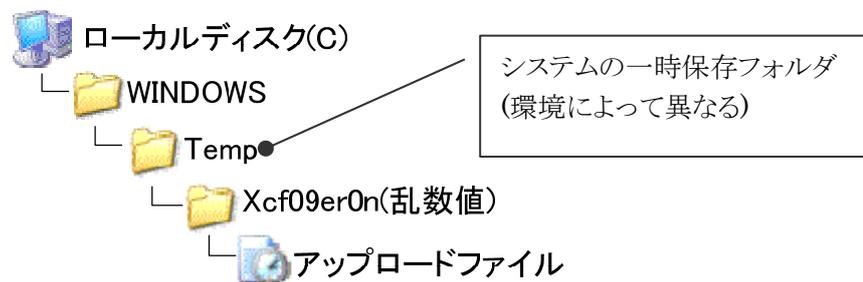


図 3 既定の一時保存ルートディレクトリを利用する場合のフォルダ構成

④ ファイルアップロード用データセットの生成

ビジネスロジックの入力データセットは、ファイルアップロード用データセット (UploadFileInfo インスタンス) を生成し、HTTP コンテキストに設定する。

ファイルアップロード用データセット (UploadFileInfo インスタンス) は DataSet を継承した型付データセットで表 4 のデータテーブルとデータカラムを持つ。

(3) ビジネスロジックの生成と実行

ビジネスロジックを実行する。ビジネスロジックの入力データセットは、HTTP コンテキストで設定したファイルアップロード用データセット (UploadFileInfo インスタンス) である。

以下の場合、リクエストコントローラは TerasolunaException をスローする。

- 呼び出し対象ビジネスロジックをインスタンス化できない場合
- 呼び出し対象ビジネスロジックが ILogic インタフェースを実装していない場合
- ビジネスロジックの返却値であるビジネスロジック結果オブジェクトが null である場合

(4) レスポンスの設定

レスポンスを書き込む前に、一時保存ディレクトリを削除する。

以下の場合、リクエストコントローラは TerasolunaException をスローする。

- 一時保存ルートディレクトリに削除権限がない場合
- 一時保存ルートディレクトリ配下に、現在のプロセス、および他のプロセスで書き込みを禁止している、もしくは開いているファイルが存在する場合

正常に一時保存ディレクトリを削除したのち、リクエストコントローラ機能のレスポンスの設定と同様の処理を実行する。詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

■ 拡張ポイント(バイナリ形式)

◆ アップロードファイル名のデコード方式を変更する

アップロードファイル名のデコード方式(既定では、Base64 エンコーディングによるエンコードによるファイル名のデコーディングを想定)を変更する場合、FileUploadRequestController クラスの DecodeFileName メソッドをオーバーライドし、任意に実装を行う。

■ 関連機能

- CM-04 ビジネスロジック生成機能
- WB-01 リクエストコントローラ機能

WB-03 ファイルダウンロード機能

■ 概要

本機能は、クライアントのリクエスト要求に対応する入力値検証、ビジネスロジックを実行し、クライアントへファイルを送信する機能である。

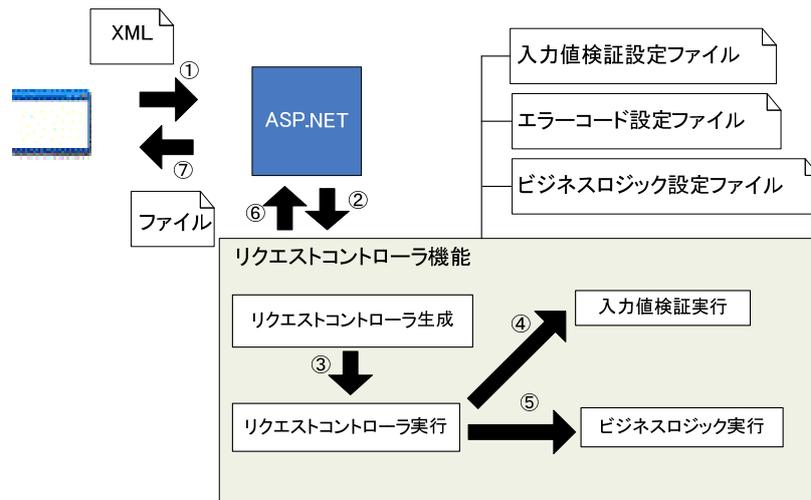


図 1 ファイルダウンロード機能

クライアントから送信されるリクエスト情報に基づいて、実行するリクエストコントローラを生成する。生成されたリクエストコントローラが入力値検証、ビジネスロジックを実行する。ビジネスロジック実行後、リクエストコントローラはビジネスロジックの結果に基づいて、クライアントへファイルをバイナリ形式で送信する。リクエストコントローラの処理の途中で例外が発生した場合は、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、入力値検証は、TERASOLUNA で提供している入力値検証機能、ビジネスロジックは TERASOLUNA で提供しているビジネスロジック生成機能を利用する。入力値検証機能、ビジネスロジック機能の詳細な内容は、「CM-02 入力値検証機能」、「CM-04 ビジネスロジック生成機能」を参照すること。本機能で説明するリクエストコントローラ機能は、クライアントからのファイルダウンロード要求に対して、ファイルを送信するリクエストコントローラである。クライアントとサーバで XML を送受信する処理に対応するリクエストコントローラは、「WB-01 リクエストコントローラ機能」、クライアントからのファイルアップロード要求に対して、ファイルを受信するリクエストコントローラは、「WB-02 ファイルアップロード機能」を参照すること。

■ 使用方法

Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイル、入力値検証設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルダウンロード機能の利用に必要なビジネスロジックの実装方法について説明する。

◆ 実装方法

IBLogic インタフェースを実装し、実装したクラスのクラス属性にリクエストタイプ名、入力データセットタイプ、入力値検証設定ファイルパス、ルールセット名を指定する。必ずリクエストタイプ名には”Download”を指定する(表 1)。返却値はファイルダウンロード用ビジネスロジック結果オブジェクト(FileDownloadBLogicResult 型のインスタンス)とする。ファイルダウンロード用ビジネスロジック結果オブジェクトに設定できるダウンロードファイルデータの形式は、ファイル情報(FileInfo クラス)形式とバイト配列形式の 2 種類がある(表 2)。

表 1 ビジネスロジックのクラス属性一覧

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	○	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。 以下の固定値を指定。 Download
	InputDataSetType	○	ビジネスロジックの入力データセットの型。
ValidationInfo	ValidationFilePath	-	入力値検証設定ファイルパス。
	RuleSet	-	ルールセット名。 デフォルト値は、Default。

表 2 FileDownloadBLogicResult のコンストラクタとその利用観点

コンストラクタ	引数	引数の説明	利用観点
FileDownloadBLogicResult (FileInfo fileInfo)	fileInfo	ダウンロードファイルのパス情報を保持する FileInfo オブジェクト	Web サーバ上にあるファイルをそのままダウンロードする場合
FileDownloadBLogicResult (FileInfo fileInfo, byte[] data)	fileInfo	ダウンロードデータと対応付けるのファイル名を保持する FileInfo オブジェクト	DB やローカルにあるファイルをバイナリ形式で保持しており、そのバイナリ形式のままダウンロードする場合
	data	ダウンロードデータ	



入力値検証機能で
利用する情報を保
持するクラス属性。

```
[ControllerInfo(RequestType = RequestTypeNames.DOWNLOAD,
                 InputDataSetType = typeof(InputDs01))]
[ValidationInfo(ValidationFilePath = @"validation¥InputDs.config")]
public class BLogic01 : IBLogic
```

呼び出し元のリクエストコ
ントローラに公開する設
定情報。
RequestType には
RequestTypeNames.D
OWNLOAD を設定。

入力値検証設定フ
ァイルパス。
例の場合、
[アプリケーションル
ー
ト]¥Bin¥validatio
n¥InputDs.config
を利用する。

ダウンロードファイルのバ
イナリデータを取得。

```
{
    public BLogicResult Execute(BLogicParam blogicParam)
    {
        // 入力データセットのキャスト
        InputDs01 inputDs = (InputDs01)bllogicParam.ParamData;
        // 入力情報の取得
        string name = inputDs.User[0].Name;
        // ダウンロードファイル取得処理
        byte[] downloadData = (取得処理は省略);
        // ビジネスロジック結果オブジェクトの生成
        if (downloadData == null)
        { // 業務エラー時(例: 適切なダウンロードデータがない)
            // エラーメッセージ情報の作成
            List<MessageInfo> msgList = new List<MessageInfo>();
            MessageInfo msgInfo = new MessageInfo(string.Format(
                "存在しないユーザーです。ユーザー名:{0}", ));
```

エラーメッセージ。
(レスポンス電文の
//errors/error/erro-
message に格納される値)

エラーコード。
(レスポンス電文の
//errors/error/erro-code
に格納される値)

```
            msgInfo.Key = "USER_NOT_FOUND";
            msgList.Add(msgInfo);
            return new FileDownloadBLogicResult("blogicError", msgList);
        }
    }
    else
```

結果文字列。
(レスポンス HTTP ヘッ
ダ(キー:exception)に格
納される値)

コンストラクタの第一引数
にファイル名を保持する
FileInfo オブジェクト、第
二引数にダウンロードデ
ータを指定する。

```
{ // 正常時
    // ダウンロードファイル情報生成
    // content-dispositionに設定するファイル名となる
    FileInfo fileName = new FileInfo(name);
    // ファイルダウンロード用ビジネスロジック結果オブジェクトの返却
    return new FileDownloadBLogicResult(fileName, downloadData);
}
```

クライアントへバイナリ送信を
するファイルの名前を指定。

リスト 2 ビジネスロジッククラスの実装例

(ダウンロードファイル情報がバイト配列の形式の場合)

■ 内部構成

◆ リクエストコントローラの生成

ファイルダウンロードリクエストコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

ファイルダウンロードリクエストコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、入力値検証、ビジネスロジックの生成と実行、レスポンスの生成を行う。ファイルダウンロードリクエストコントローラが実行する HTTP ヘッダの検証、HTTP ボディの解析、入力値検証は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ビジネスロジックの生成と実行、レスポンスの設定について解説する。

(4) ビジネスロジックの生成と実行

入力値検証でエラーがない場合、ビジネスロジックを生成して実行する。ビジネスロジックは、ビジネスロジックの実行結果として、ファイルダウンロード用ビジネスロジック結果オブジェクトを返却する。

以下の場合、リクエストコントローラは `TerasolunaException` をスローする。

- ビジネスロジックをインスタンス化できない場合
- ビジネスロジックが `IBLogic` インタフェースを実装していない場合
- ビジネスロジックの返却値であるファイルダウンロード用ビジネスロジック結果オブジェクトが `null` である場合
- ビジネスロジックの返却値であるファイルダウンロード用ビジネスロジック結果オブジェクトに、存在しないファイルの情報 (`FileInfo`) が設定されている場合

(5) レスポンスの設定

クライアントへ返却するレスポンスを書き込む。ビジネスロジック正常終了時に設定される HTTP ヘッダの情報を表 3 に示す。Content-Disposition ヘッダには、ファイル名を Base64 でエンコードされた文字列を設定する。

表 3 設定される HTTP ヘッダの値

パターン ID	終了パターン	HTTP	
		キー	値
①	ビジネスロジック 正常終了	content-type	application/octet-stream
		content-disposition	attachment; filename=?ISO-2022JP?B?{Base64 エンコード済みファイル名}?=
		status-code	200
		status-description	OK

また、HTTP ボディには、ファイルダウンロード用ビジネスロジック結果オブジェクトに設定されているダウンロードデータをレスポンスに書き込み、クライアントにバイナリファイルを送信する。

■ 拡張ポイント

◆ ダウンロードファイル名のエンコード方式を変更する

ダウンロードファイル名のエンコード方式(既定では、Base64 エンコーディング)を変更する場合、FileDownloadRequestController クラスの EncodeFileName メソッドをオーバーライドする。

■ 関連機能

- CM-02 入力値検証機能
- CM-04 ビジネスロジック生成機能
- WB-01 リクエストコントローラ機能

WC-01 セッション管理機能

■ 概要

本機能は、セッション情報をライフサイクルレベルで管理する機能である。ライフサイクルレベルとは、セッション情報をグループ化する単位のことである。ライフサイクルレベルごとにセッション情報を管理することで、セッション情報のグループごとの一括削除を実現する。

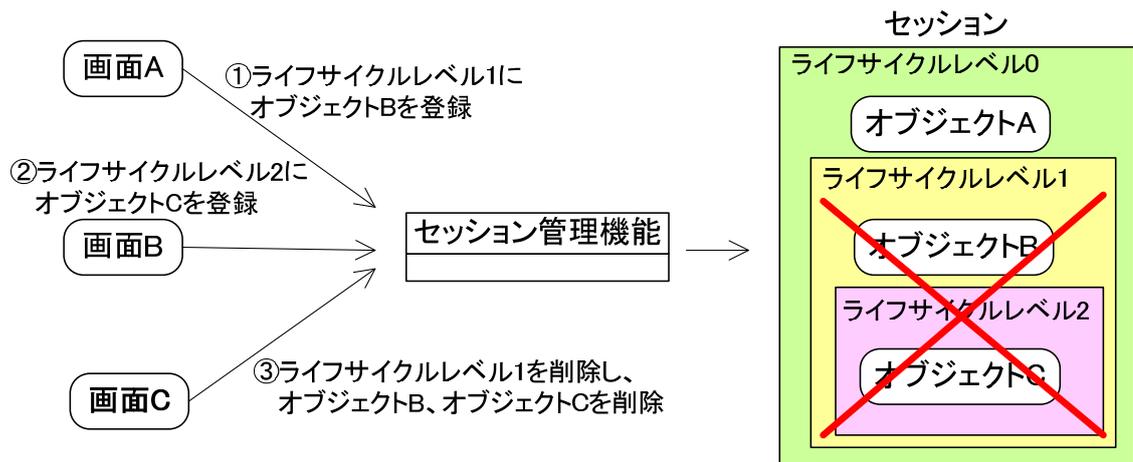


図 1 セッション管理機能

セッション情報をライフサイクルレベルごとの階層構造で管理する。ライフサイクルレベル 0 はライフサイクルレベル 1 を包含し、ライフサイクルレベル 1 はライフサイクルレベル 2 を包含する。ライフサイクルレベルは、以降同様に包含関係を形成する。

セッション内へのオブジェクトの登録は、セッションキーと登録するオブジェクトとライフサイクルレベルを指定する。

セッション内のオブジェクトの削除は、ライフサイクルレベルまたはセッションキーを指定する。ライフサイクルレベル 1 を削除すると、ライフサイクルレベル 2 で保存されているオブジェクトも削除される。このようにセッション情報をグループ単位で削除することで、セッションに登録した個別データの削除漏れを防ぐことができる。

■ 使用方法

◆ 実装方法

セッションにオブジェクトを登録する場合は、セッションキー、登録するオブジェクト、ライフサイクルレベルを指定する。

```
public partial class SC0010 : System.Web.UI.Page
{
    private SessionManager _sessionManager = null;

    protected override void OnInit(EventArgs e)
    {
        base.OnInit(e);
        _sessionManager = new SessionManager(this.Session);
    }

    protected void LoginButton_Click(object sender, EventArgs e)
    {
        // 業務処理 省略

        // ライフサイクルレベル0にログイン情報を登録する
        _sessionManager.Add("ID", TextBoxFirstName.Text, 0);

        // 画面遷移
        Response.Redirect("SC0020.aspx");
    }
}
```

リスト 1 セッションにオブジェクトを登録する実装例

セッション内のオブジェクトをライフサイクルレベルで削除する場合は、ライフサイクルレベルを指定する。単一オブジェクトを削除する場合は、セッションキーを指定する。

```
public partial class SC0010 : System.Web.UI.Page
{
    private SessionManager _sessionManager = null;

    public override void OnInit()
    {
        base.OnInit();
        _sessionManager = new SessionManager(this.Session);
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // 画面初回表示時にライフサイクルレベル1の項目を削除する
            _sessionManager.Remove(1);

            // キー値” ID” で保存されているセッション情報を削除する
            _sessionManager.Remove("ID");
        }
    }
}
```

リスト 2 セッション内のオブジェクトを削除する実装例

セッション内のオブジェクトを取得する場合は、取得するセッションキーをインデックスに指定する。

```
// セッション内のオブジェクトを取得する。
SessionManager sessionManager = new SessionManager(Session);
string id = (string)sessionManager["ID"];
```

リスト 3 セッション内のオブジェクトを取得する実装例

■ 内部構成

◆ セッション情報のライフサイクルレベル管理機能

セッション管理機能では、セッション情報のライフサイクルレベル管理を「レベル管理マスタ」と「キー管理マスタ」の 2 種類のマスタデータによって管理する。レベル管理マスタはライフサイクルレベルごとのセッションキーを管理する。キー管理マスタはセッションキーごとのライフサイクルレベルを管理する。セッション管理機能では、これらのマスタデータを利用し、ライフサイクル管理やセッション情報の一括削除を実現する。

表 1 セッションに保存されるマスタデータ

セッションキー名	型	説明
SESSION_MANAGER_LEVEL_MASTER	Dictionary<int, IList<string>>	レベル管理マスタ。 ライフサイクルごとのセッションキーを管理する。Dictionary の key 値はライフサイクルレベル、value 値は管理対象のセッションキーが格納される。
SESSION_MANAGER_KEY_MASTER	Dictionary<string, int>	キー管理マスタ。 セッションキーごとのライフサイクルレベルを管理する。Dictionary の key 値は管理対象のセッションキー、value 値はライフサイクルレベルが格納される。

(1) セッション登録時の制限事項

SessionManager の Add メソッドで新規にオブジェクトを登録するときに、以下の条件に当てはまる場合は登録できない。

- 新規にセッションに登録しようとするオブジェクトのキー名が、ライフサイクル管理外のセッションで既に登録されていた場合
- Add メソッドで指定するライフサイクルレベルと異なるライフサイクルレベルで既に管理されていた場合

(2) セッション情報の操作に関する注意事項

SessionManager によって登録したオブジェクトやマスタデータは、SessionManager 以外で操作しないこと。SessionManager 以外の手段でこれらのデータを操作した場合、マスタデータで管理している内容と実際のセッションに登録されている項目に相違が生じる可能性がある。このようなことによって生じたマスタデータの矛盾には、セッション管理機能は対処しない。従って、SessionManager を通してセッションに登録したデータは、SessionManager を通して操作すること。

◆ セッション情報のライフサイクル単位での一括削除機能

セッション管理機能では、セッションに登録されているマスタデータをライフサイクル単位での一括削除機能を提供する。一括削除機能では、クリア対象のライフサイクルレベルを指定すると、それに含まれる全てのライフサイクルレベルのセッション情報を削除する。

一括削除機能の利用例をリスト 2 に示す。まず、Add メソッドによって 4 個のオブジェクトをセッションに登録する。登録されたオブジェクトは、図 2 のようにライフサイクル管理される。その後 Remove メソッドでライフサイクルレベル 1 を指定すると、ライフサイクルレベル 1 に含まれる、全てのセッション情報を削除する。つまり、ライフサイクルレベル 1 の object11、ライフサイクルレベル 2 の object12 と object13 の 3 つのオブジェクトが削除される。

```
SessionManager manager = new SessionManager (Session);  
// Add(セッションキー名、セッションに格納するオブジェクト、ライフサイクルレベル)  
manager.Add("key10", object10, 0);  
manager.Add("key11", object11, 1);  
manager.Add("key12", object12, 2);  
manager.Add("key13", object13, 2);  
  
// Remove(ライフサイクルレベル)  
manager.Remove(1);
```

リスト 4 セッション一括削除機能の利用例

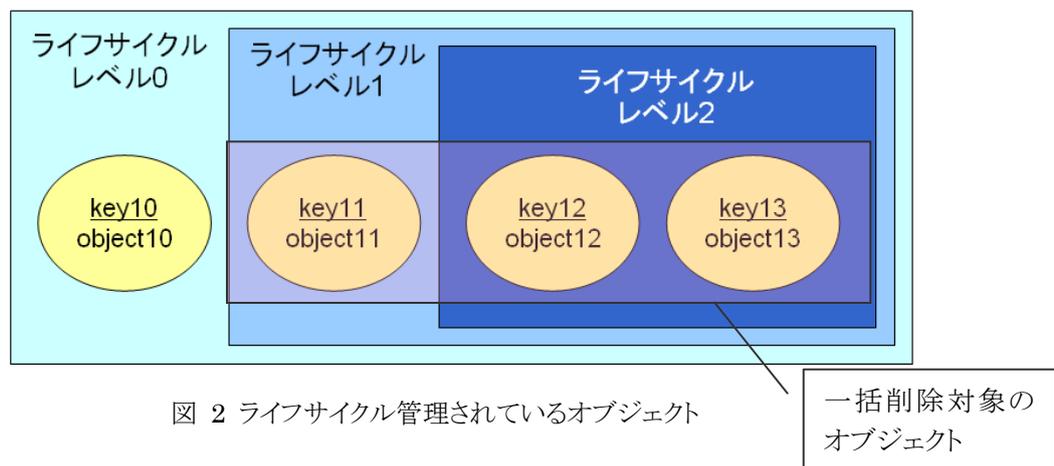


図 2 ライフサイクル管理されているオブジェクト

■ 設計ポイント

ソフトウェアアーキテクトは、ライフサイクルレベル毎のセッション登録情報と、セッションのクリアタイミングについて設計する必要がある。

ライフサイクルレベル毎のセッション登録情報の設計例を表 2 に示す。ここでは、ライフサイクルレベルを業務ごとに定義している。ライフサイクルレベル 0 には、アプリケーション共通で利用する情報を登録する。ライフサイクルレベル 1 には、予約登録・変更・照会などの各ユースケースで利用する情報を登録する。例えば、予約登録情報は、一連の予約登録ユースケースである、予約登録画面から予約登録完了画面、会員メニュー画面に戻るまで保持される。

表 2 ライフサイクルレベル毎のセッション登録情報一覧の例

ライフサイクルレベル	業務	セッション登録情報
レベル 0	-	ログイン情報
レベル 1	予約登録	予約登録情報
	予約変更	予約変更情報
	予約照会	予約照会情報
	-	戻り先画面
	-	エラーメッセージ

セッションのクリアタイミングの設計例を表 3 に示す。ここでは、ユースケースが切り替わる画面遷移のタイミングで、セッション情報をクリアする。具体的に設計すべきことは、どの画面の、どのイベントハンドラで、どのライフサイクルレベルのセッションをクリアするのかという 3 点である。

表 3 セッションのクリアタイミング一覧の例

クリアセッション	クリアタイミング	クリア処理を実装するイベントハンドラ
レベル 0	ログイン画面の初期表示処理	画面#SC0010, PageLoad
	共通エラー画面のトップページへボタン押下処理	画面#SCERR0001, ToTopPageButton_Click
レベル 1	会員メニュー画面の初期表示処理	画面#SC0020, PageLoad

画面遷移図にセッション登録情報とクリアタイミングをマージした例を図 3 に示す。画面遷移図にライフサイクルレベルに関する情報を追記することで、画面とライフサイクルの関係が分かりやすくなる。

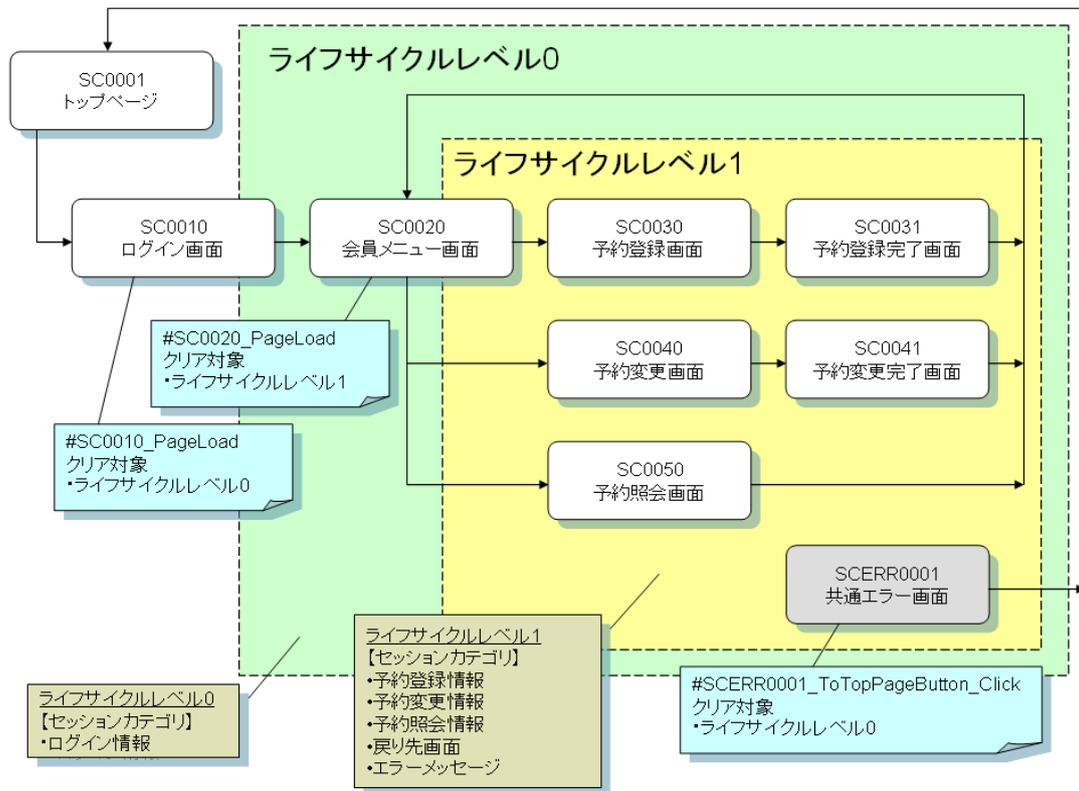


図 3 セッションカテゴリ情報とクリアタイミングを示した画面遷移図

WC-02 SQL文管理機能

■ 概要

本機能は、設定ファイルに定義した SQL 文を取得する機能を提供する。これにより、SQL 文によってパフォーマンスチューニングしたい場合などに、再ビルドを行わずに SQL 文を変更することができる。

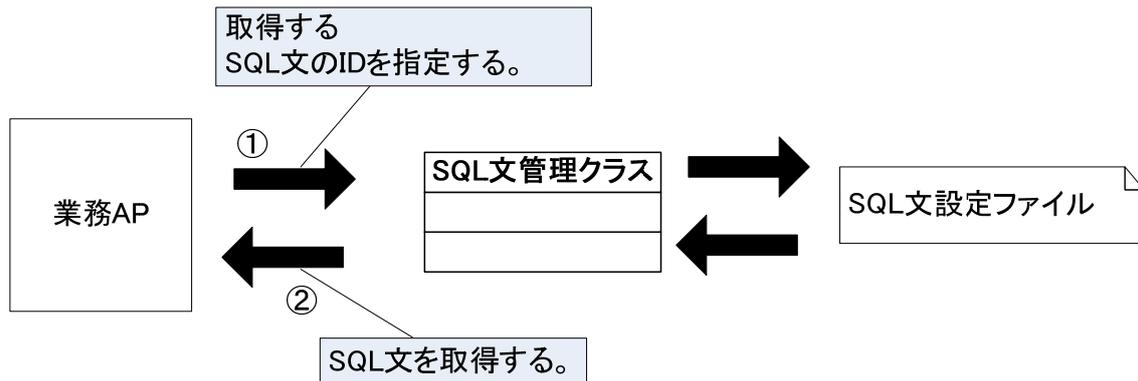


図 1 SQL 文管理機能

業務 AP で利用する SQL 文を設定ファイルに定義し、コードから SQL 文に対応する名前を指定することで、SQL 文を動的に取得することができる機能である。

SQL 文を記述する設定ファイルは、Web 構成ファイル(web.config)に利用する設定ファイルのパスを複数記述することができるため、ユースケースごとに SQL 文設定ファイルを分割することも可能である。

■ 使用方法

◆ Web構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に SQL 文設定ファイルを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name		構成要素名。 固定値。以下の値とする。 sqlConfiguration
	type		構成設定の処理を行う構成セクション ハンドラクラス名。 固定値。以下の値とする。 TERASOLUNA.Fw.Web.Configuration.Sql.SqlConfigurationSection,TERASOLUNA.Fw.Web
/configuration/sqlConfiguration/files/file			複数可
	path	○	SQL 文設定ファイルのアプリケーションルートからのパス。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
...
<configSections>
  <section name="sqlConfiguration"
           type="TERASOLUNA.Fw.Web.Configuration.Sql.SqlConfigurationSection,TERASOLUNA.Fw.Web"/>
</configSections>
<sqlConfiguration>
  <files>
    <file path="Config¥SqlConfiguration01.config"/>
    <file path="Config¥SqlConfiguration02.config"/>
  </files>
</sqlConfiguration>
```

リスト 1 Web 構成ファイル記述例

◆ SQL文設定ファイル

SQL 文設定ファイルにより、SQL 文に対応する ID (SQL ID)と SQL 文を関連付ける。

表 2 SQL 文設定ファイル

ノード	属性	必須	値
/sqlConfiguration	xmlns	○	Namespace 名。 固定値。以下の値とする。 http://www.terasoluna.jp/schema/SqlSchema.xsd
/sqlConfiguration/sql			複数可。
	name	○	SQLID。全ての SQL 文設定ファイルの中で一意になる設定をする。重複不可。

/sqlConfiguration/sql の値として、SQL 文を設定する。記入方法は、XML で利用可能な文字をすべて記述することができる用に、CDATA セクションで囲み記入する。 (“![CDATA[sql 文]”)

```
<?xml version="1.0" encoding="utf-8" ?>
<sqlConfiguration xmlns="http://www.terasoluna.jp/schema/SqlSchema.xsd">
  <sql name="selectCust"><![CDATA[SELECT 1 FROM TABLE]]></sql>
  <sql name="updateCust"><![CDATA[UPDATE TABLE SET COLUMN = 2]]></sql>
  <sql name="insertCust"><![CDATA[INSERT INTO TABLE VALUES (3)]]></sql>
</sqlConfiguration>
```

リスト 2 SQL 文設定ファイル記述例

◆ 実装方法

SQL 文設定ファイルから SQL 文を取得する場合は、SqlConfiguration クラスの GetSql メソッドに SQL ID を指定して実行する。

```
string sql = SqlConfiguration.GetSql("selectCust");
```

リスト 3 実装例

FA-01 画面遷移機能

■ 概要

本機能は、画面遷移設定ファイルに定義した情報に従って画面遷移する機能を提供する。

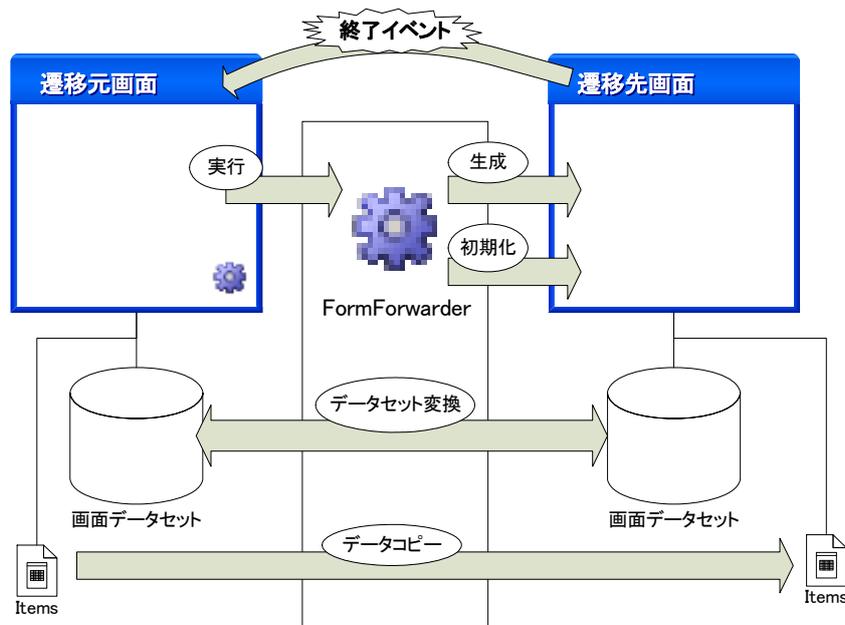


図 1 動作イメージ

画面遷移では **FormForwarder** コンポーネントを利用する。遷移元画面から遷移先画面に画面遷移する場合、**FormForwarder** コンポーネントに遷移先画面の画面 ID を設定する。画面遷移時は、画面遷移設定ファイルから遷移先画面 ID に対応する画面クラスの型を取得し、画面を生成する。このとき、遷移元画面と遷移先画面でのデータの受け渡しも行う。**FormForwarder** コンポーネントに閉じるイベントの設定した場合、遷移先画面を閉じると終了イベントが発生する。

■ 使用方法

◆ アプリケーション構成ファイル

本機能を有効にする場合、アプリケーション構成ファイル(App.config)に画面遷移設定ファイルを定義する。

表 1 アプリケーション構成ファイル

ノード	属性	必須	値
configuration/configSections/section	name	○	構成要素名。 固定値。以下の値とする。 viewConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値。以下の値とする。 TERASOLUNA.Fw.Client.Configuration.View.ViewConfigurationSection,TERASOLUNA.Fw.Client
configuration/viewConfiguration/files/file			複数可
	Path	○	画面遷移設定ファイルのアプリケーションルートからのパス。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="viewConfiguration"
type="TERASOLUNA.Fw.Client.Configuration.View.ViewConfigurationSection,
TERASOLUNA.Fw.Client"/>
  </configSections>
  <viewConfiguration>
    <files>
      <file path="config¥view_maintenance.config"/>
      <file path="config¥view_petshop.config"/>
      <file path="config¥view_foodshop"/>
    </files>
  </viewConfiguration>
</configuration>
```

画面遷移設定ファイルのパスを指定

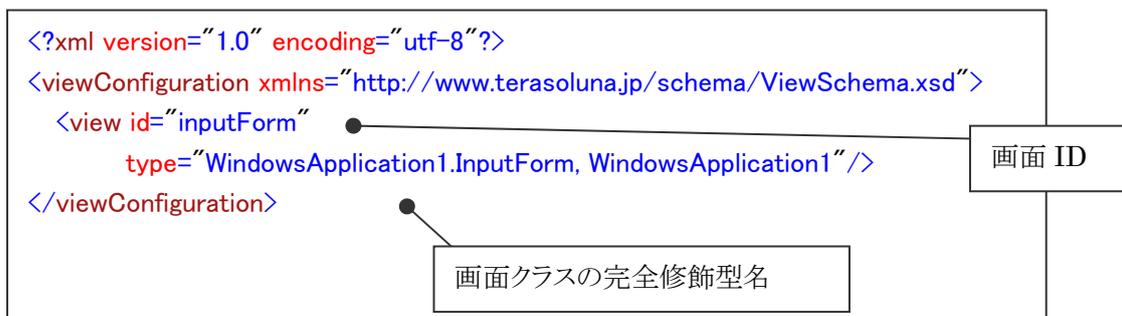
リスト 1 アプリケーション構成ファイル記述例

◆ 画面遷移設定ファイル

表 2 画面遷移設定ファイル

ノード	属性	必須	値
/viewconfiguration	xmles	○	xml スキーマ。 固定値。以下の値とする。 http://www.terasoluna.jp/schema/ViewSchema.xsd
/viewconfiguration/view			複数可
w	id	○	画面 ID
	type	○	画面クラスの完全修飾型名

画面遷移設定ファイルに画面 ID とそれに対応する画面クラスの完全修飾型名を設定する。画面 ID は、全ての画面遷移設定ファイルの中で一意でなければならない。



リスト 2 画面遷移設定ファイル記述例

◆ 実装方法

(1) 画面クラスの実装

画面クラスは Terasoluna Client Framework for .NET で提供する拡張フォーム FormBase を継承して作成する。拡張フォーム FormBase は、画面遷移に必要な IForwardable インターフェイスの標準実装を提供している。

IForwardable インターフェイスで公開する Items プロパティと ViewData プロパティは、遷移元画面と遷移先画面でデータの受け渡しをする際に利用する。

Items プロパティを利用して画面間のデータの受け渡しをする場合は、7ページの「画面間のデータの受け渡し方法」で解説する。

遷移元画面と遷移先画面の ViewData プロパティに設定したデータセット間で、データの受け渡しをする場合は、遷移元画面と遷移先画面の ViewData プロパティへのデータセットの設定が必須となる。さらに、データセット変換機能を利用するためのアプリケーション構成ファイルへの設定と、データセット変換設定ファイルの設定が必要である。データセット変換機能を利用するための設定については、「FB-02 データセット変換機能」の機能説明書を参照のこと。

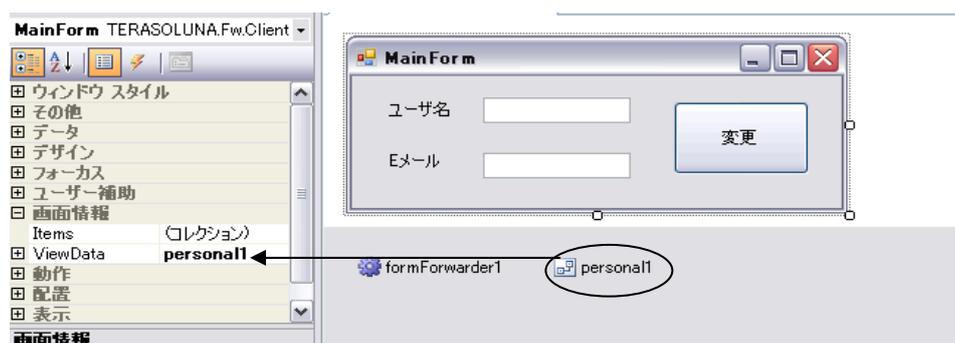


図 2 画面クラスでの ViewData プロパティの設定例

表 1 IForwardable のプロパティ

項番	プロパティ名	説明
1	Items	画面の持つ公開されたコレクションオブジェクト。画面遷移機能で遷移先画面を開くとき、遷移元画面の Items プロパティのキーと値の組が、遷移先画面の Items プロパティにセットされる。
2	ViewData	画面データセット。IForwardable では読み取りプロパティしか定義していない。

(2) FormForwarderコンポーネントの追加とプロパティの設定

FormForwarderコンポーネントを遷移元画面に追加し、画面遷移に必要なプロパティを設定する(図 3)。FormForwarderのプロパティの詳細を表 2、表 3、表 4に示す。

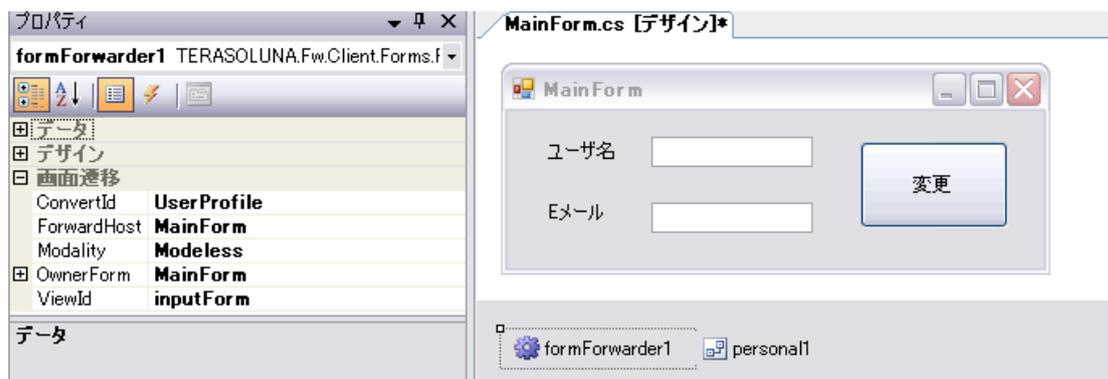


図 3 全てのプロパティを設定した状態

表 2 FormForwarder のプロパティ

項番	プロパティ名	必須	設定値	説明
1	ViewId	○	画面 ID	遷移先画面の画面 ID を設定する。画面遷移設定ファイルに定義した画面 ID に対応する。
2	ConvertId		コンバート ID	コンバート ID は遷移元画面と遷移先画面の ViewData プロパティに設定したデータセット間でデータを受け渡しする場合は設定する。データセット変換設定ファイルに定義したコンバート ID に対応する。
3	Modality	○	Modality 列挙体の値	Modality列挙体の値のいずれかを設定する(表 3)。 初期値: Modality.Modeless
4	OwnerForm		Form インスタンス	遷移先画面の Owner プロパティに設定されるフォーム。OwnerForm プロパティが設定されると、遷移先画面は遷移元画面より背後に表示されることなく、遷移元画面が最小化、閉じるなどのアクションを起こしたとき、遷移先画面も同じ動作をする。
5	ForwardHost		IForwardable 実装クラスのインスタンス	遷移元画面。画面間でデータを受け渡す場合は設定する必要がある。

表 3 FormForwarder.Modality の設定値

項番	値	説明
1	Modeless	画面をモードレスで表示する。親画面と直接の関係を持たない。(ただし、OwnerForm プロパティの影響は受ける)。Execute メソッドは ForwardResult.None を返却する。
2	FormModal	直接の親画面に対してモーダルで画面を表示する。親画面の親画面、同じ親画面から生成された兄弟画面などには影響を与えない。Execute メソッドは ForwardResult.None を返却する。
3	ApplicationModal	画面をアプリケーションモーダルで表示する。遷移先画面が閉じられるまで遷移元画面の実行は再開されない。遷移先画面が閉じた場合、Execute メソッドの戻り値はフォームの持つ DialogResult プロパティに応じた値を持つ ForwardResult 列挙値となる。

表 4 FormForwarder のイベント

項番	イベント名	説明
1	FormClosing	遷移先画面の FormClosing イベントが発生した時に発生するイベント。
2	FormClosed	遷移先画面の FormClosed イベントが発生した時に発生するイベント。

(3) 画面遷移の実行

画面遷移を行う場合、遷移元画面で FormForwarder インスタンスの Execute メソッドを呼び出す。

```
// 遷移元画面での画面遷移実行の例
private void button1_Click(object sender, EventArgs e)
{
    ForwardResult result = formForwarder1.Execute();
    if(result == ForwardResult.Abort)
    {
        // 画面の初期化に失敗
    }
}
```

戻り値によって画面遷移の結果を判断する

リスト 3 画面遷移の記述例

◆ 画面間のデータの受け渡し方法

(1) 遷移元画面から遷移先画面へデータを受け渡す方法

画面遷移をするときに遷移元画面から遷移先画面へデータを受け渡すためには、遷移元画面と遷移先画面のItemsプロパティを利用する。遷移元画面ではItemsプロパティに遷移先画面へ渡すデータを設定する。遷移先画面では、LoadイベントのイベントハンドラなどでItemsプロパティから遷移元画面のデータを受け取る。実装例をリスト 4、リスト 5に示す。

```
// 遷移元画面の実装例
private void button1_Click(object sender, EventArgs e)
{
    // 遷移先画面へ渡すデータをItemsプロパティに設定
    Items["NAME"] = "TERASOLUNA";
    Items["ADDRESS"] = "TOKYO";
    ForwardResult result = formForwarder1.Execute();

    // 省略
}
```

リスト 4 遷移元画面で遷移先画面へデータを渡す例

```
// 遷移先画面の実装例
private string _name = null;
private string _address = null;

private void Form_Load(object sender, EventArgs e)
{
    // 遷移先画面へ渡すデータをItemsプロパティに設定
    _name = Items["NAME"] as string;
    _address = Items["ADDRESS"] as string;
}
```

リスト 5 遷移先画面で遷移元画面のデータを受け取る例

(2) 遷移先画面から遷移元画面へデータを受け渡す方法

遷移先画面が閉じるときに遷移元画面へデータを受け渡すためには、遷移先画面ではItemsプロパティに遷移元画面へ渡すデータを設定する。遷移元画面では、FormForwarderのFormClosedイベント(図 4)で遷移先画面のデータを受け取る。遷移元画面のFormClosedイベントの実装例をリスト 6に示す。

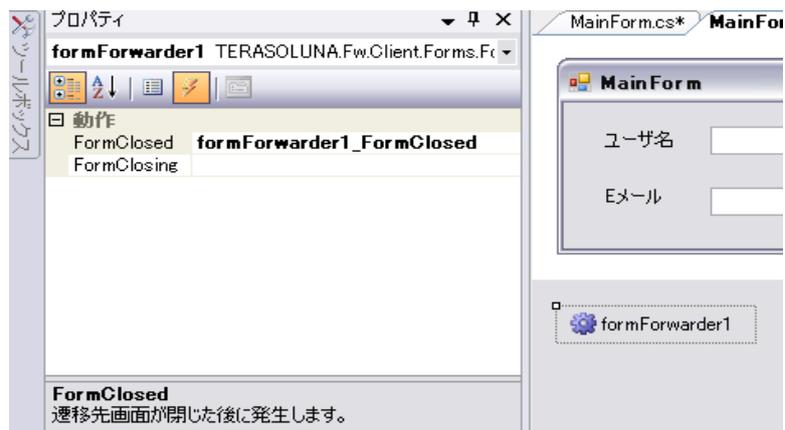


図 4 FormClosed イベントの設定

```
// 遷移元画面で遷移先画面のデータを受け取る実装例
private void formForwarder1_FormClosed(
    object sender, ForwardableFormCloseEventArgs e)
{
    string name = e.Items["NAME"] as string;
    string address = e.Items["ADDRESS"] as string;
    if (!string.IsNullOrEmpty(name))
    {
        this.textBox1.Text = name;
        this.textBox2.Text = mail;
    }
}
```

遷移先の画面が持つコレクション
を参照した処理を記述できる

リスト 6 遷移元画面で遷移先画面のデータを受け取る実装例

■ 内部構成

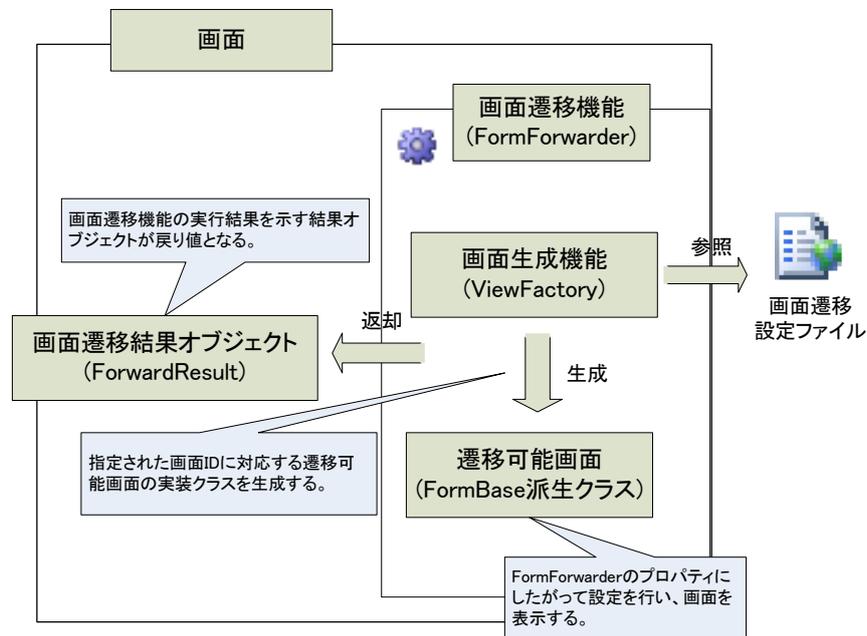


図 5 内部動作イメージ

◆ 遷移元画面から遷移先画面を表示するときの内部動作

画面遷移機能を実行するためには、FormForwarder の Execute メソッドを呼び出す。Execute メソッドは FormForwarder のプロパティに従って画面遷移を行う。以下に画面遷移処理のプロセスを示す。

(1) 遷移先画面の生成

FormForwarder の ViewId プロパティに設定された画面 ID を元に、画面生成機能を用いて遷移先画面を生成する。

(2) 遷移先画面の設定

FormForwarder の ForwardHost プロパティが設定されている場合、遷移元画面から遷移先画面に Items の値を渡す。FormForwarder の ForwardHost プロパティと ConvertId プロパティに有効な値が設定されている場合、遷移元画面と遷移先画面との間でデータセット変換が行われる。遷移先画面に Owner を設定する。

(3) 遷移先画面終了時のイベント設定

FormForwarder の FormClosed イベント、FormClosing イベントが設定されている場合、遷移先画面の対応するイベントから FormForwarder のイベントに登録された処理が実行されるようにイベントハンドラを登録する。

(4) 遷移先画面の初期化

遷移先画面の `Init` メソッドを呼び出し、初期化処理を実行する。

初期化処理が失敗（遷移先画面の `Init` メソッドの戻り値が `false`）の場合、`ForwardResult.Abort` を返却し、処理を終了する。その場合、遷移先画面は表示しない。

(5) 遷移先画面の表示

`FormForwarder` の `Modality` プロパティにしたがい、遷移先画面を表示する。

◆ 遷移先画面を閉じるときの内部動作

(1) データセット反映処理

`FormForwarder` の `ForwardHost` プロパティと `ConvertId` プロパティが設定されている場合、かつ、`DialogResult` プロパティが `OK` の場合、遷移先画面と遷移元画面との間でデータセット変換を実行する。

(2) 遷移元画面の有効化処理

`FormForwarder` の `Modality` プロパティが `Modality.FormModal` である場合、遷移先画面が閉じられる際、遷移元画面を有効化する。

(3) `FormForwarder` のイベント処理

`FormForwarder` の `FormClosing` イベント、`FormClosed` イベントが設定されている場合、登録されたイベントハンドラが遷移先画面の対応するイベントから呼び出される。`FormForwarder` の `Modality` プロパティが `Modality.ApplicationModal` である場合、遷移先画面が閉じられたのち、遷移先画面の `DialogResult` プロパティの値に対応した `ForwardResult` の値が呼び出し元に返却される。

表 5 `ForwardResult` の値

項番	値	説明
1	<code>Abort</code>	通常は"中止"というラベルが指定されたボタンから送られる。画面の初期化に失敗した場合、この値が返される
2	<code>Cancel</code>	通常は"キャンセル"というラベルが指定されたボタン、またはアプリケーションモーダルで表示したフォームの右上の×ボタンを押して閉じた際に送られる
3	<code>Ignore</code>	通常は"無視"というラベルが指定されたボタンから送られる
4	<code>None</code>	画面からの戻り値が存在しない場合に返却される。モードレス、フォームモーダルの画面を表示した際に返される
5	<code>OK</code>	通常は"OK"というラベルが指定されたボタンから送られる
6	<code>Retry</code>	通常は"再試行"というラベルが指定されたボタンから送られる
7	<code>Yes</code>	通常は"はい"というラベルが指定されたボタンから送られる
8	<code>No</code>	通常は"いいえ"というラベルが指定されたボタンから送られる

表 6 `ForwardableFormCloseEventArgs` のプロパティ

項番	プロパティ名	説明
1	DialogResult	遷移先画面の DialogResult プロパティの値が格納される
2	Items	遷移先画面の Items プロパティの内容が格納される
3	InnerEventArgs	遷移先画面でイベント発生時に渡されたイベントオブジェクトが格納される
4	ViewData	遷移先画面の ViewData プロパティの値が格納される

- 遷移先画面終了時の、各処理が実行される順序
 1. デザイン時 Form.FormClosing に登録されたイベントハンドラ
 2. FormForwarder.FormClosing に登録されたイベントハンドラ
 3. Form.Load 以降 Form.FormClosing に登録されたイベントハンドラ
 4. 遷移元画面有効化処理
 5. デザイン時 Form.FormClosed に登録されたイベントハンドラ
 6. データセット反映処理
 7. FormForwarder.FormClosed に登録されたイベントハンドラ
 8. Form.Load 以降 Form.FormClosed に登録されたイベントハンドラ

■ 拡張ポイント

独自の画面遷移機能をフレームワークに沿って利用する場合、FormForwarder を継承するか、あるいは基底クラスの ForwarderBase を継承した画面遷移機能クラスを作成する。

画面として Form を利用する場合には FormForwarder を継承する。Panel 等 Form 以外のコンポーネントを用いた遷移機能を実装する場合、ForwarderBase を継承したクラスを作成する。

■ 関連機能

- ◆ FA-02: 拡張フォーム機能
- ◆ FB-02: データセット変換機能

FA-02 拡張フォーム機能

■ 概要

本機能は、イベント処理機能のエラー処理ハンドラインターフェイス(**IErrorHandler**)と画面遷移機能の遷移可能画面インターフェイス(**IForwardable**)の標準的な実装を備えた基底クラス(**FormBase**)を提供する。

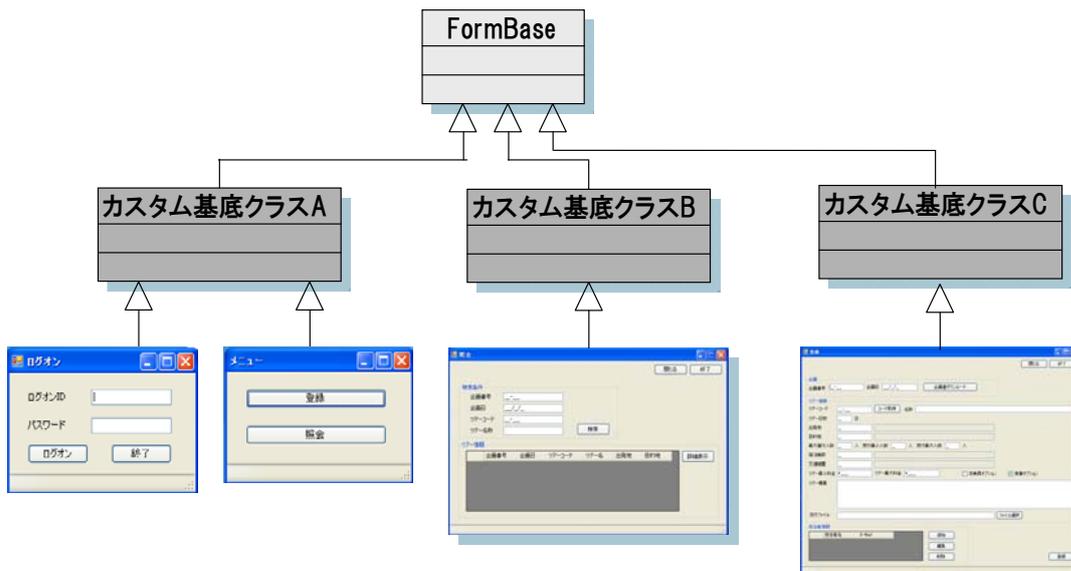


図 1 概要図

基底クラス(**FormBase**)を継承することで定型的な各機能の実装をすることなく、各業務画面の必要なカスタマイズを行うだけで業務画面の開発を行うことができる。

■ 使用方法

◆ 実装方法

本機能を利用する場合、必ず `FormBase` を継承する。

```
public partial class SampleForm : FormBase
{
    public SampleForm()
    {
        InitializeComponent();
    }
}
```



The diagram shows a box containing the text "FormBase を継承する。" (Inherit from FormBase). A line connects this box to the `FormBase` part of the class declaration `SampleForm : FormBase` in the code block above.

リスト 1 実装例

画面遷移機能に必要な設定方法は「FA-01 画面遷移機能」、エラー処理に必要な設定方法は「FB-01 イベント処理機能」を参照のこと。

■ 内部構成

◆ エラー処理の標準実装

TERASOLUNA で提供しているイベント処理機能の `IEnumeratorHandler` を実装している(表 1)。利用時に以下の処理を実行する。

- 入力値検証機能実行前に、画面データセットのカラムに登録しているエラーを解除する。
- 入力値検証エラー時に、エラー対象の画面データセットのカラムにエラーを登録する。

入力値検証エラーが発生した場合、画面にエラーアイコンを表示する方法は、「FA-01 イベント処理機能」を参照のこと。

表 1 拡張フォームの `IEnumeratorHandler` 実装メンバー一覧

項番	メンバ	説明
1	<code>HandleError</code>	画面データセットのカラムにエラーを登録するメソッド。
2	<code>ClearError</code>	画面データセットのカラムのエラーを解除するメソッド。

◆ 画面遷移の標準実装

TERASOLUNA で提供している画面遷移機能の `IForwardable` を実装している(表 2)。画面遷移機能で用いる画面間のデータ受け渡しに関する各種のプロパティを提供する。画面遷移機能の詳細は「FA-01 画面遷移機能」を参照のこと。

表 2 拡張フォームの `IForwardable` 実装メンバー一覧

項番	メンバ	説明
1	Items	画面アイテムを保持している <code>Dictionary</code> の取得を行うプロパティ。
2	ViewData	画面データセットの取得及び設定を行うプロパティ。
3	Init	初期化メソッド。本機能では処理は何もせず、必ず <code>true</code> を返却する。

◆ 構成クラス

表 3 構成クラス一覧

項番	クラス名	説明
1	FormBase	<code>ExceptionHandler</code> , <code>IForwardable</code> を実装した基底クラス。

■ 拡張ポイント

◆ 入力値検証エラー時の処理を変更する

入力値検証エラー時の処理を変更する場合、拡張フォーム (`FormBase`) の `HandleValidationError` メソッド(表 4)をオーバーライドする。

表 4 `HandleValidationError` の引数一覧

項番	メンバ	型	説明
1	Messages	<code>IList<MessageInfo></code>	入力値検証エラーの内容を保持した <code>MessageInfo</code> のリスト。
2	dataSet	<code>DataSet</code>	画面データセット。

■ 関連機能

- FA-01 画面遷移機能
- FB-01 イベント処理機能

FB-01 イベント処理機能

目次

■ 概要	2
■ 使用方法(同期実行).....	6
◆ 概要	6
◆ 実装方法.....	6
➢ イベントコントローラの実行	6
➢ 画面項目とデータセットのバインド(関連付け).....	14
➢ ビジネスロジック実行機能	21
➢ 入力値検証実行機能	30
➢ データセット変換機能	33
➢ イベント実行機能	38
➢ エラー処理機能	54
➢ その他の機能	57
■ 使用方法(非同期実行)	60
◆ 概要	60
◆ 実装方法.....	61
➢ イベントコントローラの実行	61
➢ イベント実行機能	65
■ 内部構成	70
◆ 詳細フロー.....	70
◆ 構成クラス.....	72
■ 拡張ポイント	74
■ 関連機能	76

■ 概要

イベント処理機能は、Windows アプリケーションにおける業務処理呼び出しの処理フローを定型化し、Visual Studio のデザイナーを用いてビジネスロジックや入力値検証、各種イベント処理等を簡単に実行できる仕組みを提供する。また、開発コストの高い非同期処理を容易に行えるようにする。

イベント処理機能はビジネスロジック実行機能、入力値検証機能、データセット変換実行機能、各種イベント実行機能、エラー処理機能の5つのサブ機能で構成される。

(1) ビジネスロジック実行機能

業務処理および通信処理を行うビジネスロジックを実行する機能。ビジネスロジックの同期・非同期実行を行う。非同期処理の場合には、実行中のキャンセル処理呼び出しをサポートする。

(2) 入力値検証実行機能

入力値検証設定ファイルを用いて、画面に入力された項目を検証する機能。入力値検証機能の詳細に関しては、『CM-03 入力値検証機能』を参照のこと。

(3) データセット変換実行機能

画面から入力されたデータをビジネスロジックに渡し、ビジネスロジックから返却されたデータを画面へ反映する機能。データセット変換機能の詳細に関しては、『FB-02 データセット変換機能』を参照のこと。

(4) イベント実行機能

フレームワークが定めたタイミングで、業務開発者がイベントハンドラを追加・実装できるようにする機能。たとえば、ハードコーディングによる関連チェックを行う場合は、「カスタムチェックイベント」にイベントハンドラを追加する。

イベント実行機能では、以下の7種類のイベントを提供している。

- カスタムチェックイベント
- 入力値検証失敗イベント
- 前処理イベント
- 前処理失敗イベント
- ビジネスロジック失敗イベント
- 非同期処理完了イベント
- 進行状況通知イベント

(5) エラー処理機能

イベント処理全体で統一的なエラー処理とエラークリア処理を行う機能。エラー処理は、ビジネスロジックやイベント実行中にエラーが発生した場合に、イベント処理共通のエラー処理として呼ばれる。エラー処理では、主に入力値検証エラーなどのエラー情報を画面へ表示する処理を行う。また、エラークリア処理は、イベント処理実行の都度、画面へ表示されているエラー情報をクリアするために、イベント処理実行直後に呼ばれる。

イベント処理機能とフレームワークで提供するその他機能の関係、および画面からイベントコントローラ(イベント処理コンポーネント)を通してビジネスロジック(業務処理)を実行する際の動作概念図(図 1)、処理フロー(図 2)、イベント発生タイミング(図 3)を示す。



図 1 イベント処理呼び出しの動作概念図

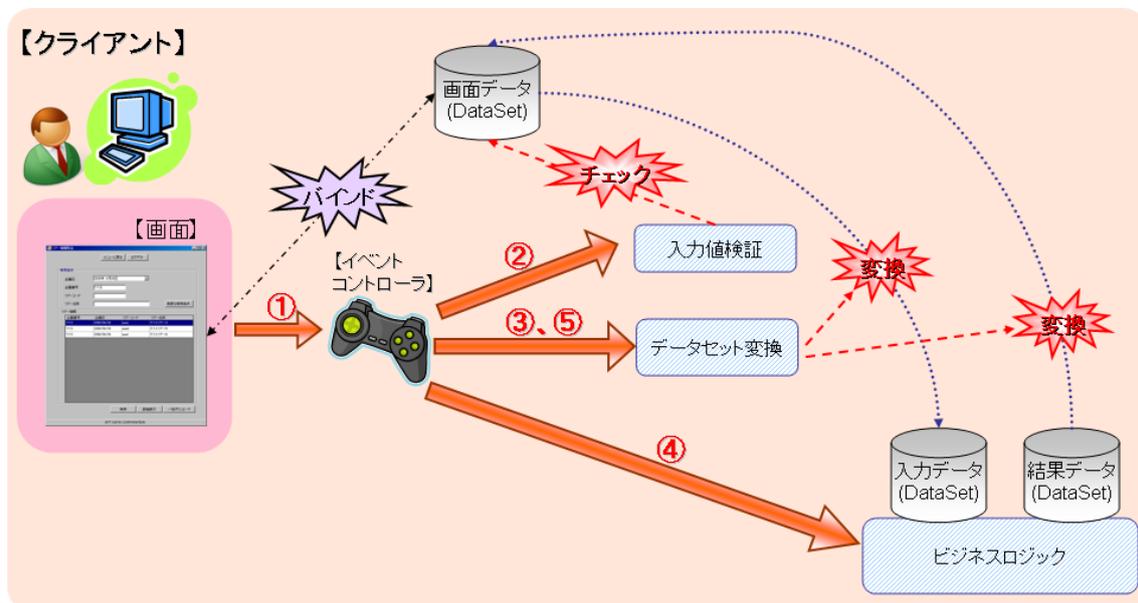


図 2 処理フロー

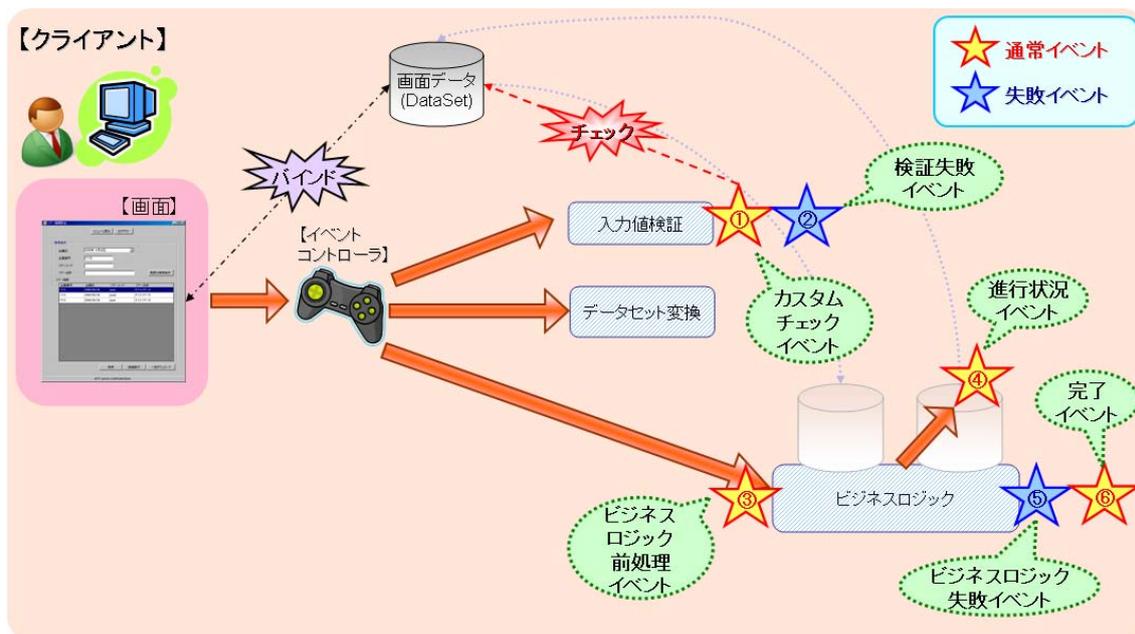


図 3 イベント発生タイミング

イベントコントローラを用いることで、入力値検証、データセット変換、ビジネスロジック実行までの一連の処理および、各種イベントに追加したイベントハンドラを実行することが可能となる。

実行形態としては、同期実行と非同期実行が選択可能である。同期実行と非同期実行で処理フローに変化はないが、コーディング方法が若干変わるので、本資料では両者のパターンを分けて説明を行う。

業務呼び出しのフローは、図 2 の通り、①画面からのデータ取得、②入力値検証、③ビジネスロジックの入力情報の生成、④ビジネスロジック実行、⑤結果を画面に反映、となる。

イベントの発生タイミングは、図 3 の通り入力値検証の後に①カスタムチェックイベントおよび②検証失敗イベントが発生し、ビジネスロジックの実行前に③前処理イベント、ビジネスロジック実行後に④ビジネスロジック失敗イベント、全イベント終了後に⑤完了イベントが発生する。この業務処理の呼び出しフローをまとめてイベント処理と呼ぶ。

それぞれのプロセスに必要な情報は、イベントコントローラを利用することで、Visual Studio のデザイナーから設定することができる。業務開発者は業務処理を行うビジネスロジックを作成し、イベントコントローラを利用してビジネスロジックを実行する。

■ 使用方法(同期実行)

◆ 概要

イベントコントローラはイベント処理を呼び出すための設定をデザイナーから行うためのコンポーネントである。イベントコントローラを実行することで、入力値検証、データセット変換、ビジネスロジックが順番に実行され、定められたタイミングで各種イベントが発生する。

同期実行を行う場合、イベントコントローラの `Execute` メソッドを実行する。`Execute` メソッドを利用した同期処理の結果の処理は、`Execute` メソッドの戻り値となる `ExecutionResult` インスタンスを確認することで行う。なお、同期実行の場合、完了イベントは発生しない。

イベント処理フローの詳細に関しては、「図 32 同期処理のイベント処理フロー」を参照のこと。

◆ 実装方法

▶ イベントコントローラの実行

業務開発者は画面を実装する際、イベントコントローラを利用して業務処理を呼び出す。そのためには、まず①イベントコントローラをツールボックスから画面へ貼り付ける。次に②イベントコントローラのプロパティ・イベントを設定する。最後に③ボタンクリックなどの画面イベントよりイベントコントローラの `Execute` メソッドを呼び出し、`Execute` メソッドの戻り値となる `ExecutionResult` インスタンスを用いて結果の確認を行う。

以下に、イベントコントローラを利用して業務処理の呼び出しを行う作業手順を示す。

① 画面へのイベントコントローラの追加

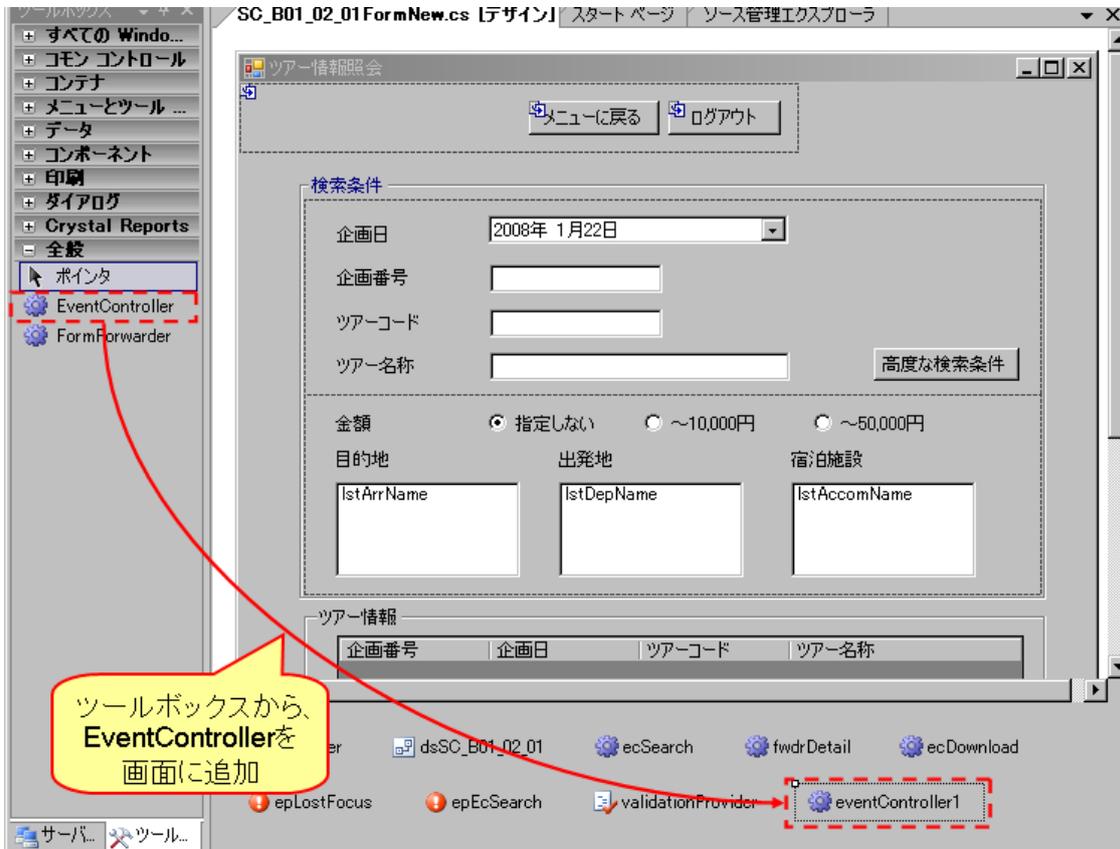


図 4 イベントコントローラの追加と設定

② イベントコントローラのプロパティ設定

イベントコントローラに設定するプロパティについては、「表 1 イベントコントローラのプロパティ」を参照のこと。

The screenshot shows the 'Properties' window for 'eventController1' with the following table of settings:

プロパティ	値
ILogicName	
ILogicParamTypeName	
ILogicResultTypeName	
ConvertId	
ErrorHandler	(なし)
RequestName	
RuleSet	
ValidationFilePath	
ViewData	(なし)

The 'デザイン' (Design) section shows:

プロパティ	値
(Name)	eventController1
GenerateMember	True
Modifiers	Private

The yellow callout box contains the following text:

要件に応じて、以下のプロパティ設定を行う。

- BlogicName: ビジネスロジック名
- BlogicParamTypeName: 入力データセット型名
- BlogicResultTypeName: 出力データセット型名
- ConvertId: コンバートID
- ErrorHandler: エラー処理クラス
- RequestName: リクエスト名
- RuleSet: 入力値検証ルールセット
- ValidationFilePath: 入力値検証設定ファイル名
- ViewData: 入力情報となる画面データセット

図 5 イベントコントローラのプロパティ設定

③ イベントコントローラの Execute メソッドの実行

ボタンクリックのイベントハンドラなどから、イベントコントローラのExecuteメソッドを実行する。Executeメソッドの戻り値のExecutionResultには、イベント処理の実行結果が格納される。成功したかどうかは、ExecutionResultのSuccessプロパティによって確認することができる。ExecutionResultの詳細に関しては、「戻り値の確認」を参照のこと。

金額 指定しない ~10,000円 ~50,000円

目的地 出発地 宿泊施設

目的地的地 出発地 宿泊施設

ツアー情報

企画番号	企画日	ツアーコード	ツアー名称
------	-----	--------	-------

検索 詳細表示

イベントコントローラのExecuteメソッドを実行する。

```
/// <summary>
/// 検索ボタンのクリックイベント。
/// </summary>
private void search_Click(object sender, EventArgs e)
{
    // イベントコントローラを用いてイベント処理を実行する。
    ExecutionResult result = eventController1.Execute();

    if (result.Success)
    {
        // 成功時の処理
        MessageBox.Show("イベント処理が成功しました");
    }
    else
    {
        // 失敗時の処理
        MessageBox.Show("イベント処理が失敗しました");
    }
}
```

Clickイベントなどから、イベントコントローラを実行する。

図 6 イベント処理の呼び出し(同期)

- イベントコントローラのプロパティ一覧
イベントコントローラで定義されているプロパティ一覧を示す。

表 1 イベントコントローラのプロパティ

項番	項目名	例(形式)	説明
1	BLogicName	Communicate (ビジネスロジック名)	ビジネスロジック名を設定する。イベントコントローラはビジネスロジック名に対応するビジネスロジックを実行する。ビジネスロジック名とビジネスロジッククラスの対応付けは、ビジネスロジック設定ファイルで行う。 ビジネスロジック名を設定しなかった場合、何も処理を行わないビジネスロジック(NopBLogic)が自動的に実行される。
2	BLogicParamTypeName	ClientPrototypeApp.ViewTourInfo, ClientPrototypeApp (ビジネスロジック入力データセットの完全修飾名)	ビジネスロジック入力情報として用いるデータセットの完全修飾名を設定する。未設定の場合、ViewData プロパティに設定された画面データセットの型を利用する。
3	BLogicResultTypeName	ClientPrototypeApp.DtoTourInfo, ClientPrototypeApp (ビジネスロジック出力データセットの完全修飾名)	ビジネスロジック出力情報として用いるデータセットの完全修飾名を設定する。未設定の場合、ViewData プロパティに設定された画面データセットの型が利用される。
4	ConvertId	GetTourCode (コンバート ID)	データセット変換で用いるコンバート ID を設定する。データセット変換機能は、コンバート ID をキーとしてデータセット変換設定ファイルから、ViewData プロパティに設定された画面データセットとビジネスロジック入出力データセットの変換情報を取得し、データセットの変換・反映処理を実行する。未設定の場合は、データセット変換機能は実行されない(『FB-02 データセット変換機能』参照)
5	ErrorHandler	RegisterForm (IErrorHandler を実装したクラスのインスタンス)	IErrorHandler を実装したクラスのインスタンスを設定する。イベント処理実行においてエラーが発生した場合に、処理が委譲される。

項番	項目名	例(形式)	説明
6	RequestName	GetTourCode	サーバ上で実行対象となるリクエスト処理の識別子を設定する。ビジネスロジックで通信処理が必要な場合、必須項目である。
7	RuleSet	Default	入力値検証で用いるルールセットを指定する。未設定の場合、“Default”が利用される。(『CM-02 入力値検証機能』参照)
8	ValidationFilePath	ClientPrototypeApp¥Validation¥GetTourCode.conf ig (入力値検証設定ファイルのパス文字列)	入力値検証設定ファイルのパスを設定する。入力値検証機能は、指定された入力値検証設定ファイルを用いて ViewData プロパティに設定された画面データセットに対し入力値検証を実施する。未設定の場合、入力値検証は実行されない。(『CM-02 入力値検証機能』参照)
9	ViewData	tourInfoDs1 (画面データセットのインスタンス)	画面データセットのインスタンスを設定する。未設定の場合、形なし DataSet のインスタンスが生成され、利用される。
10	ErrorPaths		デザイナーには表示されない。 エラー発生箇所を表すキーのリストを取得する。入力値検証エラーが発生した場合、データセットのエラーが発生した位置を表す XPath 文字列のリストとなる。
11	Items		デザイナーには表示されない。 外部からアクセス可能なオブジェクトを取得または設定する。デザイナーから設定可能なプロパティや入力となる画面データセット以外の情報をイベント処理に渡したい場合に利用する。
12	IsBusy		デザイナーには表示されない。 ExecuteAsync によって非同期実行が行われているかどうかを表す。

- ExecutionResultのプロパティ一覧
ExecutionResult のプロパティ一覧を示す。

表 2 ExecutionResult のプロパティ

項番	プロパティ名	説明
1	ResultString	実行結果を判定するための文字列。正常終了時には"success"が格納される。"sccess"格納されたかどうかは、項番 4 の Success プロパティを使うことで簡単に判別できる。
2	BLogicResultData	ビジネスロジック出力データセット。 ビジネスロジックの実行結果として、ExecutionResult の ResultData プロパティに設定されたデータセットが格納される。
3	Errors	エラーのメッセージ情報リスト。
4	Success	イベント処理が成功したかどうかを表すフラグ。ResultString プロパティの値が "success"(ExecutionResult.SUCCESS で定義された文字列)であれば true、そうでなければ false となる。

- 戻り値の確認

イベント処理が正常に実行されたかどうかを判別する場合は、**ExecutionResult** の **Success** プロパティを確認する。イベント処理が正常に完了した場合は **Success** プロパティに **true** が設定され、入力値検証エラーが発生した場合やビジネスロジックが失敗した場合などには、**Success** プロパティに **false** が設定される。

イベント処理の実行に失敗した場合のエラー種別を判別する場合は、**ExecutionResult** の **ResultString** プロパティを確認する。**ResultString** プロパティには、エラー内容に応じた文字列(イベント処理を実行した結果)が格納されているので、エラーハンドリング処理が可能となる。

ExecutionResult のプロパティの一覧を以下に記す。

表 3 **ExecutionResult** の **ResultString** 値一覧

項番	状態	値
1	単項目チェックエラー	“validationError”
2	カスタムチェックエラー	“validationError”
3	前処理エラー	“preprocessedError”または前処理イベントで設定する ResultString の値
4	ビジネスロジック実行失敗	ビジネスロジック実行結果オブジェクトの ResultString の値
5	ビジネスロジック実行成功	“success”

なお、ビジネスロジック成功/失敗は正常に処理を終えることができたことを示すものであり、業務的な意味を持たないことに注意すること。ビジネスロジックが失敗する場合には、サーバでの例外や通信途絶、サーバでの入力値検証エラーなどのパターンがある。

▶ 画面項目とデータセットのバインド(関連付け)

TERASOLUNA Client Framework for .NET では、1つの画面に対して1つのデータセットを用意して、画面項目とデータセットをバインドする開発方法を想定している。ここでは、画面項目と画面に紐づくデータセットをバインドする一般的な手順を説明する。※本資料では、画面に紐づくデータセットのことを「画面データセット」と省略して記述する。

① 画面の作成

画面フォームを作成し、作成したフォームにツールボックスからテキストボックスやボタンなどの画面項目を配置する。

② 画面データセットの作成

TERASOLUNA を使用する場合、1つの画面に対して1つのデータセットを用意する方法が推奨される。この方法を採用した場合、画面に紐づくデータセットの型がそのままビジネスロジックの入出力データセットの型となる。この場合、1つのデータセットの中に、ビジネスロジックに渡す値を格納するためのデータテーブルと、ビジネスロジックから返却される値を格納するためのデータテーブルが存在するイメージとなる(図 7)。1つの画面で複数のビジネスロジックを実行する場合は、データセットを増やすのではなく、データセットの中のデータテーブルの数を増やしていくようなイメージとなる。

データセット作成後、作成したデータセットをツールボックスから画面にドラッグアンドドロップしてデータセットを追加すること(図 8)。

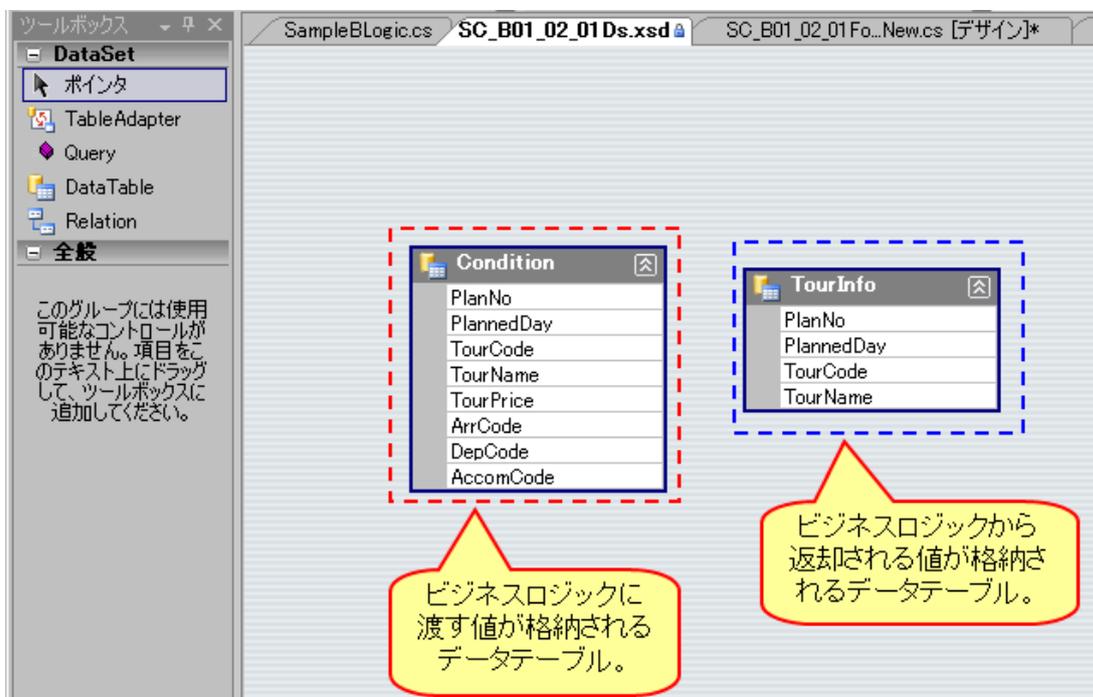


図 7 画面データセットの作成



図 8 画面データセットの追加

③ 画面項目とデータセットのバインド

Visual Studio のデザイナを使って、画面項目を対応するデータセットの列にバインドさせる。コントロールによってバインド方法が違う点に注意すること。ここでは、テキストボックスをバインドさせるサンプルと、DataGridView をバインドさせるサンプルを以下に記す。

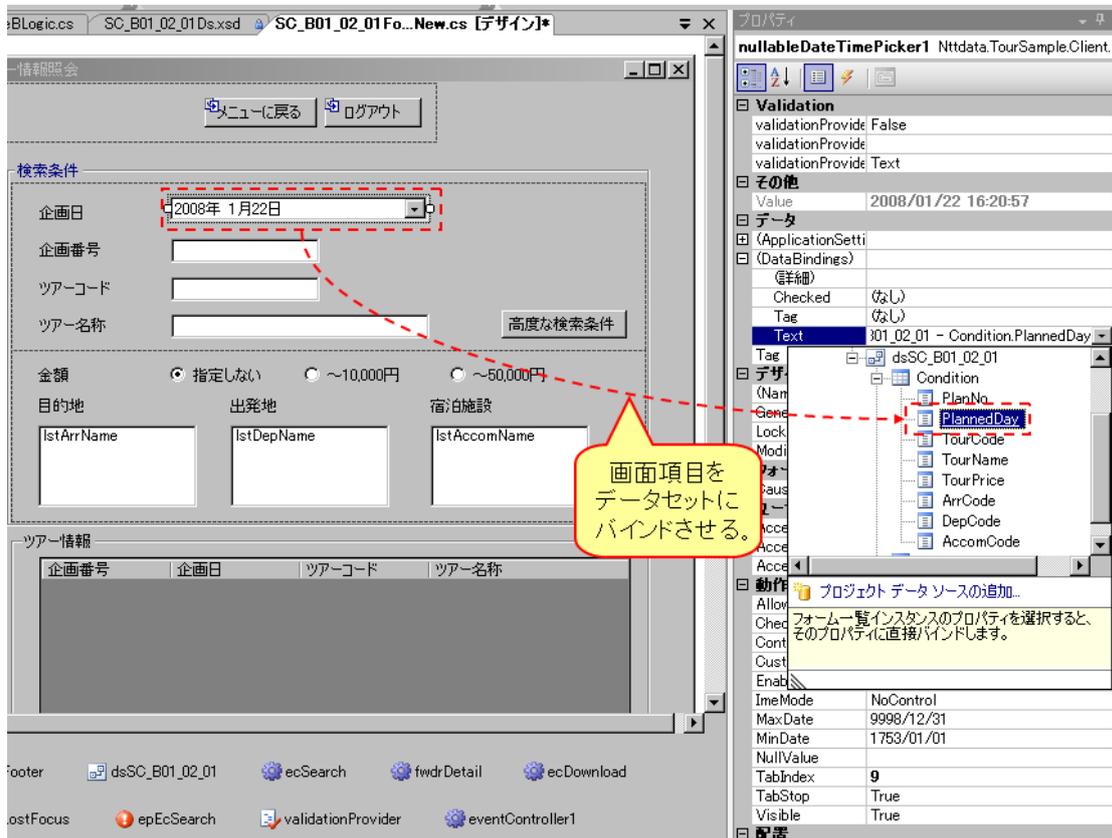


図 9 テキストボックスにおける画面項目とデータセットのバインド

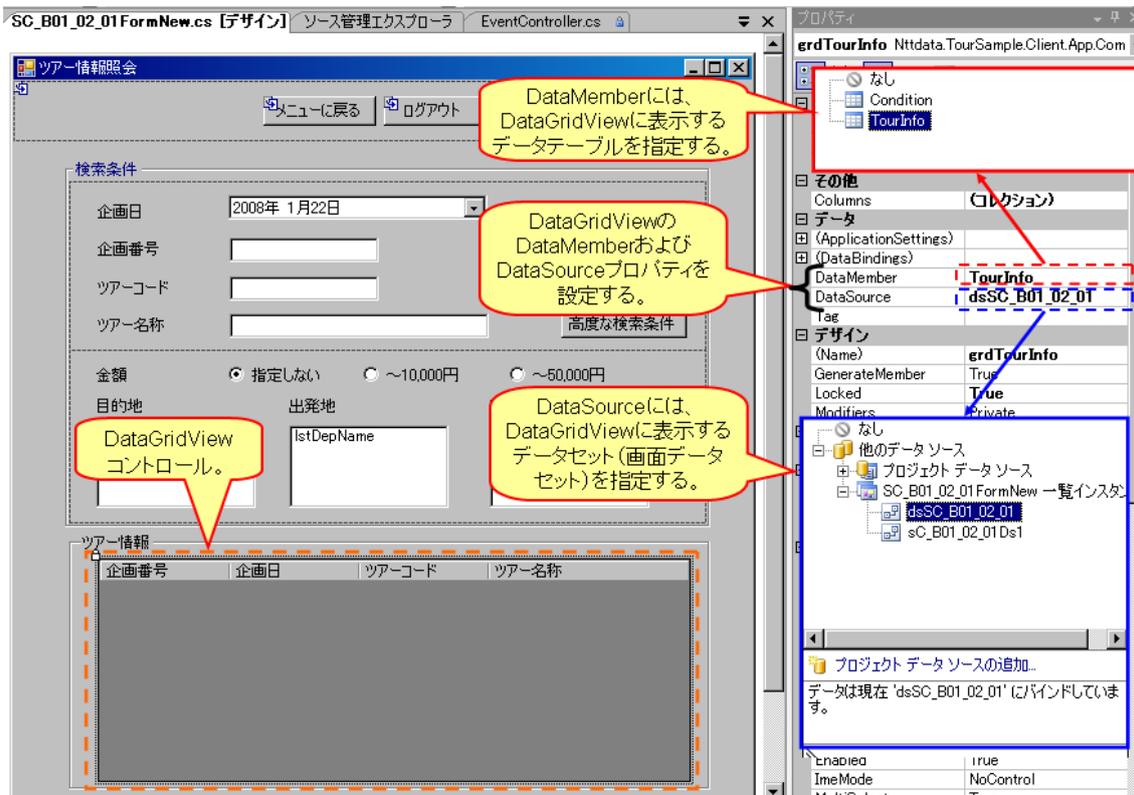


図 10 DataGridView における画面項目とデータセットのバインド(手順①)

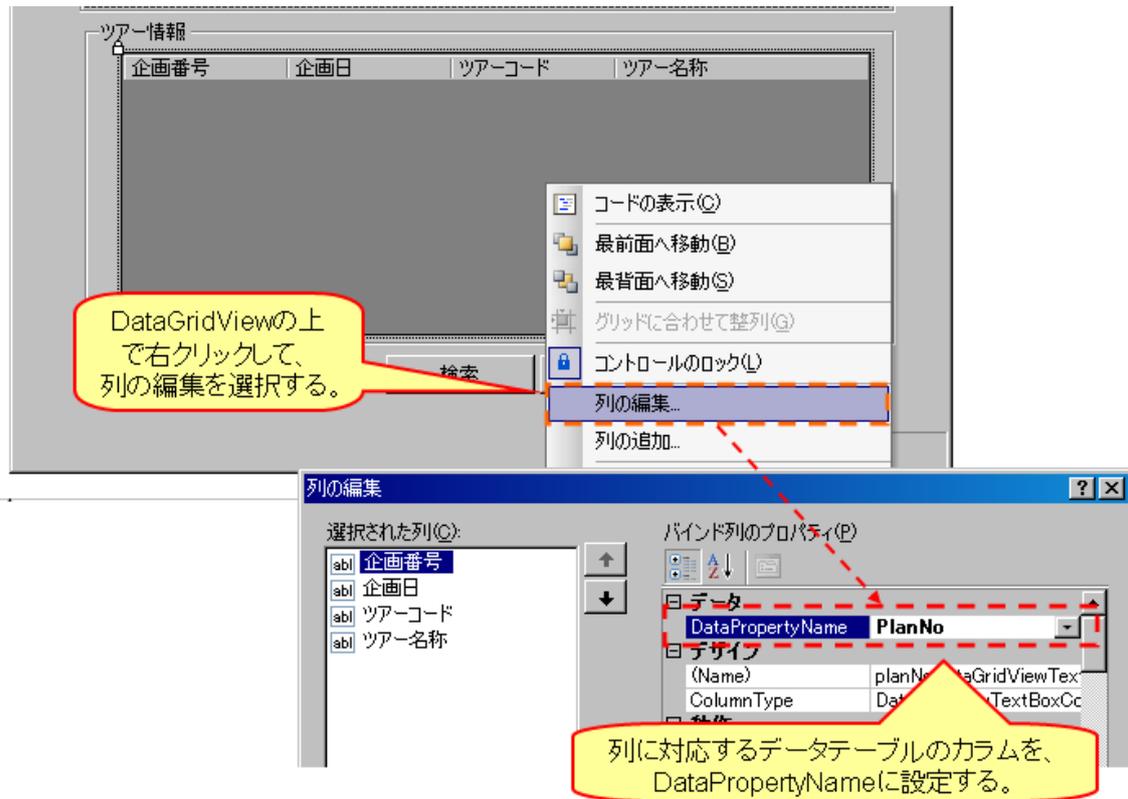


図 11 DataGridView における画面項目とデータセットのバインド(手順②)

④ イベントコントローラのプロパティにデータセットを設定

イベントコントローラの **ViewData** プロパティに画面データセットのインスタンスを設定することで、イベントコントローラとビジネスロジックの間でデータセットの内容の受け渡しが可能となる。画面項目以外の値をビジネスロジックへ渡す場合は、バインドのメカニズムは使わず、ハードコーディングで画面データセットに任意の値を格納すること。

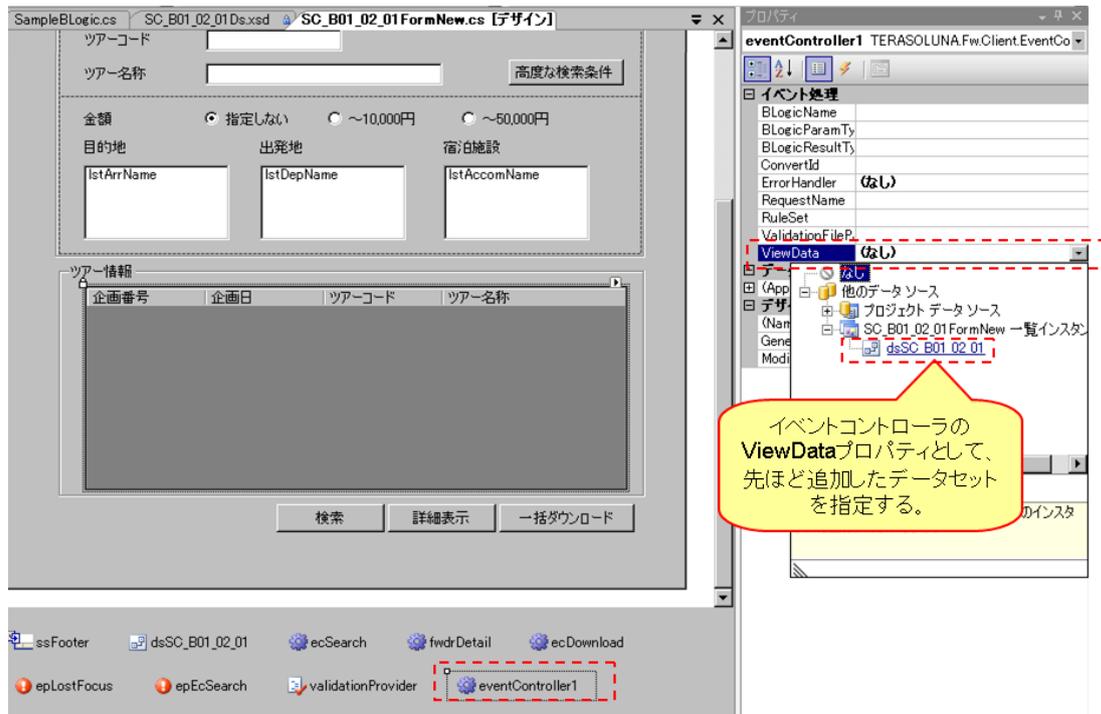


図 12 イベントコントローラのプロパティ設定(ViewData)

◇ ノウハウ

ビジネスロジックの入力データセットおよび出力データセットに、個別のデータセットの型を指定する方法

ビジネスロジック入力データセットの型名をイベントコントローラの **BLogicParamTypeName** プロパティに、ビジネスロジック出力データセットの型名を **BLogicResultTypeName** プロパティに設定することで、ビジネスロジックの入力データセットの型と出力データセットの型を個別に指定できる。

上記プロパティを省略した場合は、**ViewData**(画面データセット)の型が自動的にビジネスロジックの入出力データセットの型として使用される。

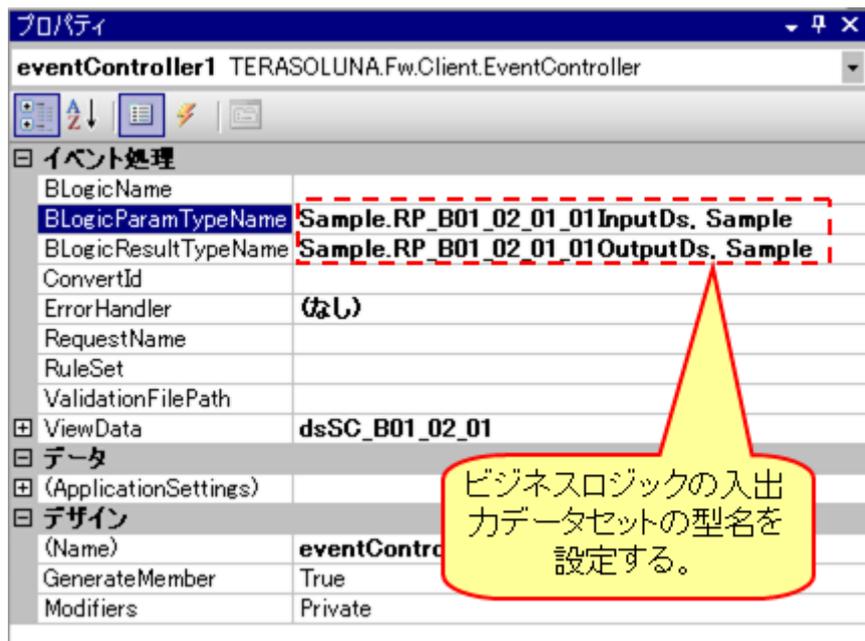


図 13 イベントコントローラのプロパティ設定(入出力データセット)

▶ ビジネスロジック実行機能

◇ 概要

ビジネスロジックの種類は大別すると、独自の処理を行うビジネスロジックと、TERASOLUNA Client Framework for .NET が用意するビジネスロジックの2種類に分けられる。

TERASOLUNA Client Framework for .NET では、サーバとの通信処理を行うビジネスロジックも提供している。通信処理用ビジネスロジックの詳細に関しては以下のドキュメントを参照のこと。

- ✓ XML 通信:『FC-01 XML 通信機能』
- ✓ ファイルアップロード:『FC-02 ファイルアップロード機能』
- ✓ ファイルダウンロード:『FC-03 ファイルダウンロード機能』

ここでは独自のビジネスロジックを実行する場合と、TERASOLUNA Client Framework for .NET が用意するビジネスロジックの中で、何も処理を行わないビジネスロジックを実行するケースについて説明する。

◇ 独自のビジネスロジックを実行する

● 業務開発者のコーディングポイント

独自のビジネスロジッククラスを作成し、イベントコントローラのプロパティに作成したビジネスロジックのビジネスロジック名を設定することで、実行できる。以下に、独自のビジネスロジックの作成方法を順番に説明する。

① ビジネスロジッククラスの作成

IBLogicインターフェイスを実装したクラスを作成し、Executeメソッド内にロジックを実装する。BLogicParam、BLogicResultはそれぞれビジネスロジックの入力情報、出力情報を表すクラスである。詳細は、「表 4 BLogicParamのプロパティ」「表 5 BLogicResultのプロパティ」を参照のこと。

エラー処理の詳細に関しては、「ビジネスロジックにおけるエラー処理方針」を参照のこと。

```
public BLogicResult Execute (BLogicParam param)
{
    // 入力データセットのキャスト
    SampleDataSet inputDataSet = (SampleDataSet)param.ParamData;
    // 入力データ(画面から設定された値)の取得
    string name = inputDataSet.User[0].Name;
    if (string.IsNullOrEmpty(name))
    {
        // 非業務エラー(システムエラー)発生時は例外をスローする
        throw new ArgumentException("名前が設定されていません。");
    }

    if (!"豊洲さん".Equals(name))
    {
        // 業務エラー発生
        // BLogicResultに"success"以外の文字列(例:"failure")を設定し、
        // エラー内容をErrorsに格納する
        BLogicResult result = new BLogicResult("failure");
        MessageInfo info = new MessageInfo("名前が間違っています。");
        result.Errors.Add(info);
        return result;
    }

    // 結果データの設定
    SampleDataSet outputDataSet = new SampleDataSet();
    outputDataSet.ResultTable.AddResultTableRow("resultValue01");
    // 結果クラスの生成および返却
    // 実行に成功した場合は、BLogicResultにSUCCESSを設定する
    return new BLogicResult(BLogicResult.SUCCESS, outputDataSet);
}
```

リスト 1 ビジネスロジックの実装例

② アプリケーション構成ファイルの記述

アプリケーション構成ファイルにビジネスロジック設定ファイルのパスを指定する。ビジネスロジック設定ファイルは複数指定できる。ビジネスロジック設定ファイルの詳細に関しては『CM-04 ビジネスロジック生成機能』を参照のこと。

以下に、ビジネスロジック設定ファイルのパスを指定する例を示す。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
type="TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection,
TERASOLUNA.Fw.Common"/>
  </configSections>
  <blogicConfiguration>
    <files>
      <file path="config¥BLogicConfiguration1.config"/>
      <file path="config¥BLogicConfiguration2.config"/>
    </files>
  </blogicConfiguration>
</configuration>
```

②で作成したビジネスロジック設定ファイルのパスを記述する。複数ファイルの指定が可能。

リスト 2 アプリケーション構成ファイルへのビジネスロジック設定ファイルのパス記述例

③ ビジネスロジック設定ファイルの記述

ビジネスロジック設定ファイルにビジネスロジック名とビジネスロジッククラスの完全修飾名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。ビジネスロジック設定ファイルの詳細については『CM-04 ビジネスロジック実行機能』を参照のこと。

以下に、ビジネスロジック設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8"?> ビジネスロジック名  
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">  
  <blogic name="sample" type="Sample.BLogic.SampleBLogic, Sample" />  
</blogicConfiguration>
```

ビジネスロジッククラスのアセンブリ修飾名

リスト 3 ビジネスロジック設定ファイルの記述例

- ビジネスロジックにおけるエラー処理方針

TERASOLUNA Client Framework for .NET では、ビジネスロジックで業務的なエラーが発生した場合と、非業務的なエラーが発生した場合のそれぞれについて処理方針を定めている。

業務的なエラーが発生した場合は、ビジネスロジックで **BLogicResultString** に "success" 以外の文字列を、**Errors** にエラー内容を追加し、呼び出し元のフォームのイベント(クリックイベントなど)で **ExecutionResult** を参照しエラー処理を行うことを想定している。ビジネスロジックでエラーを格納する実装例については、「リスト 1 ビジネスロジックの実装例」を参照すること。

非業務的(業務が意識すべきでない)なエラーが発生した場合は、**BLogicResult** を返却するのではなく例外をスローする。ただし、ビジネスロジックの呼び出し元のフォームなどで例外のキャッチなどを行うのではなく、フォームを実行する **Main** クラスでエラー処理を行うことを想定している。ビジネスロジックで例外をスローする実装例については「リスト 1 ビジネスロジックの実装例」を参照のこと。**Main** クラスでエラー処理を行う場合の実装例については、「リスト 4 非業務エラーを **Main** クラスで処理する場合の記述例」を参照のこと。

なお、TERASOLUNA Client Framework for .NET の通信用ビジネスロジックもこの方針に基づきエラー処理が実装されている。たとえば、サーバで業務エラーが発生した場合には **BLogicResultString** にサーバから返却されたエラー文字列(例: `serviceException`) が格納され、**Errors** にサーバから返却されたエラー内容が格納される。詳細は、『FC-01 XML 通信機能』を参照のこと。

```
static class Program
{
    private static ILog log = LogFactory.GetLogger(this.GetType());

    /// <summary>
    /// アプリケーションのメイン エントリ ポイントです。
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        // ThreadExceptionイベントを設定する
        Application.ThreadException += new
        ThreadExceptionHandler(Application_ThreadException);
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        // フォームを起動する
        ApplicationManager.Instance.Run("SC_B99_01_02", args);
    }

    /// <summary>
    /// フォームで例外がスローされた場合に呼び出されるイベント。
    /// </summary>
    public static void Application_ThreadException(object sender,
    ThreadExceptionEventArgs e)
    {
        // 例外のスタックトレースなどはログに出力する。
        log.Error("致命的なエラーが発生しました。", e.Exception);

        // 利用者にはシステムエラーが発生したことを伝える。
        MessageBox.Show("システムエラーが発生しました。");
    }
}
```

Application クラスに、
ThreadException イベントをを
追加する。

ビジネスロジック(フォーム)で、
例外が処理(キャッチ)されなかった場合、
このメソッドが実行される。

リスト 4 非業務エラーを Main クラスで処理する場合の記述例

- イベントコントローラのプロパティ設定
イベントコントローラの BLogicName プロパティに②で記述したビジネスロジック名を設定する。

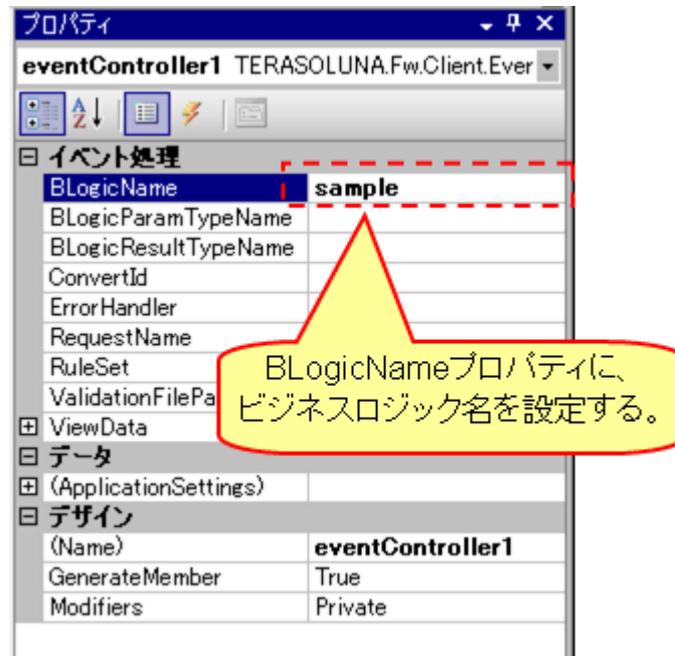


図 14 イベントコントローラのプロパティ設定(BLogicName)

- BLogicParam のプロパティ

BLogicParam のプロパティ一覧を示す。

表 4 BLogicParam のプロパティ

項番	プロパティ名	説明
1	ParamData	ビジネスロジックの入力データセット。 データセット変換機能により、画面に紐付けられたデータセットから抽出された値が格納されてくる。
2	Items	ビジネスロジック実行に必要なパラメータを格納する IDictionary。

- BLogicResult のプロパティ

BLogicResult のプロパティ一覧を示す。

表 5 BLogicResult のプロパティ

項番	プロパティ名	説明
1	ResultString	ビジネスロジックの実行結果を表す文字列が格納される。"success"が設定されていればビジネスロジック成功とし、それ以外の文字列が設定されていればビジネスロジックが失敗したと見なされ、ビジネスロジック失敗イベントおよびエラー処理が実行される。
2	ResultData	ビジネスロジックの実行結果を格納するデータセット。 データセット変換機能により、このデータセットに設定した値が画面に紐づくデータセットに反映される
3	Errors	ビジネスロジック内部で発生したエラー情報を格納したリスト。
4	Items	ビジネスロジックの実行結果情報を格納した IDictionary。

◇ 何も処理を行わないビジネスロジックを実行する

イベント処理は実行したいが、業務ロジックそのものでは何も処理を行う必要がない場合、イベントコントローラのビジネスロジックプロパティを空白のままにしておけば、自動的に何も処理を行わないビジネスロジック(NopBLogic)が実行される。

本ビジネスロジックは、画面の入力値検証のみを行う場合などに使用する。

◇ ノウハウ

TERASOLUNA Client Framework for .NETにおけるビジネスロジックの扱い

TERASOLUNA を使用するからといって、必ずしも独自のビジネスロジックを作成する必要はない。従来どおり画面のイベントハンドラ内にロジックを書いても構わない。どのような状況で独自のビジネスロジックを作成するかは、各プロジェクトの方針に従うこと。(例:SQL を発行するロジックに関しては独自のビジネスロジックに記述する)

▶ 入力値検証実行機能

◇ 概要

入力値検証実行機能は、画面に入力されたデータに対して、入力チェックを行う機能である。入力チェックは入力値検証設定ファイルの内容に基づき行われる。

イベントコントローラの `ValidationFilePath` プロパティに入力値検証設定ファイルのパスを設定することにより、入力チェックが有効化される。入力値検証機能の詳細に関しては、『CM-03 入力値検証機能』を参照のこと。

エラー処理機能と組み合わせることで、画面に入力値検証エラーの内容を表示することができる。エラー処理機能の詳細に関しては、『FA-02 エラー処理機能』を参照のこと。

◇ 業務開発者のコーディングポイント

画面項目に対する入力チェックを行うためには、入力値検証設定ファイルの作成とイベントコントローラの `ValidationFilePath` プロパティに入力値検証設定ファイルのパスを指定する。画面項目に対してチェックが行われるのではなく、画面項目がバインドされたデータセットに対してチェックが行われることに注意すること。

① 入力値検証設定ファイルの記述

入力チェック対象には、画面データセットの **DataRow** と **DataColumn** を指定する。本ファイルは手動ではなく、EnterPrise Library が提供するツール「EnterPrise Library Configuration」を用いて、作成すること。

入力値検証設定ファイルは、「出力ディレクトリにコピー」プロパティを「常にコピーする」にすること。入力値検証設定ファイルの生成単位は、プロジェクトの方針に従うこと。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="validation" type="Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0" />
    <section name="dataConfiguration" type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings, Microsoft.Practices.EnterpriseLibrary.Data, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0" />
  </configSections>
  <validation>
    <type assemblyName="TourSample.Client.App.Ucb0102.Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
      name="TourSample.Client.App.Ucb0102.SC_B01_02_01Ds+ConditionRow" />
    <ruleset name="Default" />
    <properties>
      <property name="PlanNo">
        <validator messageTemplate=""
          messageTemplateResourceName="" messageTemplateResourceType="" tag=""
          type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.OrCompositeValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0" />
        <validator negated="true" messageTemplate="" messageTemplateResourceName=""
          messageTemplateResourceType="" tag=""
          type="TERASOLUNA.FW.Common.Validation.Validators.RequiredValidator, TERASOLUNA.FW.Common.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0"
          name="Required Validator" />
        <validator messageTemplate=""
          messageTemplateResourceName="" messageTemplateResourceType="" tag=""
          type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.AndCompositeValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0"
          name="And Composite Validator" />
        <validator negated="false"
          messageTemplate="" messageTemplateResourceName=""
          messageTemplateResourceType="" tag="企画番号" name="AlphaNumericStringValidator"
          type="TERASOLUNA.FW.Common.Validation.Validators.AlphaNumericStringValidator, TERASOLUNA.FW.Common.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0" />
        <validator lowerBound="4" lowerBoundType="Inclusive" upperBound="4"
          upperBoundType="Inclusive" negated="false"
          messageTemplateResourceName="MSG_ERR_0019" messageTemplateResourceType="TourSample.Client.Resources.Messages"
          messageTemplate=""
          tag="企画番号" type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.StringLengthValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0"
          name="String Length Validator" />
        </validator>
      </validator>
    </property>
    <property name="TourCode">
      <validator messageTemplate=""
        messageTemplateResourceName="" messageTemplateResourceType="" tag=""
        type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.OrCompositeValidator, Microsoft.Practices.EnterpriseLibrary.Validation, Version=3.1.0.0, Culture=neutral, PublicKeyToken=b03f567ba0a98da0"
        name="Or Composite Validator" />
    </property>
  </properties>

```

図 15 入力値検証設定ファイルのサンプル

② イベントコントローラのプロパティ設定

ValidationFilePath プロパティに入力値検証設定ファイルのパスを設定する。

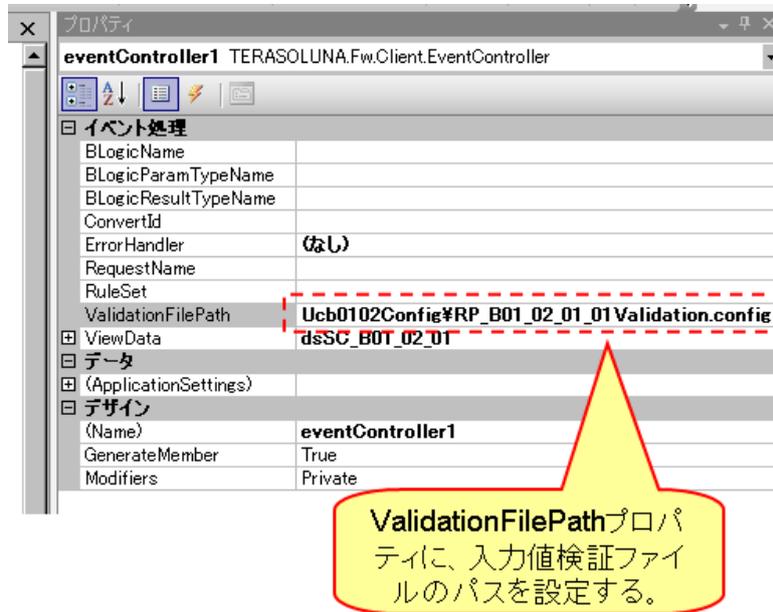


図 16 イベントコントローラのプロパティ設定(ValidationFilePath)

▶ データセット変換機能

◇ 概要

画面で入力された値をビジネスロジックに渡し、ビジネスロジックから返却された値を画面に反映する機能である(図 17)。データセット変換設定ファイルの内容に基づき、画面データセットとビジネスロジック入出力データセットの間でデータのやり取りが行われる。

データセット変換機能の詳細に関しては、『FB-02 データセット変換機能』を参照のこと。

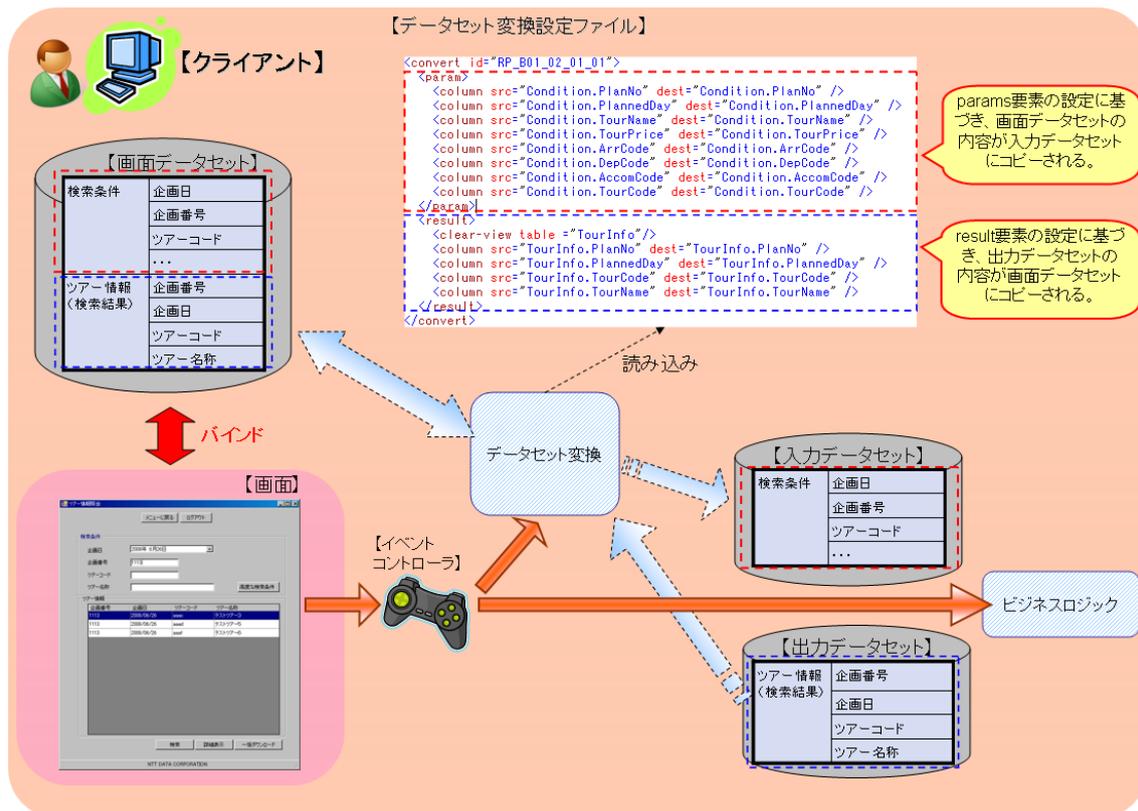


図 17 データセット変換の概念図

◇ 業務開発者のコーディングポイント
アプリケーション構成ファイルを記述する
アプリケーション構成ファイルでデータセット変換設定ファイルのパスを指定する。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- blogicConfiguration要素と解析クラスを宣言 -->
    <section name="blogicConfiguration" type="
TERASOLUNA.Fw.Client.Configuration.BLogic.BLogicConfigurationSection,
TERASOLUNA.FW.Client" />
    <!-- conversionConfiguration要素と解析クラスを宣言 -->
    <section name="conversionConfiguration" type="
TERASOLUNA.Fw.Client.Configuration.Conversion.ConversionConfigurationSection,
TERASOLUNA.FW.Client" />
  </configSections>
  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="BLogicConfiguration.config" />
    </files>
  </blogicConfiguration>
  <conversionConfiguration>
    <files>
      <!-- データセット変換設定ファイルのパス設定 -->
      <file path="Ucb9901Config¥ConversionConfiguration.config" />
      <file path="Ucb0101Config¥ConversionConfiguration.config" />
      <file path="Ucb0102Config¥ConversionConfiguration.config" />
    </files>
  </conversionConfiguration>
  .
  . (省略)
  .
</configuration>
```

①で作成したデータセット変換設定ファイルのパスを記述する。複数ファイルの指定が可能。

リスト 5 アプリケーション構成ファイルへのデータセット変換ファイルのパス設定例

③ データセット変換設定ファイルを記述する

データセット変換設定ファイルには、以下の設定を行う。

- ✓ **convert** 要素の **id** 属性に一意のコンバート ID を設定する
- ✓ **param** 要素の **src** 属性には、変換元のデータセット(画面データセット)のデータテーブルとデータカラムの名前をドット区切りで指定する
- ✓ **param** 要素の **dest** 属性には、変換先のデータセット(入力データセット)のデータテーブルとデータカラムの名前をドット区切りで指定する
- ✓ **result** 要素の **src** 属性には、変換先のデータセット(画面データセット)のデータテーブルとデータカラムの名前をドット区切りで指定する
- ✓ **param** 要素の **dest** 属性には、変換元のデータセット(出力データセット)のデータテーブルとデータカラムの名前をドット区切りで指定する

上記設定の **param** 要素の内容に応じて画面データセットの値が入力データセットにコピーされ、**result** 要素の内容に応じて出力データセットの値が画面データセットに反映される。

データセット変換設定ファイルは「出力ディレクトリにコピー」プロパティを「常にコピーする」にすること。データセット変換設定ファイルの生成単位は、プロジェクトの方針に従うこと。

```
<?xml version="1.0" encoding="utf-8" ?>
<conversionConfiguration
xmlns="http://www.terasoluna.jp/schema/ConversionSchema.xsd">

  <convert id="RP_B01_02_01_01">
    <param>
      <column src="Condition.PlanNo" dest="Condition.PlanNo" />
      <column src="Condition.PlannedDay" dest="Condition.PlannedDay" />
      <column src="Condition.TourName" dest="Condition.TourName" />
      <column src="Condition.TourPrice" dest="Condition.TourPrice" />
      <column src="Condition.ArrCode" dest="Condition.ArrCode" />
      <column src="Condition.DepCode" dest="Condition.DepCode" />
      <column src="Condition.AccomCode" dest="Condition.AccomCode" />
      <column src="Condition.TourCode" dest="Condition.TourCode" />
    </param>
    <result>
      <clear-view table="TourInfo"/>
      <column src="TourInfo.PlanNo" dest="TourInfo.PlanNo" />
      <column src="TourInfo.PlannedDay" dest="TourInfo.PlannedDay" />
      <column src="TourInfo.TourCode" dest="TourInfo.TourCode" />
      <column src="TourInfo.TourName" dest="TourInfo.TourName" />
    </result>
  </convert>

  <convert id="RP_B01_02_01_03">
    <param>
      <column src="TourInfo.TourCode" dest="TourInfo.TourCode" />
    </param>
    <result>
      <column src="TourInfo.TourCode" dest="TourInfo.TourCode" />
    </result>
  </convert>

</conversionConfiguration>
```

画面データセットから、ビジネスロジックの入力データセットに渡す値を設定する。

一意のコンバートIDを付与する。

ビジネスロジックの出力データセットから画面データセットにコピーする値を設定する。

リスト 6 データセット変換設定ファイルの記述例

- ④ イベントコントローラのプロパティを設定する
ConvertId プロパティに①で記述したコンバート ID を設定する。

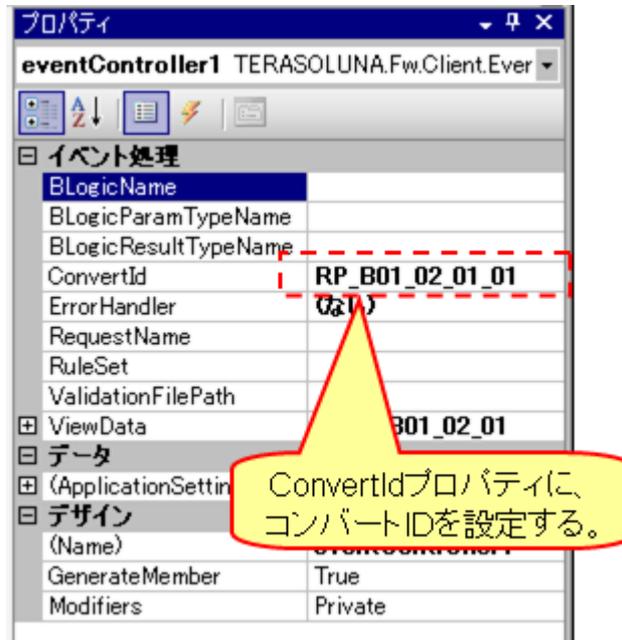


図 18 イベントコントローラのプロパティ設定(ConvertId)

▶ イベント実行機能

◇ 概要

イベントコントローラによってイベント処理が実行する際に、既定のタイミングにイベントハンドラを追加できる機能を提供する。イベントハンドラの追加は Visual Studio のデザイナから追加する。本機能を用いることで、業務開発者が様々なタイミングで画面操作などの処理を実装することができる。例えば、標準で提供している通信処理用ビジネスロジックを使用する際、通信前や通信後などに画面操作に関する処理を実装する場合に、これらイベントにイベントハンドラを追加して画面操作に関する処理を実装する。

同期処理の場合、6つのイベントを扱うことができる。

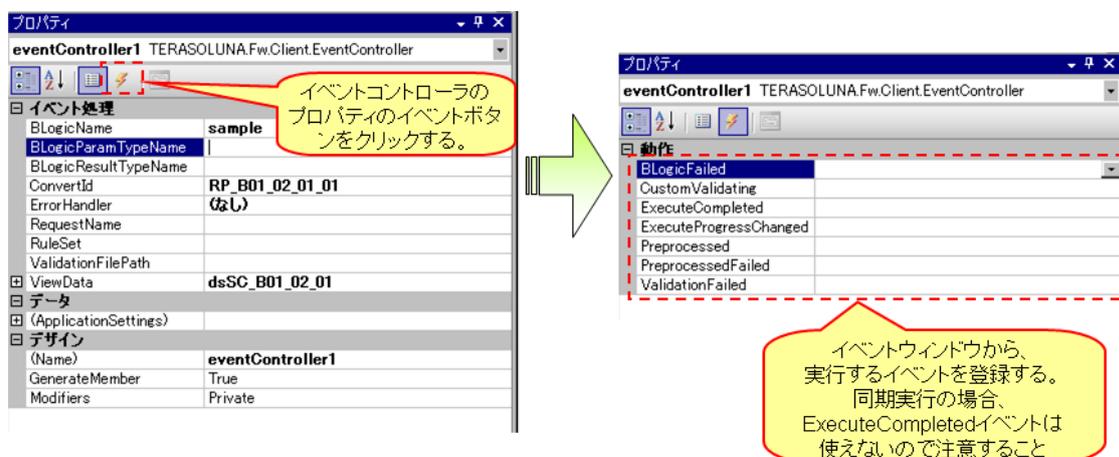


図 19 イベントコントローラのイベント登録方法

◇ イベントコントローラのイベント一覧

イベントコントローラから実行可能なイベントの一覧を以下に記す

表 6 イベントコントローラのイベント一覧

項番	イベント名	説明
1	CustomValidating	カスタムチェックイベント。 イベント処理機能が実行する入力値検証の直後に発生する。 独自の入力チェック(関連チェックなど)を行う場合に利用する。
2	ValidationFailed	入力値検証失敗イベント。 CustomValidating を含む、入力値検証でエラーとなった場合に発生する。
3	Preprocessed	前処理イベント。 全ての入力チェックが成功し、ビジネスロジック入力オブジェクトが作成された段階で発生する。ビジネスロジック入力オブジェクトに対して独自の処理を行う場合に利用する。
4	PreprocessedFailed	前処理失敗イベント。 前処理イベント中でエラーとなった場合に発生する。
5	BLogicFailed	ビジネスロジック失敗イベント。 ビジネスロジックが失敗(BLogicResult の ResultString が”success”以外)した場合に発生する。
6	ExecuteCompleted	非同期処理完了イベント。 ExecuteAsync によって実行された非同期処理が完了した場合に発生する。 本イベントは同期処理の場合は実行されない。
7	ExecuteProgressChanged	進行状況通知イベント。 ビジネスロジックの進行状況が変化した場合に発生する。ビジネスロジックが IProgressChangedEventInvoker インターフェイスを実装している場合のみ利用できる。

◇ カスタムチェックイベント

● 概要

本イベントは、入力値検証機能で入力チェックが行われた後のタイミングで発生する。本イベントを使用することで、入力値検証を使わずに、業務開発者がハードコーディングで入力チェックを実装できる。主に、入力値検証ではカバーできないチェック(関連チェックなど)を行う場合に使用する。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

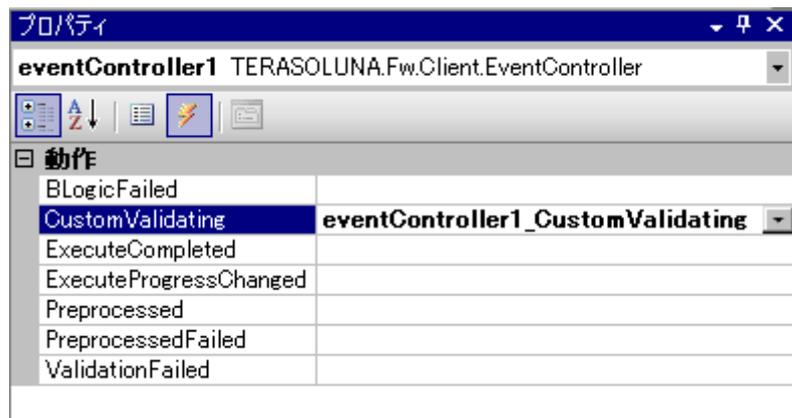


図 20 イベントコントローラのイベント(CustomValidating)

② イベントハンドラを実装する

①で追加したイベントハンドラに、ハードコーディングでチェックロジックを実装する。

カスタムチェックを行った結果、入力値検証エラーとするには、**ValidationResult** プロパティから取得した入力値検証結果オブジェクトの **Errors** プロパティに **ValidationMessageInfo** のインスタンスを設定する。エラーとした場合、後に **ValidationFailed** イベントが発生する。**ValidationResult**、**MessageInfo**、**ValidationMessageInfo** の詳細に関しては、『CM-02 入力値検証機能』を参照のこと。

引数の **CustomValidatingEventArgs** の詳細については、「表 7 CustomValidatingEventArgsのプロパティ」を参照のこと。

ここでは、画面データセットに対してハードコーディングでチェックを行い、エラーを追加する実装例を示す。

```
/// <summary>
/// カスタムチェックを行う。
/// </summary>
private void eventController1_CustomValidating(object sender,
CustomValidatingEventArgs e)
{
    // 画面データセットから値を取得する
    string planNo = dsSC_B01_02_01.Condition[0].PlanNo;
    string tourCode = dsSC_B01_02_01.Condition[0].TourCode;

    // 企画番号が空かつ、ツアーコードが空の場合、入力チェックエラーとする
    if (string.IsNullOrEmpty(planNo) && string.IsNullOrEmpty(tourCode))
    {
        ValidationMessageInfo info =
            new ValidationMessageInfo(
                "PlanNo",
                "企画番号とツアーコードの両方未入力にすることはできません",
                "Condition[1]/PlanNo");

        // エラーを追加する
        e.ValidationResult.Errors.Add(info);
    }
}
```

リスト 7 カスタムチェックイベントの実装例

- CustomValidatingEventArgsのプロパティ
- CustomValidatingEventArgs の定義するプロパティの一覧を以下に示す。

表 7 CustomValidatingEventArgs のプロパティ

項番	プロパティ名	説明
1	ValidationResult	イベント処理における入力値検証処理の結果を格納したイベント処理実行結果オブジェクト。(読み取り専用)

◇ 入力値検証失敗イベント

● 概要

本イベントは、入力値検証(もしくはカスタムチェックイベント)で入力チェックエラーが発生し、フレームワークによるエラー処理機能が実行された後のタイミングで発生する。

入力値検証に失敗した内容に応じて、処理を行う場合に使用する。たとえば、入力値検証に失敗した場合に、本イベント内で任意のテキストボックスにフォーカスを当てるなどの処理を実装することができる。

画面イベント(イベントコントローラを実行するクリックイベントなど)の中でも同様の処理を記述できるので、画面イベントと入力値検証失敗イベントのどちらで処理するのかは、プロジェクトで方針を定める必要がある。詳細は「エラー処理機能」の節を参照のこと。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

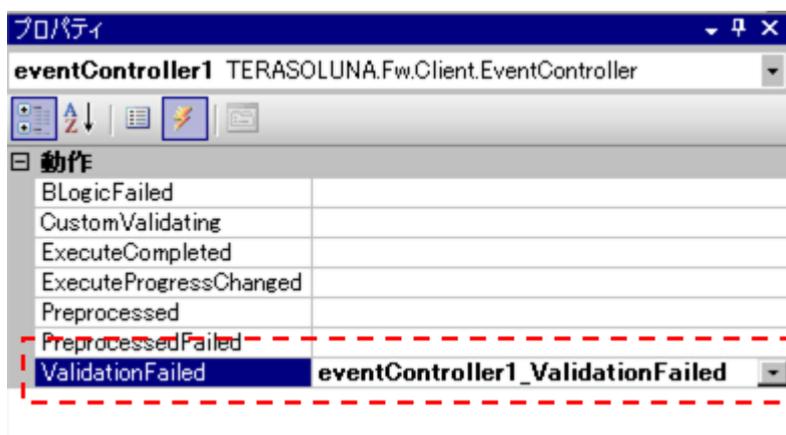


図 21 イベントコントローラのイベント(ValidationFailed)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。

引数の `ValidationFailedEventArgs` の詳細に関しては、「表 8 `ValidationFailedEventArgs`のプロパティ」を参照のこと。

ここでは、入力値検証に失敗した場合に、最初に見つかったエラーに対応するテキストボックスにフォーカスを当てる実装例を示す。

```
/// <summary>
/// 入力値検証失敗イベント。
/// 本サンプルコードでは、一番最初に見つかったエラーに対応するテキストボッ
/// クスにフォーカスを当てる。
/// (入力値検証機能を用いた場合、データセットのカラムの順番にエラーが入っ
/// てる)
/// </summary>
private void eventController1_ValidationFailed(object sender,
ValidationFailedEventArgs e)
{
    foreach(MessageInfo info in e.ExecutionResult.Errors)
    {
        if ("PlanNo".Equals(info.Key))
        {
            // 企画番号のテキストボックスにフォーカスを当てる
            this.ActiveControl = this.txtPlanNo;
            break;
        }
        else if ("TourCode".Equals(info.Key))
        {
            // 企画番号のテキストボックスにフォーカスを当てる
            this.ActiveControl = this.txtTourCode;
            break;
        }
    }
}
```

リスト 8 入力値検証失敗イベントの実装例

- ValidationFailedEventArgsのプロパティ
ValidationFailedEventArgs の定義するプロパティの一覧を以下に示す。

表 8 ValidationFailedEventArgs のプロパティ

項番	プロパティ名	説明
1	ExecutionResult	全ての入力値検証処理の結果を格納したイベント処理実行結果オブジェクト。(読み取り専用)

◇ 前処理イベント

● 概要

本イベントは、入力値検証でエラーとならず、『FB-02 データセット変換機能』を用いたデータセット変換が正常に完了し、ビジネスロジック入力オブジェクトが生成されたタイミングで発生する。

ビジネスロジックに渡すパラメータを途中で変更する(画面データセットが変換されたビジネスロジック入力データセットの内容を通信前に加工する)場合などに利用する。

本イベント内で、PreprocessedEventArgs から取得できる ExecutionResult の Errors プロパティに MessageInfo のインスタンスを追加すると、フレームワークによるエラー処理機能が実行され、ビジネスロジックは実行されない。その場合、前処理失敗イベントが発生する。

エラー処理機能の詳細に関しては、『FA-02 エラー処理機能』を参照のこと。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

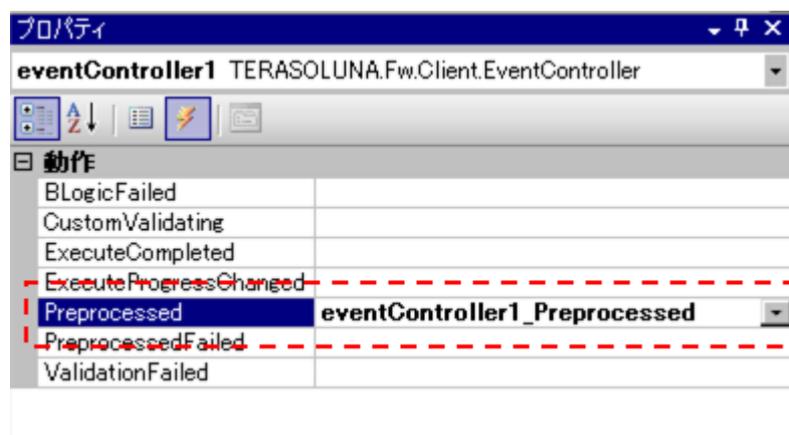


図 22 イベントコントローラのイベント(Preprocessed)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。

画面イベント(イベントコントローラを実行するクリックイベントなど)の中でも同様の処理を記述できるが、前処理イベントを使えば画面データセットに影響を与えずに、ビジネスロジックに渡す入力データをカスタマイズできるので安全である。

引数のPreprocessedEventArgsの詳細に関しては、「表 9 PreprocessedEventArgsのプロパティ」を参照のこと。

ここでは、画面データセットにバインドされたラジオボタンの内容を通信前にサーバが受信できる形に変換する(この例ではラジオボタンがバインドされたカラムの内容をもとに、TourPrice カラムに値を格納する)実装例を示す。

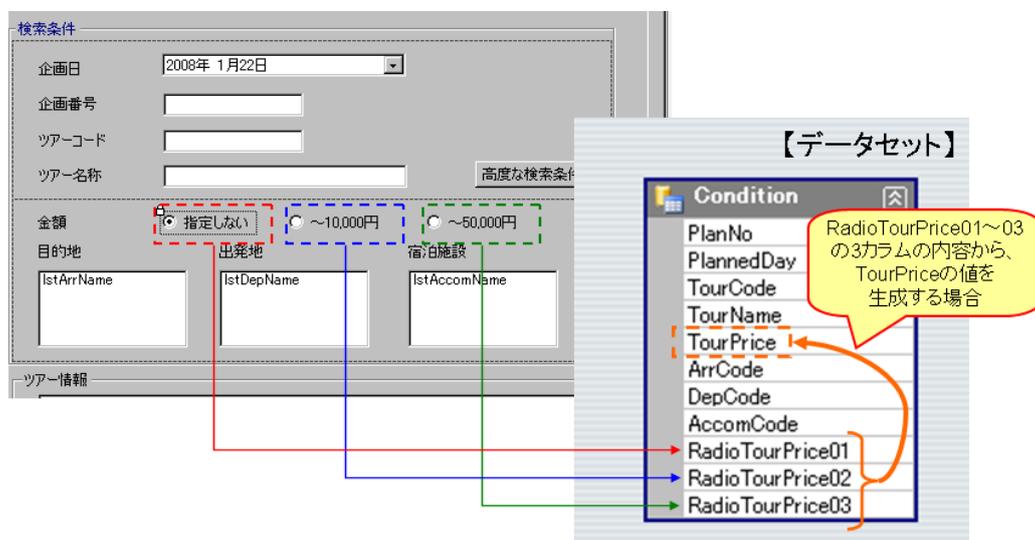


図 23 ラジオボタンと画面データセットのバインド

```
/// <summary>
/// 前処理イベント。
/// ラジオボタンの内容をもとに、TourPriceカラムに値を設定する。
/// </summary>
private void eventController1_Preprocessed(object sender,
PreprocessedEventArgs e)
{
    SC_B01_02_01Ds inputDataSet = e.BLogicParamData as SC_B01_02_01Ds;

    // 入力データセットのラジオボタンの内容に応じて、TourPriceカラムに
    // ~の値を格納する
    if (inputDataSet.Condition[0].RadioTourPrice01)
    {
        // 「指定しない」が選択された場合
        dsSC_B01_02_01.Condition[0].TourPrice = 0;
    }
    else if (inputDataSet.Condition[0].RadioTourPrice02)
    {
        // 「～10,000円」が指定された場合
        dsSC_B01_02_01.Condition[0].TourPrice = 10000;
    }
    else
    {
        // 「～50,000円」が指定された場合
        dsSC_B01_02_01.Condition[0].TourPrice = 50000;
    }
}
```

リスト 9 前処理イベントの実装例

- PreprocessedEventArgsのプロパティ

PreprocessedEventArgs の定義するプロパティの一覧を以下に示す。

表 9 PreprocessedEventArgs のプロパティ

項番	プロパティ名	説明
1	ExecutionResult	全ての入力値検証処理の結果を格納したイベント処理実行結果オブジェクト。(読み取り専用)
2	BLogicParamData	ビジネスロジック実行に利用されるビジネスロジック入力オブジェクト。

◇ 前処理失敗イベント

● 概要

本イベントは、前処理イベントが失敗し、フレームワークによるエラー処理機能が実行された後のタイミングで発生する。具体的には、前処理イベントにおいて、**ExecutionResult** インスタンスの **Errors** プロパティに **MessageInfo** インスタンスが設定された場合、本イベントが発生する。

エラー処理機能の詳細に関しては、『FA-02 エラー処理機能』を参照のこと。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

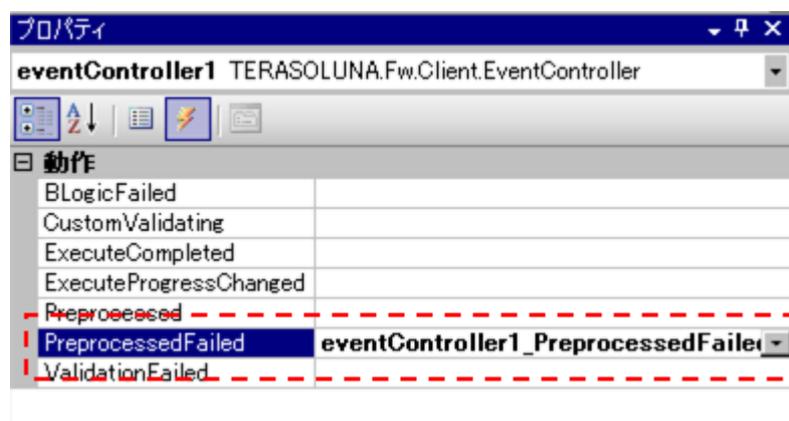


図 24 イベントコントローラのイベント(PreprocessedFailed)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。

引数の **PreprocessedFailedEventArgs** の詳細に関しては、「表 10 **PreprocessedFailedEventArgs**のプロパティ」を参照のこと。

```
/// <summary>
/// 前処理失敗イベント。
/// </summary>
private void eventController1_PreprocessedFailed(object sender,
PreprocessedFailedEventArgs e)
{
    // 任意の処理を記述する(説明は省略する)
}
```

リスト 10 前処理失敗イベントの実装例

- PreprocessedFailedEventArgsのプロパティ
PreprocessedFailedEventArgs の定義するプロパティの一覧を以下に示す。

表 10 PreprocessedFailedEventArgs のプロパティ

項番	プロパティ名	説明
1	ExecutionResult	全ての入力値検証処理の結果を格納したイベント処理実行結果オブジェクト。(読み取り専用)
2	BLogicParamData	ビジネスロジック実行に利用されるビジネスロジック入力オブジェクト。(読み取り専用)

◇ ビジネスロジック失敗イベント

● 概要

本イベントは、ビジネスロジックの実行に失敗(ビジネスロジックが正常に結果を返し、ビジネスロジック実行結果オブジェクトの **ResultString** プロパティに”success”以外の文字列が設定されていた場合)し、フレームワークによるエラー処理機能が実行された後のタイミングで発生する。

たとえば、非同期処理時にビジネスロジックの結果に応じて、ポップアップ画面でメッセージなどを表示した場合などに使用される。(非同期処理時は、ビジネスロジックが UI スレッドで実行されないため、画面系のロジックはビジネスロジック以外の場所に実装する必要がある)。

BLogicFailedEventArgs から取得できる **ExecutionResult** には、ビジネスロジックの実行結果が格納される。

エラー処理機能の詳細に関しては、『FA-02 エラー処理機能』を参照のこと。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

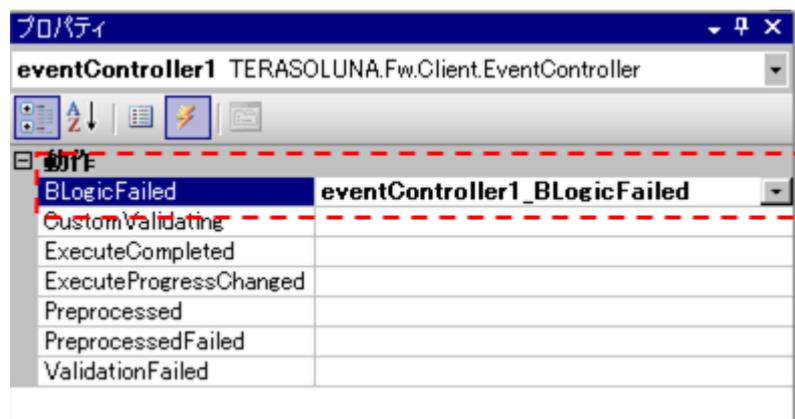


図 25 イベントコントローラのイベント(BLogicFailed)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。引数の `BLogicFailedEventArgs` の詳細に関しては、「表 11 `BLogicFailedEventArgs` のプロパティ」を参照のこと。

ここでは、非同期処理時にビジネスロジックの結果に応じて、ポップアップ画面でメッセージを表示する実装例を示す。

```
/// <summary>
/// ビジネスロジック失敗イベント。
/// </summary>
private void eventController1_BLogicFailed(object sender, BLogicFailedEventArgs e)
{
    if (e.ExecutionResult.Errors.Count > 0)
    {
        if ("E0001".Equals(e.ExecutionResult.Errors[0].Key))
        {
            MessageBox.Show("業務エラー①が発生しました。");
        }
        else if ("E0002".Equals(e.ExecutionResult.Errors[0].Key))
        {
            MessageBox.Show("業務エラー②が発生しました。");
        }
        else if ("E0003".Equals(e.ExecutionResult.Errors[0].Key))
        {
            MessageBox.Show("業務エラー③が発生しました。");
        }
    }
}
```

リスト 11 ビジネスロジック失敗イベントの記述例

● `BLogicFailedEventArgs` のプロパティ

`BLogicFailedEventArgs` の定義するプロパティの一覧を以下に示す。

表 11 `BLogicFailedEventArgs` のプロパティ

項番	プロパティ名	説明
1	ExecutionResult	失敗したビジネスロジックの実行結果を格納したイベント処理実行結果オブジェクト。(読み取り専用)

◇ 進行状況通知イベント

● 概要

本イベントは、ビジネスロジックが業務処理の進行状況をユーザに通知する処理を実装するためのイベントである。本イベントの実行タイミングはビジネスロジックの実装に依存する。たとえば、『FC-01 XML 通信機能』で提供する通信用ビジネスロジックでは、8k バイトの通信を行うごとに本イベントが発生する。

画面から同期処理を実行した場合、進行状況通知イベント内で画面操作を行ったとしても、同期処理が完了しないと画面が再描画されないために逐次的・段階的な表示とはならない。

通信処理など時間のかかるビジネスロジックの進行状況を、プログレスバーなどを利用して利用者に通知する場合には、非同期実行を利用する。本イベントの説明は、「使用方法(非同期処理)」の「進行状況通知イベント」を参照のこと。

▶ エラー処理機能

◇ 概要

エラー処理機能は、イベント処理全体で統一的なエラー処理とエラークリア処理を行う機能である。エラー処理は、ビジネスロジックやイベント実行中にエラーが発生した場合に、イベント処理共通のエラー処理として呼ばれる。エラー処理では、主に入力値検証エラーなどのエラー情報を画面へ表示する処理を行う。また、エラークリア処理は、イベント処理実行の都度、画面へ表示されているエラー情報をクリアするために、イベント処理実行直後に呼ばれる。

◇ エラー処理機能を有効にする

● 業務開発者のコーディングポイント

エラー処理機能を有効にするためには、イベントコントローラの **ErrorHandler** プロパティに **ErrorHandler** 実装クラスのインスタンスを設定する。通常は **ErrorHandler** プロパティには、**FormBase** 派生クラスである画面のインスタンスを設定する。TERASOLUNA Client Framework for .NET では、画面は **FormBase** 派生クラスとして作成することを想定しており、**FormBase** は標準で **ErrorHandler** インターフェイスを実装している。



図 26 ErrorHandler プロパティに FormBase 派生クラスのインスタンスを設定

イベント処理機能では、エラー処理機能を実装したオブジェクトにエラー処理を委譲している。したがって、イベントコントローラの **ErrorHandler** プロパティに画面のインスタンス(**FormBase** 派生クラス)を設定した場合は、画面の **HandleError** メソッドにエラー処理が委譲される。また、エラークリア処理も、画面の **ClearError** メソッドに処理が委譲される。以下にエラー処理が呼ばれるケースとエラークリア処理が呼ばれるケースの一覧を示す。

表 12 エラー処理が呼ばれるケース

項番	ケース
1	入力値検証エラーが発生したとき
2	前処理完了イベントでエラーが発生したとき
3	ビジネスロジックの実行が失敗したとき (ビジネスロジックの実行結果を表す結果文字列が”success”でないとき)

表 13 エラークリア処理が呼ばれるケース

項番	ケース
1	イベント処理を実行した直後

- エラー処理機能の標準実装

FormBase でのエラー処理の標準実装では、画面データセットに対して入力値検証エラーがあったときに、エラーパスにしたがって画面データセットへエラーメッセージを設定する処理を行っている。ErrorProvider と合わせて使用することで、入力値検証エラーをユーザに伝えることができる。



図 27 ツールボックスから ErrorProvider を追加



図 28 ErrorProvider で入力値検証エラーを表示した例

また、エラークリア処理の標準実装では、画面データセットに設定されたエラーメッセージを全てクリアする処理を行っている。ErrorProvider と合わせて使用している場合、ここでは画面に表示された ErrorProvider のエラーアイコンをクリアする。

- エラー処理機能のカスタマイズ

イベント処理共通のエラー処理やエラークリア処理をカスタマイズしたい場合は、FormBase が実装している **IErrorHandler** インターフェイスのメソッド(**HandleError** メソッド、**ClearError** メソッド)をオーバーライドすること。

イベント処理共通のエラー処理ではなく、個別のイベント処理でのエラー処理をカスタマイズしたい場合は、入力値検証失敗イベント、前処理失敗イベント、ビジネスロジック失敗イベントを利用すること。これらのイベントは、イベント処理共通のエラー処理機能が呼ばれた後に実行される。

▶ その他の機能

☆ Items

● 概要

TERASOLUNA Client Framework for .NET では画面データセットとビジネスロジック間のデータのやり取りをデータセットで行うが、Items を用いてやり取りを行うこともできる。

● Itemsを使用する場合の注意点

イベントコントローラの Items は自動的にクリアされないことに注意すること。もし Items の値をクリアする場合は、業務開発者がイベントコントローラ実行前に Clear メソッドを呼び出しておく必要がある。

- 業務開発者のコーディングポイント

- ① イベントコントローラの Items を設定・取得する

ビジネスロジックに Items を使って値を渡す場合は、Execute メソッド実行前にイベントコントローラの Items プロパティに値を設定する。

ビジネスロジックから返却された Items の値を取得する場合は、Execute メソッド実行後にイベントコントローラの Items プロパティから取得する。

```
/// <summary>
/// 検索ボタンのクリックイベント。
/// </summary>
private void search_Click(object sender, EventArgs e)
{
    // 事前にイベントコントローラのItemsをクリアする
    eventController1.Items.Clear();

    // Itemsに値を設定する
    eventController1.Items["key01"] = "value01";

    // イベントコントローラを用いてイベント処理を実行する
    ExecutionResult result = eventController1.Execute();

    if (result.Success)
    {
        // 成功時の処理
        // Itemsの値を取得する
        MessageBox.Show("Itemsの値→" + eventController1.Items["key02"]);
    }
    else
    {
        // 失敗時の処理
        MessageBox.Show("イベント処理が失敗しました");
    }
}
```

リスト 12 クリックイベントでの、Items に設定された値の取得・設定例

② ビジネスロジックで Items の設定・取得を行う

イベントコントローラで設定された Items は、BLogicParam の Items プロパティから取得する。
イベントコントローラに Items の値を渡す場合は、BLogicResult の Items プロパティに設定する。

```
public class SampleBLogic : IBLogic
{
    public BLogicResult Execute(BLogicParam param)
    {
        // イベントコントローラで設定された Items の値を取得する
        string inputValue = param.Items["key01"] as string;

        Console.WriteLine("イベントコントローラから取得した値→" + inputValue);

        // イベントコントローラに渡す Items の値を設定する
        BLogicResult result = new BLogicResult(BLogicResult.SUCCESS);
        result.Items["key02"] = "value02";

        return result;
    }
}
```

イベントコントローラで設定された値
を Items から取得する。

イベントコントローラに
渡す値を Items に設定する。

リスト 13 ビジネスロジックでの Items に設定された値の取得・設定例

■ 使用方法(非同期実行)

◆ 概要

基本的な処理フローは同期処理と変わらない。

非同期(ワーカースレッド)で実行されるのはビジネスロジックのみであり、ビジネスロジックの実行前に行われる入力値検証やデータセット変換、ビジネスロジック進行状況の通知については同期処理実行時と同様に UI スレッドで実行される。

同期処理と異なる点は、非同期実行結果の確認方法である。`ExecuteAsync` メソッドを利用した非同期処理の結果の処理は、完了イベントの引数に格納された `ExecutionResult` インスタンスを確認することで行う。

1つのイベントコントローラに対して、同時に複数の非同期実行処理を行うことはできない。非同期処理の実行中に `ExecuteAsync` メソッドが呼び出された場合、例外がスローされる。イベントコントローラが非同期実行中であるかどうかを確認するには `IsBusy` プロパティを用いる。

イベント処理フローの詳細に関しては、「図 33 非同期処理のイベント処理フロー」を参照のこと。

◆ 実装方法

▶ イベントコントローラの実行

業務開発者は画面を実装する際、イベントコントローラを利用して業務処理を呼び出す。そのためには、まず①イベントコントローラをツールボックスから画面へ貼り付ける。次に②イベントコントローラのプロパティを設定する。続けて、③イベントコントローラに非同期実行完了イベントの登録を行う。最後に④ボタンクリックなどの非同期実行画面イベントよりイベントコントローラの `ExecuteAsync` メソッドを呼び出し、完了イベントの引数に格納された `ExecutionResult` インスタンスを用いて結果の確認を行う。

非同期実行を行う場合は、基本的に非同期実行完了イベントも合わせて使用することになる。(非同期実行完了イベントでしか、ビジネスロジックの実行結果を確認することができない)

以下に、イベントコントローラを利用して業務処理の呼び出しを行う作業手順を示す。

① 画面へのイベントコントローラの追加

本手順に関しては、同期実行処理と同様であるので、「[図 4 イベントコントローラの追加と設定](#)」を参照のこと。

② イベントコントローラのプロパティ設定

本手順に関しては、同期実行処理と同様であるので、「[図 5 イベントコントローラのプロパティ設定](#)」を参照のこと。

イベントコントローラに設定するプロパティについては、「[表 1 イベントコントローラのプロパティ](#)」を参照のこと。

③ 非同期処理完了イベントの登録

本手順に関しては、「[非同期処理完了イベント](#)」を参照のこと。

④ イベントコントローラの ExecuteAsync メソッドの実行

ボタンクリックのイベントハンドラなどから、イベントコントローラのExecuteAsyncメソッドを実行する。非同期実行完了イベントに渡ってくる引数ExecuteCompletedEventArgsに格納されるExecutionResultには、イベント処理の実行結果が格納される。成功したかどうかは、ExecutionResultのSuccessプロパティによって確認することができる。ExecutionResultの詳細に関しては、「イベントコントローラのExecuteメソッドの実行」の「戻り値の確認」を参照のこと。

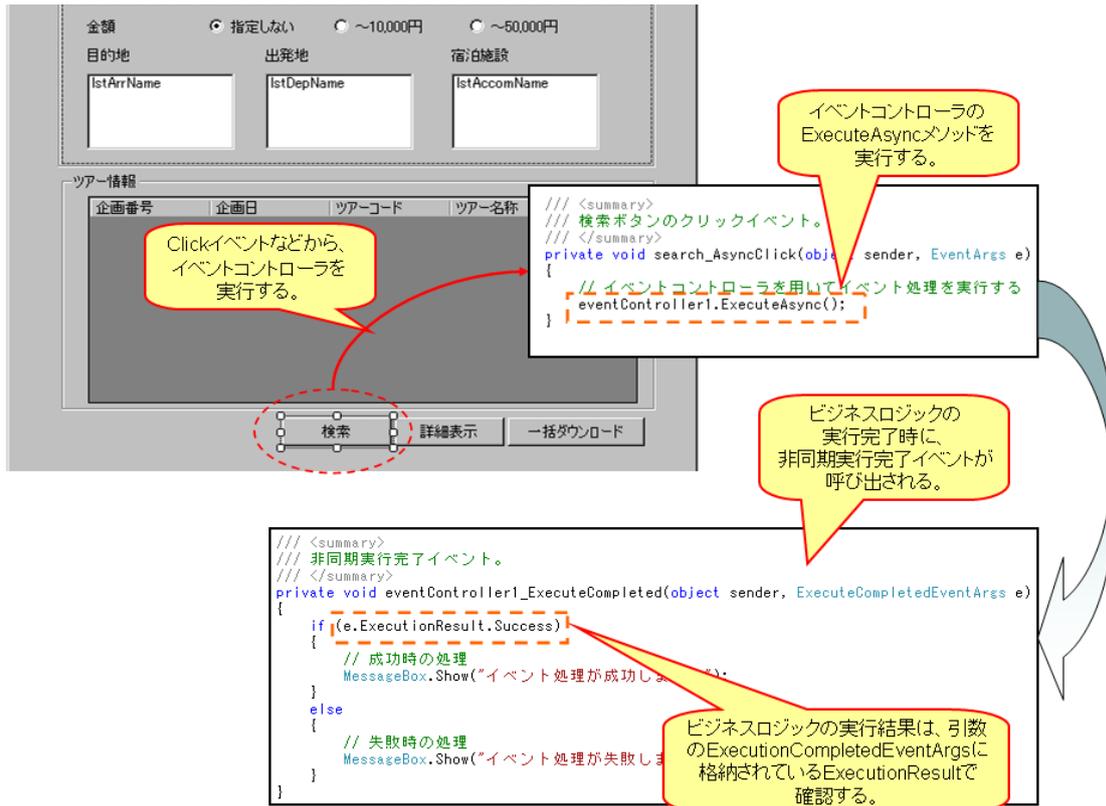


図 29 イベント処理の呼び出し(非同期)

◇ キャンセル処理

● 概要

非同期実行時かつ、ビジネスロジックが `ICancelable` インターフェイスを実装している場合、非同期実行を行っているイベントコントローラの `ExecuteAsyncCancel` メソッドを呼び出すことでビジネスロジックのキャンセル処理が可能となる。

TERASOLUNA が提供する通信ビジネスロジックは標準で `ICancelable` インターフェイスを実装しているので、キャンセルが可能である。

キャンセルされた場合、`ExecutionResult` の `Cancel` プロパティに `true` が設定される。

● 実業務開発者のコーディングポイント

① イベントコントローラの `ExecuteAsyncCancel` メソッドを呼び出す

```
/// <summary>
/// キャンセルを実行するボタンのクリックイベント。
/// </summary>
private void buttonCancel_Click(object sender, EventArgs e)
{
    eventController1.ExecuteAsyncCancel();
}
```

リスト 14 イベントコントローラのキャンセルメソッドを呼び出す実装例

② 非同期実行完了イベントを実装する

キャンセル処理が行われた場合は、`ExecutionResult` プロパティにアクセスすると例外がスローされるので、`Cancelled` プロパティが `false` でないことを確認した後で、`ExecutionResult` を扱うこと。

```
/// <summary>
/// 非同期実行完了イベント。
/// </summary>
private void eventController1_ExecuteCompleted(object sender,
ExecuteCompletedEventArgs e)
{
    // キャンセルされた場合は処理を終了する
    if (e.Cancelled)
    {
        return;
    }

    if (e.ExecutionResult.Success)
    {
        // 成功時の処理
        MessageBox.Show("イベント処理が成功しました");
    }
    else
    {
        // 失敗時の処理
        MessageBox.Show("イベント処理が失敗しました");
    }
}
}
```

リスト 15 非同期実行完了イベントの実装例

▶ イベント実行機能

◇ 概要

イベントコントローラによってイベント処理が実行する際に、既定のタイミングにイベントハンドラを追加できる機能を提供する。イベントハンドラの追加は **Visual Studio** のデザイナーから追加する。本機能を用いることで、業務開発者が様々なタイミングで画面操作などの処理を実装することができる。例えば、標準で提供している通信処理用ビジネスロジックを使用する際、通信前や通信後などに画面操作に関する処理を実装する場合に、これらイベントにイベントハンドラを追加して画面操作に関する処理を実装する。

非同期処理の場合、7つのイベントを扱うことができる。非同期処理完了イベントは実行結果の確認のため、必ず使用することになる。

◇ イベントコントローラのイベント一覧

イベントコントローラから実行可能なイベントに関しては、「表 6 イベントコントローラのイベント一覧」を参照のこと。

◇ 非同期処理完了イベント

● 概要

本イベントは、非同期で実行されたビジネスロジックの実行が完了したタイミングで発生する。本イベントは同期処理実行では使用することができない点に注意すること。

非同期実行されたビジネスロジックの実行結果は、本イベントの引数 `ExecuteCompletedEventArgs` に格納される `ExecutionResult` でしか確認することができないので、非同期実行とこのイベントはセットで使用する。

入力値検証、ビジネスロジック入力オブジェクトの生成、前処理が成功した場合、ビジネスロジックが成功、失敗、あるいは例外をスローした場合であっても必ずこのイベントが発生する。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

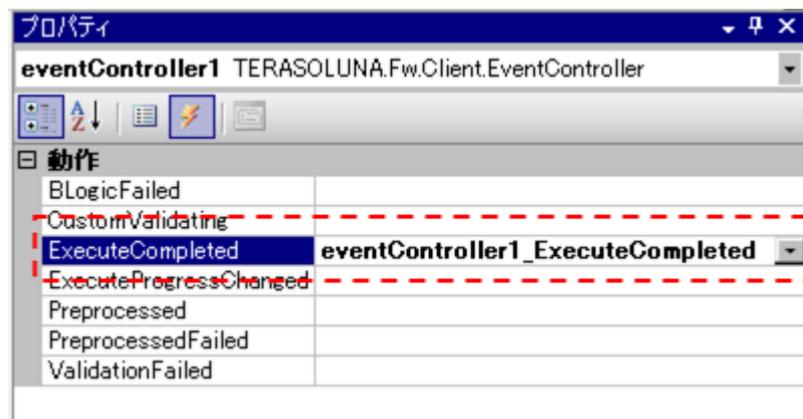


図 30 イベントコントローラのイベント(ExecuteCompleted)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。

引数の `ExecuteCompletedEventArgs` に格納されている `ExecutionResult` をもとにビジネスロジックの実行結果を確認する。

引数 `ExecuteCompletedEventArgs` の詳細に関しては、「表 14 `ExecuteCompletedEventArgs`のプロパティ」を参照のこと。

ここでは、非同期処理実行完了後の `ExecutionResult` の結果に応じて、表示するメッセージを切り替える実装例を示す。

```

/// <summary>
/// 非同期実行完了イベント。
/// </summary>
private void eventController1_ExecuteCompleted(object sender,
ExecuteCompletedEventArgs e)
{
    if (e.ExecutionResult.Success)
    {
        // 成功時の処理
        MessageBox.Show("イベント処理が成功しました");
    }
    else
    {
        // 失敗時の処理
        MessageBox.Show("イベント処理が失敗しました");
    }
}

```

引数の `ExecutionCompletedEventArgs` に格納されている `ExecutionResult` を使い、ビジネスロジックの実行結果を確認する。

リスト 16 非同期実行完了イベントの実装例

- `ExecuteCompletedEventArgs`のプロパティ

`ExecuteCompletedEventArgs` の定義するプロパティの一覧を以下に示す。

表 14 `ExecuteCompletedEventArgs` のプロパティ

項番	プロパティ名	説明
1	<code>ExecutionResult</code>	ビジネスロジックの実行結果を格納したイベント処理実行結果オブジェクト。(読み取り専用) 本クラスの詳細に関しては、 <code>ExecutionResult</code> のプロパティを参照のこと。
2	<code>Cancelled</code>	キャンセルされたかどうかを示すフラグ。キャンセルされた場合、 <code>true</code> が設定される。
3	<code>Error</code>	非同期実行時に発生した例外

◇ 進行状況通知イベント

● 概要

本イベントは、ビジネスロジックが業務処理の進行状況をユーザに通知する処理を実装するのに利用するイベントである。本イベントの実行タイミングはビジネスロジックの実装に依存する。たとえば、『FC-01 XML 通信機能』で提供する通信用ビジネスロジックでは、8k バイトの通信を行うごとに本イベントが発生する。

主に通信処理など実行に時間がかかるビジネスロジックの通信状態をステータスバーなどでリアルタイム表示するために使用する。

TERASOLUNA が提供する通信ビジネスロジックは標準で本イベントを実行するが、もし自分で実装したビジネスロジックから本イベントを呼び出したい場合は、開発者がビジネスロジック内に本イベントを実行するロジックを記述する必要がある。

● 業務開発者のコーディングポイント

① イベントハンドラを追加する

Visual Studio のデザイナを用いて、イベントハンドラを追加する。

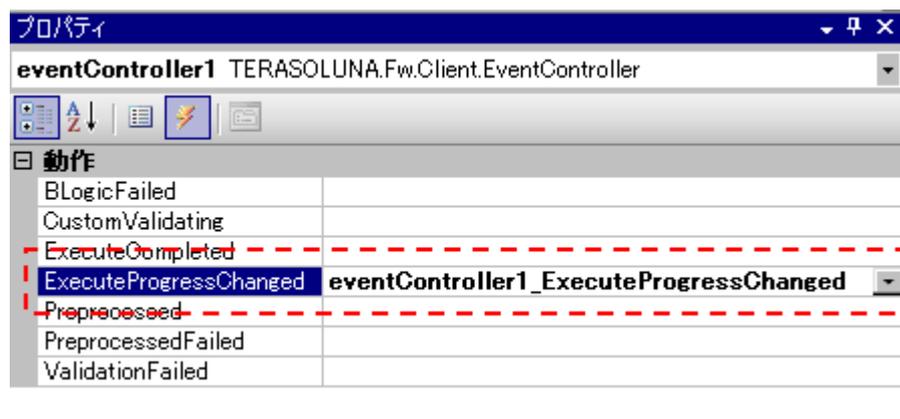


図 31 イベントコントローラのイベント(ExecuteProgressChanged)

② イベントハンドラを実装する

ハードコーディングでイベントハンドラを実装する。

引数の `ExecuteProgressChangedEventArgs` の詳細に関しては、「表 15 `ExecuteProgressChangedEventArgs`のプロパティ」を参照のこと。

ここでは、TERASOLUNA が提供する通信用ビジネスロジック実行時に、進行状況をプログレスバーに表示する実装例を示す。

```

/// <summary>
/// 進行状況通知イベントを用いて、プログレスバーを表示する
/// </summary>
private void eventController1_ExecuteProgressChanged(object sender,
ExecuteProgressChangedEventArgs e)
{
    /// プログレスバーのValueに進行状況のパーセンテージを格納する
    progressBar1.Value = e.ProgressPercentage;
}

```

リスト 17 進行状況通知イベントの実装例

● `ExecuteProgressChangedEventArgs`のプロパティ

`ExecuteProgressChangedEventArgs` の定義するプロパティの一覧を以下に示す。

表 15 `ExecuteProgressChangedEventArgs` のプロパティ

項番	プロパティ名	説明
1	Items	ビジネスロジックと情報をやりとりするための <code>IDictionary</code> 。(読み取り専用) ファイルダウンロード用ビジネスロジックを使用する場合は、本プロパティを用いてダウンロードファイル名を設定する必要がある。
2	ProgressPercentage	ビジネスロジックの進捗状況を表す百分率で表す整数値。(読み取り専用)

■ 内部構成

◆ 詳細フロー

(1) 同期処理のイベント処理フロー

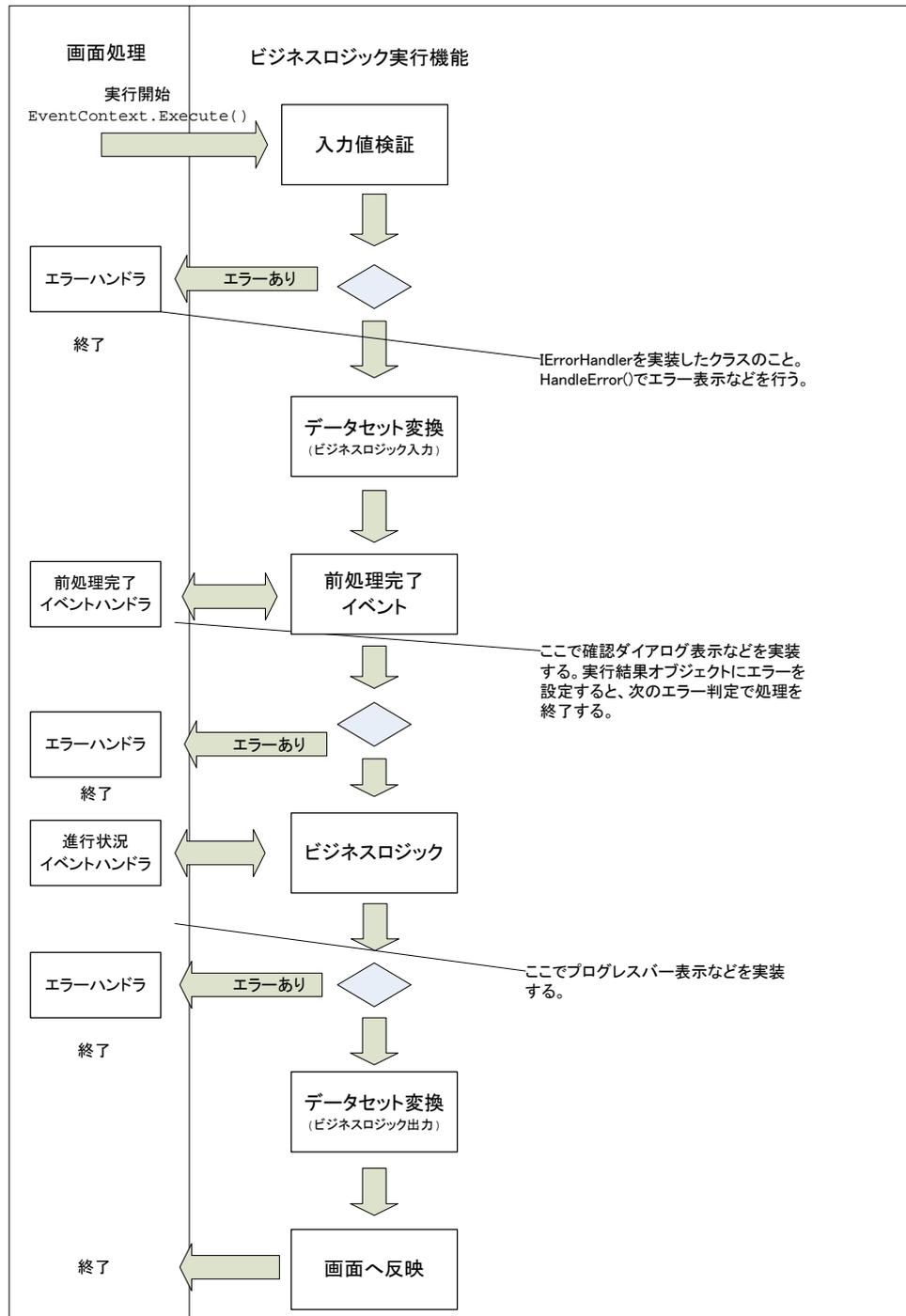


図 32 同期処理のイベント処理フロー

(2) 非同期処理のイベント処理フロー

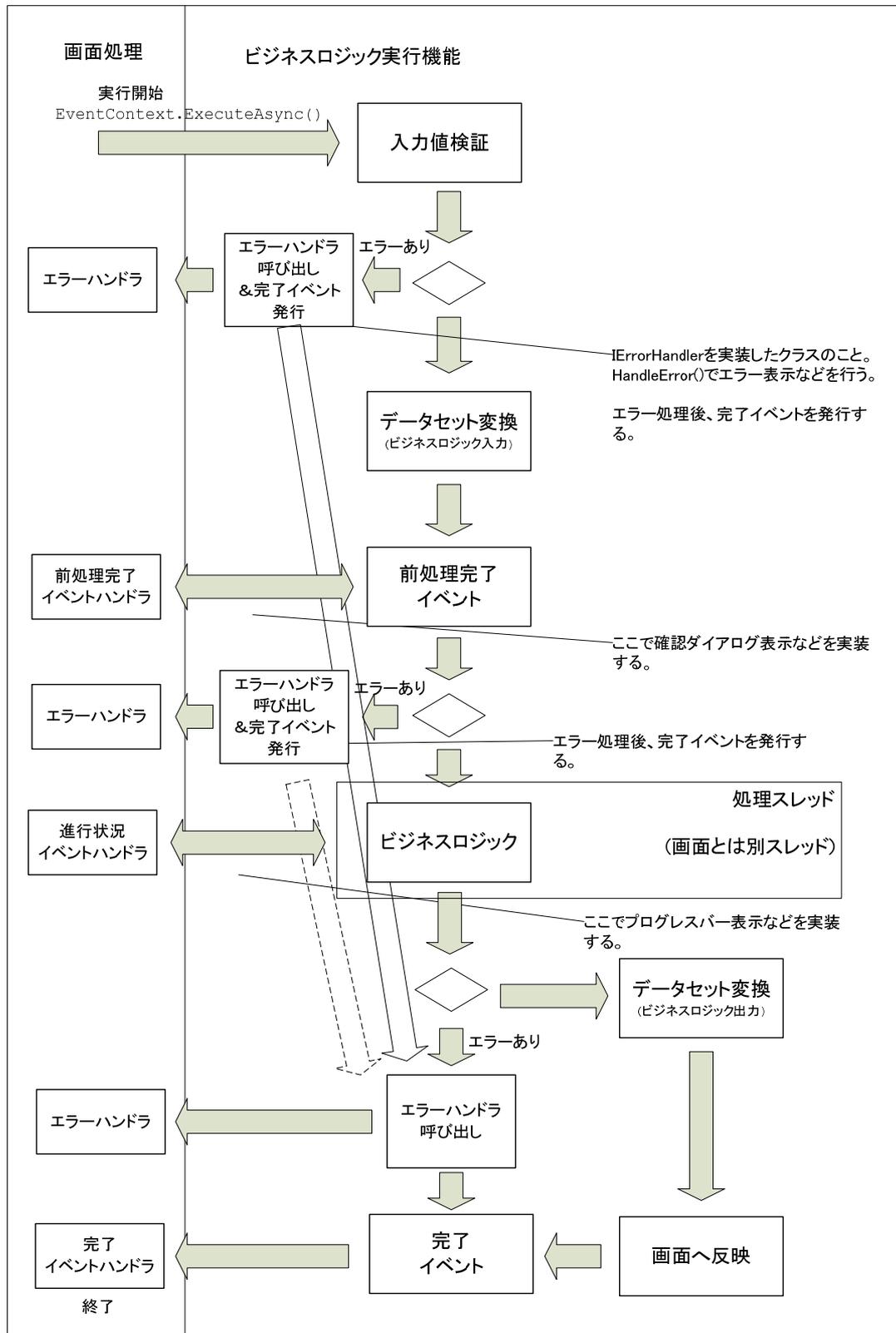


図 33 非同期処理のイベント処理フロー

◆ 構成クラス

(1) Client 名前空間

表 16 構成クラス一覧(Client)

項番	クラス名	説明
1	EventController	イベント処理機能を提供するクラス。 イベントコントローラのプロパティで、実行する機能を設定する。
2	Coordinator	EventController から呼び出され、入力値検証、データセット変換、ビジネスロジック実行を順番に実行する。
3	IValidator	入力値検証を実行するインターフェイス。 Coordinator から呼び出される。
4	VabValidator	入力値検証設定ファイルで検証ルールを指定し、DataSet に対する入力値検証を行う IValidator 実装クラス。 内部処理に Validation Application Block で提供される Validator 機能を利用している。
5	IConverter	データセット変換機能を提供するインターフェイス。 Coordinator から呼び出される。
6	DataSetConverter	データセット変換機能を提供する IConverter 実装クラス。
7	ClientBLogicBase	ビジネスロジックの基底クラス。 IBLogic を実装している。 キャンセル処理と進行状況イベントの通知処理を実装している。
8	CommunicateBLogicBase	サーバ通信を行うビジネスロジックの基底クラス。 ClientBLogicBase を継承している。
9	DataSetXmlCommunicateBLogic	XML データの送受信を行うビジネスロジッククラス。 CommunicateBLogicBase を継承している。
10	BinaryFileDownloadBLogic	XML データの送信、ファイルの受信を行うビジネスロジッククラス。 CommunicateBLogicBase を継承している。
11	BinaryFileUploadBLogic	ファイルの送信、XML データの受信を行うビジネスロジッククラス。 CommunicateBLogicBase を継承している。
12	MultipartUploadBLogic	マルチパートデータの送信、XML の受信を行うビジネスロジッククラス。 CommunicateBLogicBase を継承している。

(2) Common 名前空間

表 17 構成クラス一覧(Common)

項番	クラス名	説明
1	IBLogic	ビジネスロジックが実装すべきインターフェイス。 Coordinator から呼び出される。
2	NopBLogic	何も処理を行わないビジネスロジック。 IBLogic を実装している。 BLogicResultString として必ず”success”を返却する。 イベントコントローラの BLogicName プロパティを空文字列とした(何も設定しない)場合、自動的にこのビジネスロジックが使用される。

■ 拡張ポイント

(1) 利用するコーディネータ実装クラスの変更

アプリケーション構成ファイルの `appSettings` セクションに `Coordinator` を拡張したクラスの完全修飾名を指定することで、利用するコーディネータの実装を差し替えることができる。完全修飾名の指定を省略した場合は、TERASOLUNA が提供する `Coordinator` クラスが使用される。なお、完全修飾名を指定する `appSettings` セクションのキーは「`CoordinatorTypeName`」である。

※本機能説明書ではコーディネータの差し替えは行わないことを前提に解説している。

以下に、コーディネータの差し替えを設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="CoordinatorTypeName" value="Sample.CoordinatorEx, Sample"/>
  </appSettings>
</configuration>
```

リスト 18 利用するコーディネータクラスの設定例

(2) 利用するビジネスロジック生成クラスの変更

アプリケーション構成ファイルの `appSettings` セクションに `BLogicFactory` を拡張したクラスの完全修飾名を指定することで、利用するビジネスロジック生成クラスの実装を差し替えることができる。完全修飾名の指定を省略した場合は、TERASOLUNA が提供する `BLogicFactory` クラスが使用される。なお、完全修飾名を指定する `appSettings` セクションのキーは「`BLogicFactoryTypeName`」である。

※本機能説明書ではビジネスロジック生成クラスの差し替えは行わないことを前提に解説している。

以下に、ビジネスロジック生成クラスの差し替えを設定する例を示す。

```
<configuration>
  <appSettings>
    <add key="BLogicFactoryTypeName"
        value="Sample.BLogic.BLogicFactoryEx, Sample" />
  </appSettings>
</configuration>
```

リスト 19 利用するビジネスロジック生成クラスの設定例

(3) 利用するデータセット変換クラスの変更

アプリケーション構成ファイルの `appSettings` セクションに `IConverter` を実装したクラスの完全修飾名を指定することで、利用するデータセット変換クラスの実装を差し替えることができる。詳細は「**FB-02** データセット変換機能」の拡張ポイントを参照のこと。

(4) 利用する入力値検証クラスの変更

アプリケーション構成ファイルの `appSettings` セクションに `IValidator` を実装したクラスの完全修飾名を指定することで、利用するデータセット変換クラスの実装を差し替えることができる。詳細は「**CM-02** 入力値検証機能」の拡張ポイントを参照のこと。

■ 関連機能

- FA-02 エラー処理機能
- FA-03 拡張フォーム機能
- FB-02 データセット変換機能
- FC-01 XML 通信機能
- FC-02 ファイルアップロード機能
- FC-03 ファイルダウンロード機能
- CM-02 入力値検証機能
- CM-04 ビジネスロジック生成機能

FB-02 データセット変換機能

■ 概要

本機能は、スキーマが異なる 2 つのデータセット間で、相互にデータを変換する機能を提供する。

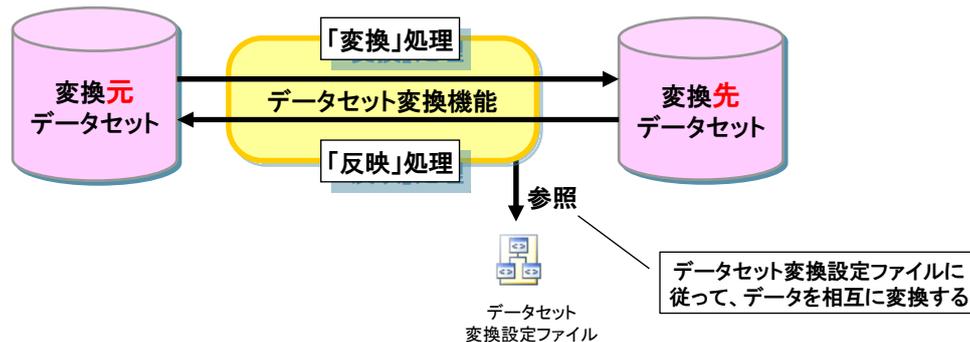


図 1 動作イメージ

XML 形式のデータセット変換設定ファイルに定義したデータセット変換情報に従って、変換元データセットと変換先データセット間でデータ変換を行う。本機能は主に「FA-01 画面遷移機能」、
「FB-01 イベント処理機能」から呼び出される。

データ変換では、データの流れの違いによって次のように処理の種類を定義している。

- 「変換」処理：変換元データセットから変換先データセットにデータ変換する処理
 - 変換先データセットのテーブルは空でなければならない
- 「反映」処理：変換先データセットから変換元データセットにデータ変換する処理
 - 変換元データセットのデータの扱いが 2 通りある
 - ◇ 変換元のデータを 1 度削除してから変換先のデータをコピーする
 - ◇ 変換先から変換元へデータを上書きする

■ 使用方法

◆ アプリケーション構成ファイル

本機能を有効にする場合、アプリケーション構成ファイル(App.config)にデータセット変換設定ファイルのパスを定義する。

表 1 アプリケーション構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name	○	構成要素名。 固定値。以下の値とする。 ConversionConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値。以下の値とする。 TERASOLUNA.Fw.Client.Configuration. Conversion.ConversionConfigurationSection,TERASOLUNA.Fw.Client
/configuration/conversionConfiguration/files/file			複数可
	path	○	データセット変換設定ファイルのパス

<pre> <?xml version="1.0" encoding="utf-8" ?> <configuration> <configSections> <section name="conversionConfiguration" type="TERASOLUNA.Fw.Client.Configuration.Conversion.ConversionConfigurationSection, TERASOLUNA.Fw.Client"/> </configSections> <conversionConfiguration> <files> <file path="config¥Converter1.config"/> <file path="config¥Converter2.config"/> </files> </conversionConfiguration> </configuration> </pre>	<p>conversionConfiguration 要素を利用するための設定</p>
	<p>データセット変換設定ファイルのパス</p>

リスト 1 アプリケーション構成ファイル設定例

◆ データセット変換設定ファイル

データセット変換設定ファイルにデータセット変換情報を定義する。データセット変換情報は複数定義することができ、コンバート ID によって管理する。1 つのデータセット変換情報には、1 組の変換元・変換先データセットに対する、「変換」処理と「反映」処理の設定情報を定義する。コンバート ID は全てのデータセット変換設定ファイルの中で一意でなければならない。

表 2 データセット変換設定ファイル

ノード	属性	必須	値
/conversionConfiguration	xmlns	○	xml スキーマ。 固定値。以下の値とする。 http://www.terasoluna.jp/schema/ConversionSchema.xsd
/conversionConfiguration /convert	id	○	データセット変換情報を管理するためのコンバート ID。一意な値。
/conversionConfiguration /convert/param		○	param 要素の定義は必須。
/conversionConfiguration /convert/param/column			複数可
	src	○	「変換」処理を行う際の変換元データセットの列名。“テーブル名.列名”で定義する。
	dest		「変換」処理を行う際の変換先データセットの列名。“テーブル名.列名”で定義する。省略した場合は src 属性に定義した列名となる。
/conversionConfiguration /convert/result		○	result 要素の定義は必須。
/conversionConfiguration /convert/result/clear-view	table		「反映」処理において、変換元データセットのデータを 1 度削除してから変換先のデータをコピーしたい場合、clear-view 要素を記述し、table 属性にテーブル名を指定する。 clear-view 要素を記述しない場合、変換先から変換元へデータが上書きされる。
/conversionConfiguration /convert/result/column			複数可
	src	○	「反映」処理を行う際の変換元データセットの列名。“テーブル名.列名”で定義する。
	dest		「反映」処理を行う際の変換先データセットの列名。“テーブル名.列名”で定義する。省略した場合は src 属性に定義した列名となる。

```
<?xml version="1.0" encoding="utf-8" ?>
<conversionConfiguration
xmlns="http://www.terasoluna.jp/schema/ConversionSchema.xsd">
  <convert id="getInfo">
    <param>
      <column src="InfoA.name" dest="InfoB.name" />
      <column src="InfoA.birth" dest="InfoB.birth" />
    </param>
    <result>
      <clear-view table="InfoA"/>
      <column src="InfoA.name" dest="InfoB.name" />
      <column src="InfoA.birth" dest="InfoB.birth" />
    </result>
  </convert>
  <convert id="getUser">
    <param>
      <column src="UserA.id" />
    </param>
    <result>
      <column src="UserA.id" />
    </result>
  </convert>
</conversionConfiguration>
```

コンバートID

param要素には「変換」処理の設定を定義する

result要素には「反映」処理の設定を定義する

column要素
src属性 : 変換元データセットの[テーブル名.カラム名]
dest属性: 変換先データセットの[テーブル名.カラム名]

dest属性を省略すると、srcに定義したカラムが変換先データセットのカラムとなる

リスト 2 データセット変換設定ファイル設定例

「変換」処理、「反映」処理、それぞれの処理イメージを以下に示す。なお、「反映」処理は `clear-view` 要素を指定する場合と指定しない場合を区別する。

- 「変換」処理

```
<convert id="foo">
  <param>
    <column src="DataTableA.Col2" dest="DataTableB.ColB" />
  </param>
</convert>
```

DataTableAテーブルのCol2カラムのデータを
DataTableBテーブルのColBカラムにコピーする

リスト 3 「変換」処理のデータセット変換設定ファイル設定例

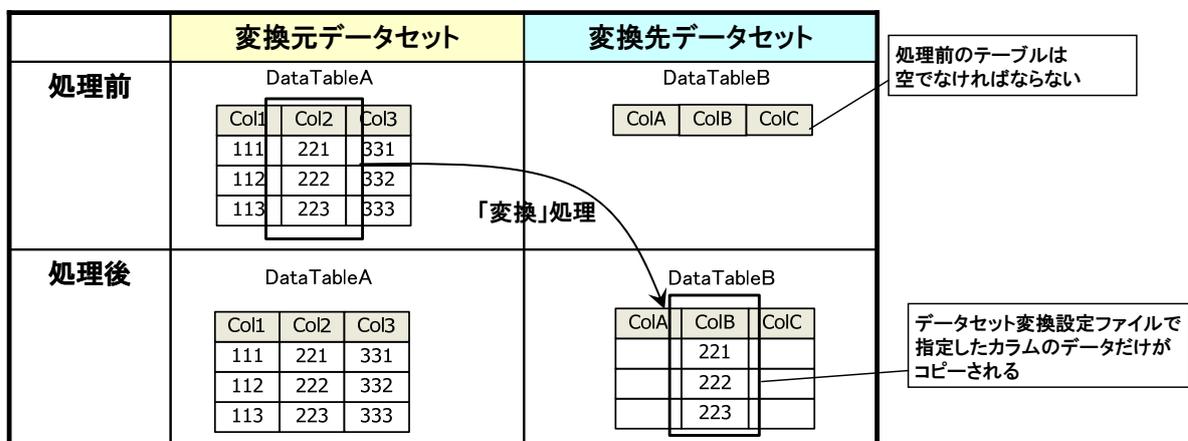
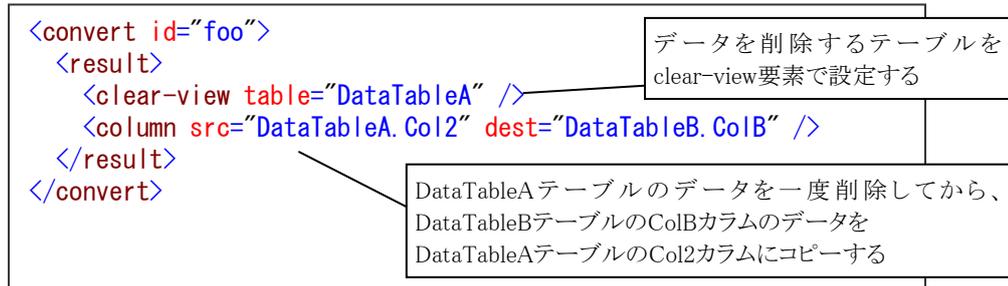


図 2 「変換」処理の処理イメージ

- 「反映」処理 (clear-view 要素指定有り)



リスト 4 「反映」処理 (clear-view 要素指定有り) のデータセット変換設定ファイル設定例

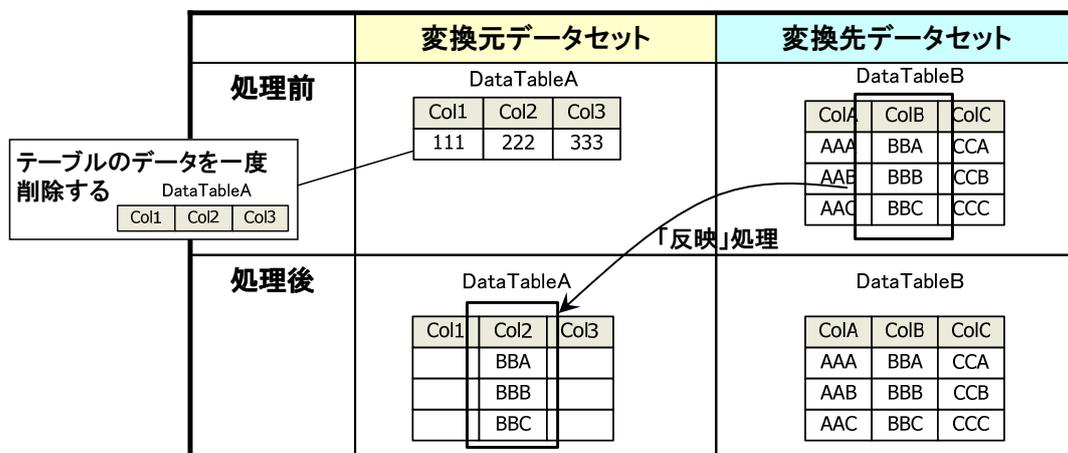


図 3 「反映」処理 (clear-view 要素指定有り) の処理イメージ

- 「反映」処理 (clear-view 要素指定無し)

```
<convert id="foo">
  <result>
    <column src="DataTableA.Col2" dest="DataTableB.ColB" />
  </result>
</convert>
```

DataTableBテーブルのColBコラムのデータを
DataTableAテーブルのCol2コラムに上書きする

リスト 5 「反映」処理 (clear-view 要素指定無し) のデータセット変換設定ファイル設定例

	変換元データセット	変換先データセット																								
処理前	DataTableA <table border="1"> <thead> <tr><th>Col1</th><th>Col2</th><th>Col3</th></tr> </thead> <tbody> <tr><td>111</td><td>221</td><td>331</td></tr> <tr><td>112</td><td>222</td><td>332</td></tr> <tr><td>113</td><td>223</td><td>333</td></tr> </tbody> </table>	Col1	Col2	Col3	111	221	331	112	222	332	113	223	333	DataTableB <table border="1"> <thead> <tr><th>ColA</th><th>ColB</th><th>ColC</th></tr> </thead> <tbody> <tr><td>AAA</td><td>BBA</td><td>CCA</td></tr> <tr><td>AAB</td><td>BBB</td><td>CCB</td></tr> <tr><td>AAC</td><td>BBC</td><td>CCC</td></tr> </tbody> </table>	ColA	ColB	ColC	AAA	BBA	CCA	AAB	BBB	CCB	AAC	BBC	CCC
Col1	Col2	Col3																								
111	221	331																								
112	222	332																								
113	223	333																								
ColA	ColB	ColC																								
AAA	BBA	CCA																								
AAB	BBB	CCB																								
AAC	BBC	CCC																								
処理後	DataTableA <table border="1"> <thead> <tr><th>Col1</th><th>Col2</th><th>Col3</th></tr> </thead> <tbody> <tr><td>111</td><td>BBA</td><td>331</td></tr> <tr><td>112</td><td>BBB</td><td>332</td></tr> <tr><td>113</td><td>BBC</td><td>333</td></tr> </tbody> </table>	Col1	Col2	Col3	111	BBA	331	112	BBB	332	113	BBC	333	DataTableB <table border="1"> <thead> <tr><th>ColA</th><th>ColB</th><th>ColC</th></tr> </thead> <tbody> <tr><td>AAA</td><td>BBA</td><td>CCA</td></tr> <tr><td>AAB</td><td>BBB</td><td>CCB</td></tr> <tr><td>AAC</td><td>BBC</td><td>CCC</td></tr> </tbody> </table>	ColA	ColB	ColC	AAA	BBA	CCA	AAB	BBB	CCB	AAC	BBC	CCC
Col1	Col2	Col3																								
111	BBA	331																								
112	BBB	332																								
113	BBC	333																								
ColA	ColB	ColC																								
AAA	BBA	CCA																								
AAB	BBB	CCB																								
AAC	BBC	CCC																								

「反映」処理

変換元と変換先でテーブルの行数が一致しない場合、例外をスローする

関係のないコラムのデータは影響を受けない

図 4 「反映」処理 (clear-view 要素指定無し) の処理イメージ

◆ 実装方法

データセット変換機能は、主に「FA-01 画面遷移機能」、「FB-01 イベント処理機能」から呼び出される。

- 「FA-01 画面遷移機能」からの呼び出し
 - FormForwarder コンポーネントの ConvertId プロパティに、データセット変換設定ファイルに定義したコンバート ID を設定する。
 - 画面遷移が実行されると、内部でデータセット変換機能が呼ばれ、遷移元画面のデータセットと遷移先画面のデータセット間でデータセット変換が行われる。
 - 詳細は「FA-01 画面遷移機能」を参照のこと。
- 「FB-02 イベント処理機能」からの呼び出し
 - EventController コンポーネントの ConvertId プロパティにデータセット変換設定ファイルに定義したコンバート ID を設定する。
 - イベント処理が実行されると、内部でデータセット変換機能が呼ばれ、イベント処理を実行した画面のデータセットとビジネスロジック入出力データセット間でデータセット変換が行われる。
 - 詳細は「FB-01 イベント処理機能」を参照のこと。

■ 内部構成

◆ 構成図

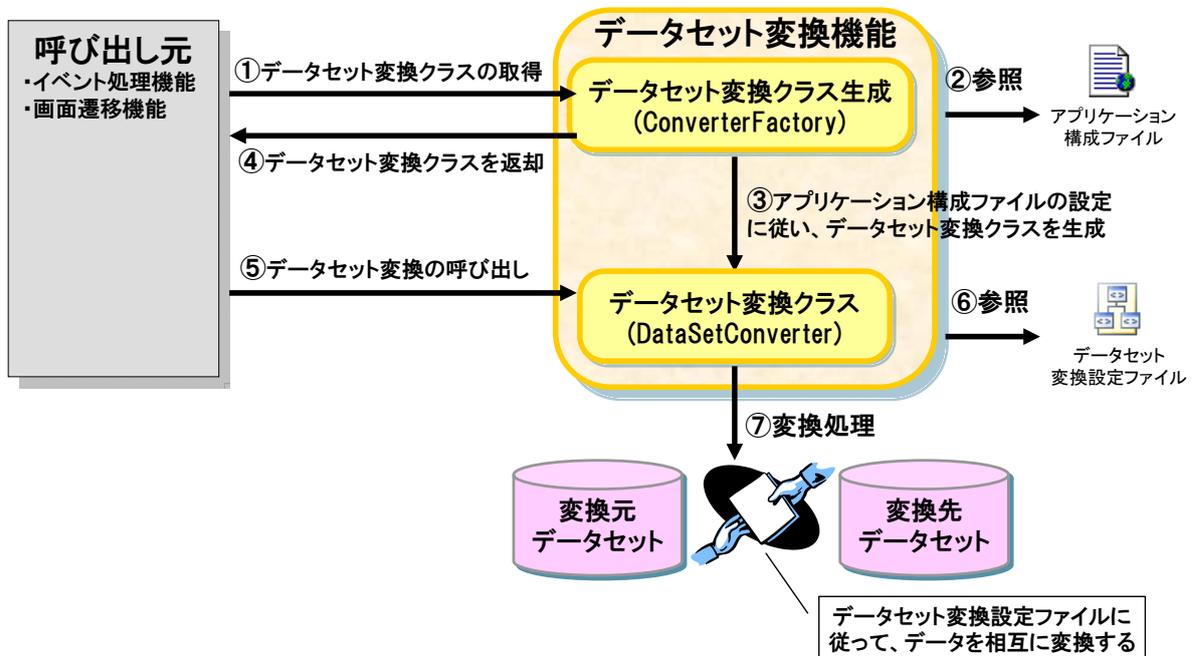


図 5 構成図

◆ 構成クラス

表 3 構成クラス一覧

項番	クラス名	概要
1	IConverter	データセット変換機能を提供するインターフェイス
2	ConverterFactory	IConverter 実装クラスのインスタンスを生成するファクトリクラス
3	ConversionException	データセット変換時に発生したエラーを表す例外クラス
4	DataSetConverter	データセット変換機能を提供する IConverter 実装クラス

■ 拡張ポイント

独自のデータセット変換機能をフレームワークに沿って利用する場合、`DataSetConverter` 派生クラスを作成するか、あるいは `IConverter` を実装したデータセット変換クラスを作成する。

使用するデータセット変換クラス(`IConverter` 実装クラス)を変更する場合は、アプリケーション構成ファイルの `appSettings` 要素に、“`ConverterTypeName`”をキーとして完全修飾型名を記述する。この設定をもとに、`ConverterFactory` がデータセット変換クラスのインスタンスを生成する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="conversionConfiguration"
      type="TERASOLUNA.Fw.Client.Configuration.Conversion.
        ConversionConfigurationSection, TERASOLUNA.Fw.Client"/>
  </configSections>
  <appSettings>
    <add key="ConverterTypeName"
      value="TERASOLUNA.Sample.Converter, TERASOLUNA.Sample"/>
  </appSettings>
</configuration>
```

使用するデータセット変換クラスの完全修飾型名

リスト 6 データセット変換クラスを変更する場合の設定例

■ 関連機能

- FA-01 画面遷移機能
- FB-01 イベント処理機能

FC-01 XML通信機能

■ 概要

本機能は、HTTP 通信を行い、TERASOLUNA Server Framework for Java/.NET で開発されたサーバ AP と連携して XML 電文の送受信する機能を提供する。

本機能を利用することで、イベントコントローラ経由、または XML 通信クラスを直接利用して、XML 通信処理を実行できる。

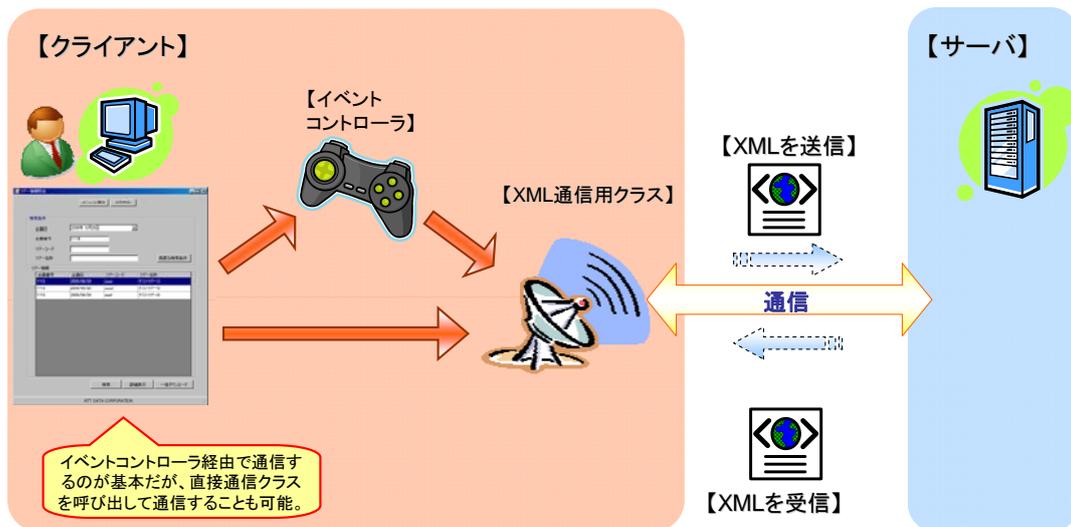


図 1 概念図

本資料では、イベントコントローラを使用するパターンと、直接通信クラスを使用するパターンに分けて説明を行う。

■ 事前準備

XML 通信機能を利用してサーバと XML 通信を行う場合、事前に以下の点について取り決めなければならない。

表 1 通信処理を実装する上で事前に決めておくこと

項番	項目	説明
1	通信先 URL	XML 電文を送信するサーバの URL。
2	リクエスト名	サーバで実行する処理を一意に識別するための文字列。
3	送信 XML スキーマ	サーバ AP に対して、POST によって送信される XML 電文の XML スキーマ。
4	受信 XML スキーマ	サーバ AP から受信する XML 電文の XML スキーマ。

送信 XML スキーマと受信 XML スキーマは、型指定されたデータセットとして定義することでオブジェクトとして扱することができる。また、これらのスキーマはサーバで実行する処理の種類に対応するため、1 つのリクエスト名に対して送信 XML スキーマと受信 XML スキーマを定義しなければならない。

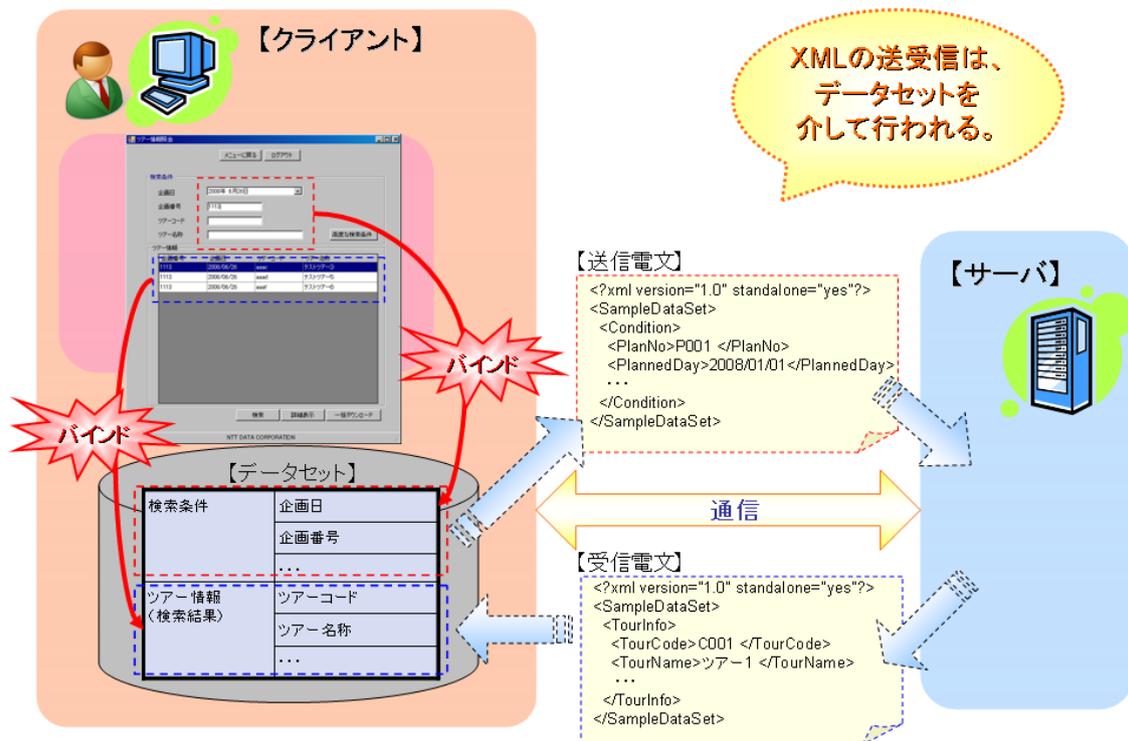


図 2 データセットと XML の関係

■ 使用方法(イベントコントローラ経由で通信を行う場合)

XML 通信する場合、ビジネスロジッククラスとして DataSetXmlCommunicateBLogic クラスを利用することで、イベントコントローラ経由で通信処理を実現することができる。

◆ アプリケーション構成ファイル

(1) 接続先URLの設定

アプリケーション構成ファイル(App.config)に接続先 URL を設定する。接続先 URL は、アプリケーション構成ファイルの appSettings/add 要素に、key 属性を”BaseUrl”として、value 属性に通信先 URL を設定する。

以下に、接続先 URL を設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="BaseUrl" value="http://terasoluna.local/index.aspx"/>
  </appSettings>
</configuration>
```

リスト 1 アプリケーション構成ファイルに接続先 URL を設定する例

(2) リクエストタイムアウト時間の設定

リクエストタイムアウト時間を設定するには、アプリケーション構成ファイルの appSettings/add 要素に、key 属性を”RequestTimeout”として、value 属性にタイムアウト時間(ミリ秒)を設定する。

アプリケーション構成ファイルに設定したタイムアウト時間は、全リクエストに対して適用される。リクエストごとにタイムアウト時間を変更する場合は、後述の「リクエストごとにタイムアウト時間を変更する場合 - (3)リクエストごとにタイムアウト時間を変更する場合」を参照すること。

以下に、リクエストタイムアウト時間を設定する方法を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="RequestTimeout" value="30000"/>
  </appSettings>
</configuration>
```

リスト 2 アプリケーション構成ファイルにリクエストタイムアウト時間を設定する例

(3) ビジネスロジック設定ファイルとデータセット変換設定ファイルのパスの設定

XML 通信用ビジネスロジック(DataSetXmlCommunicateBLogic)を利用するには、利用するビジネスロジックをビジネスロジック設定ファイルに記述し、さらにビジネスロジック入力データセットへの変換設定をデータセット変換ファイルに記述する必要がある。

記述する各設定ファイルのパスは、アプリケーション構成ファイルで指定する。以下に、ビジネス

ロジック設定ファイルのパスとデータセット変換設定ファイルのパスを指定する例を示す。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
      type="TERASOLUNA.Fw.Common.Configuration.BLogic.
        BLogicConfigurationSection, TERASOLUNA.Fw.Common"/>
    <section name="conversionConfiguration"
      type="TERASOLUNA.Fw.Client.Configuration.Conversion.
        ConversionConfigurationSection, TERASOLUNA.Fw.Client"/>
  </configSections>

  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="Config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>

  <conversionConfiguration>
    <files>
      <!-- データセット変換定義ファイルのパス設定 -->
      <file path="Config¥ConverterConfiguration.config"/>
    </files>
  </conversionConfiguration>
</configuration>
```

リスト 3 アプリケーション構成ファイルにビジネスロジック設定ファイルのパスと
データセット変換設定ファイルのパスを記述する例

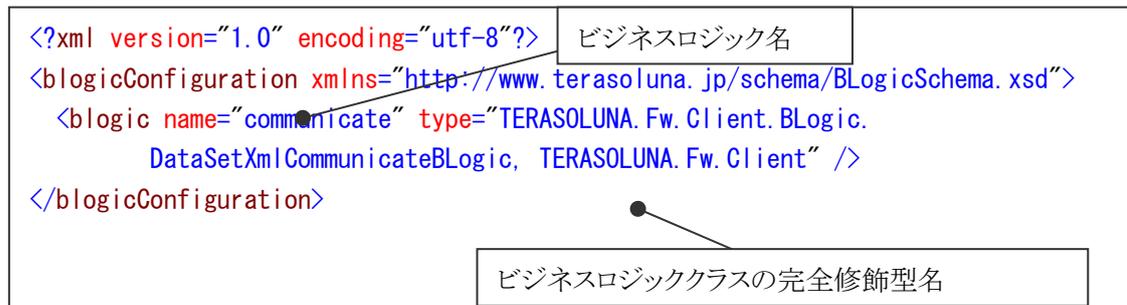
ビジネスロジック設定ファイルの詳細に関しては『CM-04 ビジネスロジック生成機能』を、データ
セット変換設定ファイルの詳細に関しては『FB-02 データセット変換機能』を参照すること。

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルに XML 通信用ビジネスロジックのビジネスロジック名とクラスの完全修飾型名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。

以下に、ビジネスロジック設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <blogic name="communicate" type="TERASOLUNA.Fw.Client.BLogic.
    DataSetXmlCommunicateBLogic, TERASOLUNA.Fw.Client" />
</blogicConfiguration>
```



リスト 4 ビジネスロジック設定ファイルの記述例

ビジネスロジック設定ファイルの詳細に関しては『CM-04 ビジネスロジック実行機能』を参照すること。

◆ データセット変換設定ファイル

データセット変換設定ファイルには、画面データセットからビジネスロジック入出力データセットへの変換設定を記述する。

以下に、データセット変換設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<conversionConfiguration
  xmlns="http://www.terasoluna.jp/schema/ConversionSchema.xsd">
  <convert id="sample">
    <param>
      . . . 省略 . . .
    </param>
    <result>
      . . . 省略 . . .
    </result>
  </convert>
</conversionConfiguration>
```

リスト 5 データセット変換設定ファイルの記述例

データセット変換設定ファイルの詳細については『FB-02 データセット変換機能』を参照すること。

◆ 実装方法

ファイルダウンロード処理を実装するには、(1)画面データセットの作成、(2)ビジネスロジック入出力データセットの作成、(3)イベントコントローラのプロパティの設定、(4)イベントコントローラの実行の順番で実装する。

(1) 画面データセットの作成

画面の情報を保持するデータセットを作成する。画面データセット作成方法の詳細に関しては、『FB-01 イベント処理機能』の「画面項目とデータセットのバインド(関連付け)」を参照すること。

(2) ビジネスロジック入出力データセットの作成

XML 電文としてサーバに送受信するデータを格納するためのデータセットを作成する。なお、画面データセットをそのまま利用することも出来る。その際は、後述のイベントコントローラのプロパティの設定において、“BLogicParamTypeName”、“BLogicResultTypeName”を空文字列にする。

(3) イベントコントローラのプロパティの設定

イベントコントローラのプロパティに、ファイルアップロードビジネスロジックの実行に必要な設定を行う。設定しなければならないプロパティを以下に示す。

表 2 設定しなければならないイベントコントローラのプロパティ

項番	プロパティ	説明
1	BLogicName	ビジネスロジック設定ファイルに記述した XML 通信用ビジネスロジックのビジネスロジック名
2	BLogicParamTypeName	ビジネスロジック入力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
3	BLogicResultTypeName	ビジネスロジック出力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
4	ConvertId	データセット変換設定ファイルに記述したコンバートID
5	RequestName	サーバ側での処理を識別するためのリクエスト名
6	ViewData	画面データセットのインスタンス

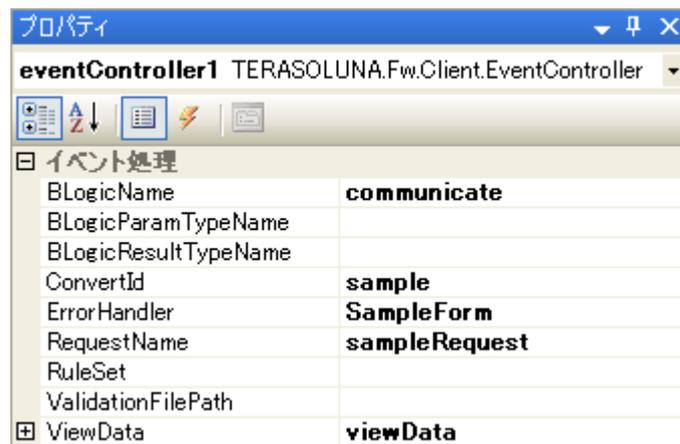


図 3 イベントコントローラのプロパティ設定例

(4) イベントコントローラの実行

イベントコントローラを実行し、実行結果の確認を行う実装例を以下に示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();

    if (result.Success)
    {
        // 成功時の処理
        // ...
    }
    else
    {
        // 失敗時の処理
        // ...
    }
}
```

リスト 6 XML 通信処理の実装例

イベントコントローラの実行後、**ExecutionResult** の **ResultString** プロパティにイベント処理が成功したかどうかを示す文字列が格納されるので、この値を確認する。失敗した場合は、エラー内容に応じた文字列(イベント処理を実行した結果)が格納されているので、適切にエラーハンドリング処理を実装する。

以下に、**ResultString** プロパティの値の一覧を示す。

表 3 ExecutionResult の ResultString 値一覧

項番	状態	値
1	単項目チェックエラー	“validationError”
2	カスタムチェックエラー	“validationError”
3	前処理エラー	“preprocessedError” または前処理イベントで設定する ResultString の値
4	ビジネスロジック実行失敗 (通信中に例外発生)	"communicationException"
5	ビジネスロジック実行失敗 (サーバ側で入力値検証エラー発生)	"serverValidateException"
6	ビジネスロジック実行成功	“success”
7	ビジネスロジック実行失敗 (サーバ側で業務エラー発生)	サーバでレスポンスヘッダに設定された任意の文字列 (例: "serviceException")

(5) その他

その他、XML 通信処理における設定を変更する方法を示す。

● リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更したい場合、イベントコントローラの Items プロパティに“RequestTimeout”をキーとして、リクエストタイムアウト時間を設定する。

以下に、実装例を示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // 省略

    // Itemsにリクエストタイムアウト時間 (5000ms) を設定する
    eventController1.Items["RequestTimeout"] = 5000;
    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();
}
```

リスト 7 リクエストタイムアウト時間を変更する場合の実装例

なお、Items プロパティとアプリケーション構成ファイルの両方にリクエストタイムアウト時間を設定した場合、Items プロパティに設定した値が利用される。

● リクエストヘッダの設定

通信時のリクエストヘッダに値を格納する場合は、IDictionary ジェネリックインターフェイス実装クラスのインスタンスを利用して、イベントコントローラの Items プロパティに設定する。

なお、リクエスト名はイベントコントローラのプロパティに設定すると自動的にリクエストヘッダに格納されて送信される。

```
Dictionary<string, object> requestHeaders
    = new Dictionary<string, object>();

requestHeaders.Add("customData", "foo");
```

リスト 8 リクエストヘッダの設定

- 通信のキャンセル

通信中に送受信を中止したい場合は、イベントコントローラの **Cancel** メソッドを呼び出すことで処理をキャンセルすることができる。キャンセルは非同期処理実行時のみ行うことができる。詳細は、『FB-01 イベント処理機能』を参照すること。

■ 使用方法(直接通信クラスを利用する場合)

XML 通信を行うには、DataSetXmlCommunicator クラスを直接利用して実現することもできる。DataSetXmlCommunicator クラスは、イベントコントローラ経由で通信する場合に利用した DataSetXmlCommunicateBLogic クラスの内部で利用している。

なお、DataSetXmlCommunicateBLogic クラスを直接利用して通信する場合は、例外処理などを自分で実装しなければならないことに注意すること。

◆ アプリケーション構成ファイル

本設定に関しては、「

使用方法(イベントコントローラ経由で通信を行う場合)」を参照すること。

◆ 実装方法

(1) XML通信処理の実装

XML 通信処理を実装するには、①DataSetXmlCommunicator クラスのインスタンスの作成、②送信用データセットの作成、③リクエストヘッダの作成、④実行とエラー処理、⑤受信用データセットの取得の順番で実装する。

以下に DataSetXmlCommunicator クラスを利用した XML 通信処理の実装例を示す。

```
// ①DataSetXmlCommunicatorクラスのインスタンスを作成
DataSetXmlCommunicator<Sample2Ds> communicator =
    new DataSetXmlCommunicator<Sample2Ds>();

// ②送信用データセットの作成
Sample1Ds paramData = new Sample1Ds();

// ※送信用データセットにデータを格納
//...

// ③リクエストヘッダの作成(リクエスト名には"sampleRequest"を指定)
IDictionary<string, string> requestHeaders =
    new Dictionary<string, string>();
requestHeaders.Add("requestName", "sampleRequest");

// ※必要に応じてリクエストヘッダに値を設定
//...

// ④実行とエラー処理
CommunicationResult result = null;
try
{
    result = communicator.Communicate(paramData, requestHeaders);
}
catch (ServerException ex)
{
    // サーバで発生したエラーの処理
    //...
}

// ⑤受信用データセットの取得
Sample2Ds resultData = (Sample2Ds)result.ResultData;
```

リスト 9 XML 通信処理の実装例

リスト 9では、Sample1Dsクラスのインスタンスを送信用データセットとしてAPサーバに送信し、リクエスト名"sampleRequest"で指定されたサーバ処理を実行、サーバ処理結果を受信用データセットとしてSample2Dsクラスのインスタンスで取得している。

- DataSetXmlCommunicator クラスのインスタンスの作成

XML 通信機能を利用するためには、DataSetXmlCommunicator クラスのインスタンスを利用する。最初に DataSetXmlCommunicator クラスのインスタンスを作成する。このとき、受信用

データセットの型を型パラメータとして指定する。なお、指定できる型パラメータは **DataSet** クラスとその派生クラスのみである。

- 送信用データセットの作成

サーバに送信するための送信用データセットを作成する。作成した後、データセットにデータを格納する。

- リクエストヘッダの作成

送信時に **HTTP** リクエストのヘッダに情報を格納するために、**IDictionary** ジェネリックインターフェイス実装クラスのインスタンスを利用する。ここでは、サーバ処理を一意に識別するための文字列であるリクエスト名を格納する必要がある。リクエスト名を格納する際のキーとして **requestName** を指定する。

- 実行とエラー処理

通信を実行するには、**DataSetXmlCommunicator** クラスのインスタンスに対して、**Communicate** メソッドを呼び出す。このとき、引数に送信用データセットと、リクエストヘッダ情報を格納したディクショナリを渡す。

通信が失敗した場合や、サーバでエラーが発生してエラー電文が返却された場合、**DataSetXmlCommunicator** は例外をスローする。以下にスローされる例外の種類を示す。

表 4 例外の種類

項番	例外クラス	発生原因
1	ServerException	サーバ処理において例外が発生し、エラー電文が返却された場合。
2	CommunicationException	サーバが見つからないなど、通信に失敗した場合。

ServerException のインスタンスには、エラー種別を示す文字列(**ErrorType** プロパティ)と、エラー情報のリスト(**Errors** プロパティ)が格納される。以下にエラー種別とその内容について示す。

表 5 エラー種別とエラーの内容

項番	ErrorType の値	説明	備考
1	"serverException"	サーバで例外が発生したことを示す。	
2	"serverValidationException"	サーバで入力値検証エラーが発生したことを示す。	Errors プロパティは、 ValidationMessageInfo クラスのインスタンスのリストとなる。
3	(その他の文字列)	その他、サーバでクライアントにエラーを通知したいときに、任意の文字列をサーバで格納する。	

- 受信用データセットの取得

サーバ処理が正常に終了した際の結果を取得する場合は、`Communicate` メソッドの戻り値である `CommunicationResult` クラスのインスタンスに格納されている受信用データセットを取得する。

以下に、`CommunicationResult` クラスのプロパティ一覧を示す。

表 6 `CommunicationResult` クラスのプロパティ一覧

項番	プロパティ名	説明
1	<code>ResponseHeaders</code>	HTTP レスポンスヘッダが格納される。
2	<code>ResultData</code>	サーバ処理の結果のデータセットが格納される。

(2) リクエストごとに通信先URLを変更する場合

リクエストごとに通信先URLを変更する場合は、`DataSetXmlCommunicator` クラスのインスタンスの `Address` プロパティにURLを設定する。実装例をリスト 10に示す。

```
communicator.Address = "http://terasoluna.local/index.aspx";
```

リスト 10 リクエストごとに通信先 URL を変更する場合

なお、両方指定していた場合は、`Address` プロパティに指定した方が優先される。

(3) リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更する場合は、`DataSetXmlCommunicator` クラスのインスタンスの `RequestTimeout` プロパティにリクエストタイムアウト時間を設定する。実装例をリスト 11に示す。

```
// リクエストタイムアウト時間を5秒に設定  
communicator.RequestTimeout = 5000;
```

リスト 11 リクエストごとにリクエストタイムアウト時間を変更する場合

なお、両方指定していた場合は、`RequestTimeout` プロパティに指定した方が優先される。

(4) 通信のキャンセル

通信中に送受信を中止したい場合は、`Cancel` メソッドを呼び出すことで処理をキャンセルすることができる。

```
communicator.Cancel();
```

リスト 12 通信のキャンセル

ただし、`Cancel` メソッドを呼び出すのは、`Communicate` メソッドを呼び出して通信処理を実行したスレッドとは別のスレッドでなければならない。

(5) 進捗状況イベント

通信中に別の処理を実行したい場合は、`ProgressChanged` イベントを利用する。

ProgressChanged イベントは、固定バイト数ごとに送受信が処理されると通知されるイベントである(既定では 8192 バイト)。

なお、進捗状況を示す値は **ProgressChanged** イベントのイベントハンドラの引数として渡される **ExecuteProgressChangedEventArgs** クラスのインスタンスに格納されている。進捗状況を表す値は、百分率で表された整数値で、送信終了で 50、受信終了で 100 となる。

```
private void communicator_ProgressChanged(object sender,
                                           ExecuteProgressChangedEventArgs e)
{
    int percentage = e.ProgressPercentage;

    // 進捗状況を示す百分率を画面に表示するなど
    //...
}
```

リスト 13 進捗状況イベントのイベントハンドラ

■ 内部構成

以下では、TERASOLUNA が提供する通信関連機能 (XML 通信機能、ファイルアップロード機能、ファイルダウンロード機能) に関する内部構成について説明する。

◆ 全体像

通信関連機能を構成する主なクラスの関係の全体を以下に示す。

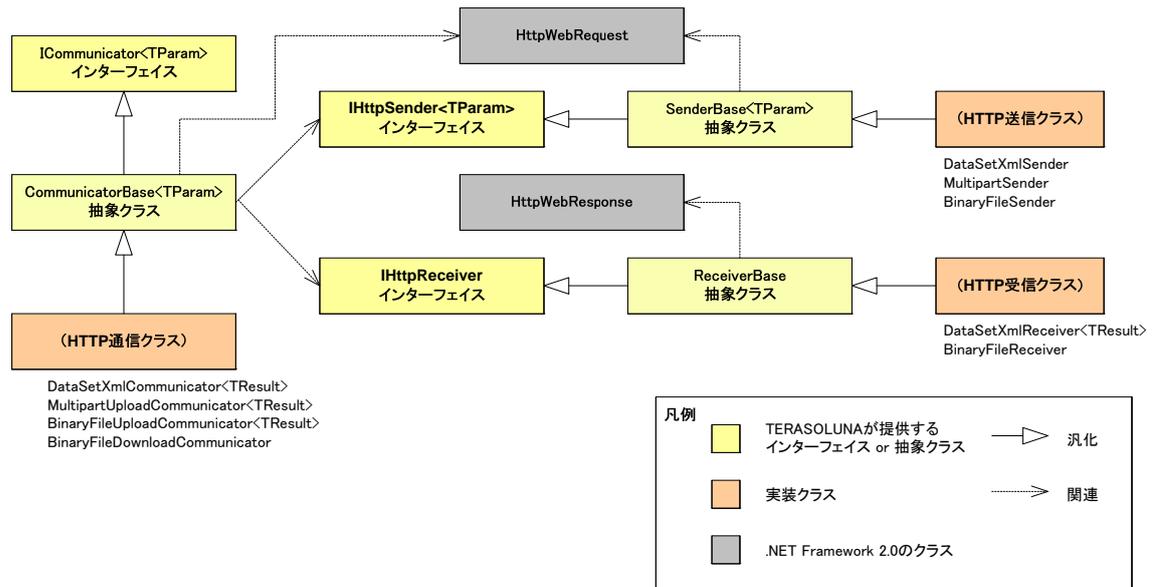


図 4 通信関連クラスの関係図

TERASOLUNA では、HTTP 送信クラスと HTTP 受信クラスを組み合わせることで、通信処理を実現している。

表 7 通信関連機能を構成するクラスの役割

項番	種類	説明
1	HTTP 通信クラス	通信処理を提供するためのクラス。通信処理を行う際にはこのクラスを利用する。内部では HTTP 送信クラスと HTTP 受信クラスを組み合わせ、通信処理を実現している。なお、HttpWebRequest クラスのインスタンスはこのクラスで作成される。
2	HTTP 送信クラス	HttpWebRequest クラスのインスタンスを利用して、データを送信するクラス。通常直接利用することはない、HTTP 通信クラスによって呼び出される。
3	HTTP 受信クラス	HttpWebRequest クラスのインスタンスから HttpWebResponse クラスのインスタンスを取得し、データを受信するクラス。通常直接利用することはない、HTTP 通信クラスによって呼び出される。

各通信機能は、その目的に応じた HTTP 通信クラスを提供している。以下に各通信関連機能で提供する HTTP 通信クラスとそれを構成する主なクラスの関係を示す。

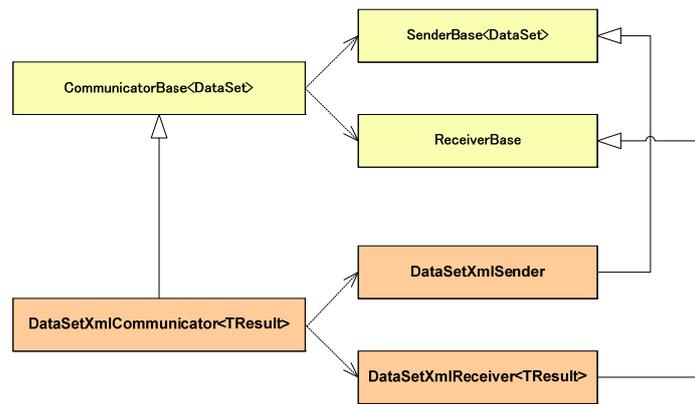


図 5 XML 通信機能における主なクラスの関係図

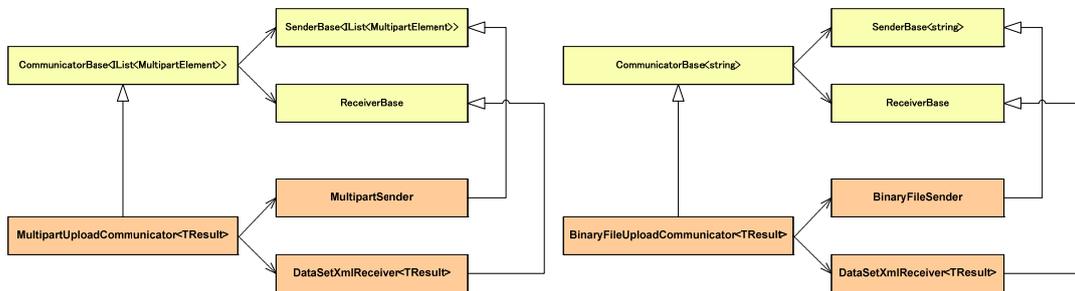


図 6 ファイルアップロード機能における主なクラスの関係図

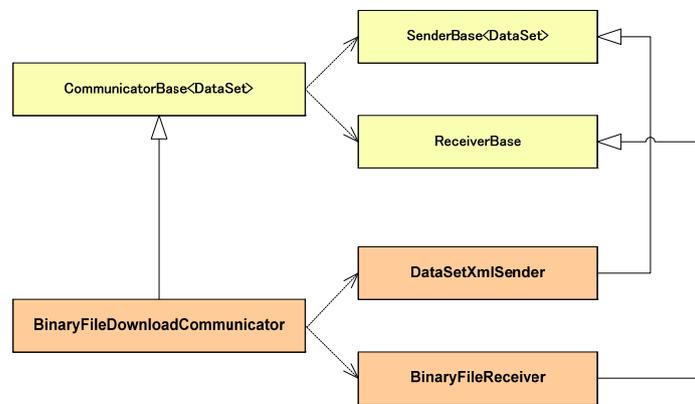


図 7 ファイルダウンロード機能における主なクラスの関係図

◆ Communicateメソッドの処理フロー

Communicate メソッドは、通信処理を実行する主要なメソッドである。以下に処理フローの概念図を示す。

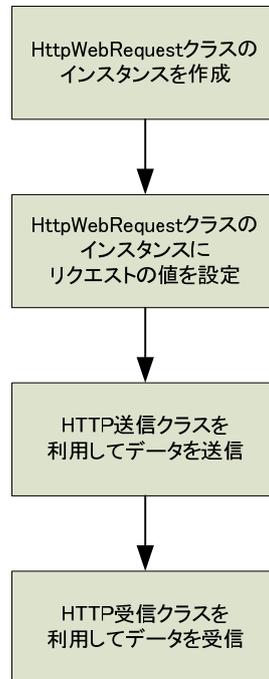


図 8 Communicate メソッドの処理フロー

◆ リクエストに対する設定

通信関連機能において、通信には `HttpWebRequest` クラスのインスタンスを利用している。通信関連機能では、このインスタンスに対して既定でプロパティを設定している。以下に、HTTP 受信クラスで設定しているプロパティとその値を示す。

表 8 HttpWebRequest のプロパティ設定値

項番	プロパティ名	説明	設定値
1	Method	インターネットリソースと通信するために使用する要求メソッド。	POST (固定値)
2	AllowAutoRedirect	要求がリダイレクト応答に従うかどうかを示す値。	false (固定値)
3	ContentType	Content-Type HTTP ヘッダの値。	送信する形式に応じた値。 XML 通信機能/ファイルダウンロード機能は"text/xml"。
4	ContentLength	インターネット リソースに送信するデータのバイト数。	送信データのバイト数。データの大きさに応じて設定する。

ContentType プロパティに関しては、各通信機能によって値が異なる。以下にその一覧を示す。

表 9 ContentType の設定値

項番	機能	値
1	XML 通信機能 ファイルダウンロード機能	text/xml
2	ファイルアップロード機能 (マルチパートアップロード)	multipart/form-data; boundary={0}; charset={1} {0} : バウンダリ文字列 {1} : 文字セット ※詳細は「FC-02 ファイルアップロード機能」参照
3	ファイルアップロード機能 (バイナリアップロード)	application/octet-stream

◆ エラー電文

サーバで例外が発生した場合、サーバ AP は TERASOLUNA では既定のエラー電文形式で XML データをクライアントに返却する。

サーバで例外が発生した場合と、入力値検証エラーが発生した場合について、返却される XML データの例を以下に示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- ルートノードなど、XMLの構造は規定しない -->
<root>
  <!-- エラー時に読み込まれる形式 -->
  <errors>
    <error>
      <error-code>EE99</error-code>
      <error-message>ユーザー認証に失敗しました。</error-message>
    </error>
    <error>
      <!-- エラーの数だけ繰り返す -->
    </error>
  </errors>
</root>
```

リスト 14 サーバエラー時の電文形式の例

```

<?xml version="1.0" encoding="utf-8" ?>
<!-- ルートノードなど、XMLの構造は規定しない -->
<root>
  <!-- 入力値検証エラー時に読み込まれる形式 -->
  <errors>
    <error>
      <error-code>0E01</error-code>
      <error-message>ユーザー名が不正です。</error-message>
      <error-field>userName</error-field>
    </error>
    <error>
      <!-- 入力値検証エラーの数だけ繰り返す -->
    </error>
  </errors>
</root>

```

リスト 15 サーバ入力値検証エラー時の電文形式の例

リスト 14、リスト 15で示したエラー電文はTERASOLUNAで既定したエラー電文形式である。通信機能において、エラー電文を受信すると、受信したXMLデータからerror要素を取得し、その内容をMessageInfoまたはValidationMessageInfo格納して、例外をスローする。

MessageInfo と ValidationMessageInfo の主なプロパティと、対応する XML 要素の一覧を以下に示す。

表 10 MessageInfo のプロパティと XML 要素の対応

項番	プロパティ名	説明	対応する XML 要素
1	Key	サーバで発生したエラーコード文字列	error 要素 / error-code 要素
2	Message	サーバのエラーメッセージ文字列	error 要素 / error-message 要素
3	Detail	エラー情報を表す XmlNode オブジェクト	error 要素を読み込んだ XmlNode

表 11 ValidationMessageInfo のプロパティと XML 要素の対応

項番	プロパティ名	説明	対応する XML 要素
1	Key	サーバで発生したエラーコード文字列	error 要素 / error-code 要素
2	Message	サーバのエラーメッセージ文字列	error 要素 / error-message 要素
3	ErrorPath	エラー発生箇所を表す XPath	error 要素 / error-field 要素
4	Detail	エラー情報を表す XmlNode オブジェクト	error 要素を読み込んだ XmlNode

■ 拡張ポイント

◆ 通信用ビジネスロジックのカスタマイズ

業務に応じて、カスタマイズした通信用ビジネスロジックを作成する場合は、`CommunicateBLogicBase<TParam>`クラスを継承して作成する。

`CommunicateBLogicBase<TParam>`クラスは、利用する HTTP 通信クラスを作成する `CreateCommunicator` メソッドと、通信処理を実行する抽象メソッド `Communicate` を定義している。派生クラスでカスタマイズするには、`Communicate` メソッドに通信処理の呼び出しを実装し、必要に応じて `CreateCommunicate` メソッドをオーバーライドする。

```
protected abstract CommunicationResult Communicate(BLogicParam blogicParam,  
                                                    IDictionary<string, string> requestHeaders);
```

リスト 16 `Communicate` 抽象メソッドの定義

◆ HTTP通信クラスのカスタマイズ

HTTP 通信クラスをカスタマイズする場合、標準で提供する HTTP 通信クラスを継承して拡張するか、`CommunicatorBase<TParam>`クラスなどを継承して作成する。

以下に、HTTP 通信クラスをカスタマイズした例を示す。

```
/// <summary>
/// リクエストではマルチパート形式でファイルをアップロードし、
/// レスポンスはファイルをダウンロードするHTTP通信クラス
/// </summary>
public class MyCommunicator : CommunicatorBase<IList<MultipartElement>>
{
    public MyCommunicator ()
    {
        this.Sender = new MultipartSender ();
        this.Receiver = new BinaryFileReceiver ();
    }

    public virtual DownloadResult Download(IList<MultipartElement> paramData,
                                           IDictionary<string, string> requestHeaders)
    {
        CommunicationResult cResult = Communicate(paramData, requestHeaders);

        DownloadResult dResult = cResult as DownloadResult;
        if (dResult == null)
        {
            dResult = new DownloadResult(null);
            dResult.AddResponseHeaders(cResult.ResponseHeaders);
            dResult.ResultData = cResult.ResultData;
        }

        return dResult;
    }
}
```

リスト 17 HTTP 通信クラスをカスタマイズした例

また、必要に応じて、カスタマイズした HTTP 通信クラスを利用する通信用ビジネスロジックを作成する。

以下に、リスト 17で定義したMyCommunicatorを利用したビジネスロジックの例を示す。

```
public class MyCommunicateBLogic : MultipartUploadBLogic<DataSet>
{
    public MyCommunicateBLogic()
    {
    }

    protected override ICommunicator<IList<MultipartElement>> CreateCommunicator ()
    {
        // 利用するMyCommunicator の生成と初期化処理
        return new MyCommunicator ();
    }

    // 通信処理後、ダウンロードしたファイルの保存先を設定する。
    protected override void AfterCommunicate (
        CommunicationResult communicationResult, BLogicResult blogicResult)
    {
        base.AfterCommunicate(communicationResult, blogicResult);

        DownloadResult downloadResult = communicationResult as DownloadResult;
        if (string.IsNullOrEmpty(downloadResult.DownloadFilePath))
        {
            return;
        }

        blogicResult.Items[BinaryFileDownloadBLogic.DOWNLOAD_FILEPATH] =
            downloadResult.DownloadFilePath;
    }
}
```

リスト 18 カスタマイズした HTTP 通信クラスを利用するビジネスロジックの例

◆ HTTP送受信クラスのカスタマイズ

HTTP 送受信クラスのカスタマイズする場合、SenderBase<TParam>、ReceiverBase クラスを継承する。また、カスタマイズした HTTP 送受信クラスを利用する HTTP 通信クラスと通信用ビジネスロジックをそれぞれ作成する。

なお、HTTP 送受信クラスには進捗状況を通知するための IProgressChangeReporter の実装クラスが渡される。通信の進捗状況に応じて、ReportProgressChanged メソッドを呼び出して進捗状況を通知する。

以下に、SenderBase、ReceiverBase を利用した送信・受信処理について簡単な実装例を示す。

```
public class MyDataSetXmlSender : SenderBase<DataSet>
{
    public override string ContentType
    {
        get { return "text/xml;charset=utf-8"; }
    }

    protected override void SendRequest(HttpWebRequest request, DataSet paramData,
        IDictionary<string, string> headerCollection,
        IProgressChangeReporter reporter)
    {
        if (reporter != null)
        {
            reporter.ReportProgressChanged(new ExecuteProgressChangedEventArgs(0));
        }
        using (Stream reqStream = request.GetRequestStream())
        {
            paramData.WriteXml(reqStream, XmlWriteMode.IgnoreSchema);
        }
        if (reporter != null)
        {
            reporter.ReportProgressChanged(new ExecuteProgressChangedEventArgs(50));
        }
    }
}
```

ContentType プロパティが返す値は
リクエストの Content-Type ヘッダに設定される

SendRequest メソッドは PrepareRequest メソッドの後に呼び出
される。リクエストの初期化処理を変更する場合は
PrepareRequest メソッドをオーバーライドする

ここではリクエストストリームに一括で XML データを書き込んで
いる

リスト 19 DataSet を引数に取る HTTP 送信クラスの実装例

```
public class MyDataSetXmlReceiver<TResult> : ReceiverBase
    where TResult : DataSet, new()
{
    protected override CommunicationResult ReceiveResponse(HttpWebResponse response,
        IProgressChangeReporter reporter)
    {
        TResult resultDataSet = new TResult();
        if (reporter != null)
        {
            reporter.ReportProgressChanged(new ExecuteProgressChangedEventArgs(51));
        }
        using (Stream resStream = response.GetResponseStream())
        {
            resultDataSet.ReadXml(resStream, XmlReadMode.IgnoreSchema);
        }
        if (reporter != null)
        {
            reporter.ReportProgressChanged(new ExecuteProgressChangedEventArgs(100));
        }
        // サーバで業務エラーが発生していれば、ServerExceptionとする
        string errorType = response.Headers[SERVER_ERROR_TYPE];
        if (errorType != null)
        {
            ServerException exception = new ServerException();
            exception.ErrorType = ReceiverBase.SERVER_EXCEPTION;
            throw exception;
        }
        // 結果オブジェクトの組み立て
        CommunicationResult result = new CommunicationResult();
        result.ResultData = resultDataSet;
        foreach (string key in response.Headers.Keys)
        {
            result.ResultHeaders.Add(key, response.Headers[key]);
        }
        return result;
    }
}
```

ReceiveResponse メソッドは Receive メソッドから呼び出される

ここでは型パラメータで与えられた型のデータセットに読み込んでいる

リスト 20 XML データを読み込むデータセットを指定した HTTP 受信クラスの実装例

リスト 19、リスト 20で示した例では、データセットを入出力に用いたXMLデータのHTTP送受信クラスを実装している。また、SendRequest、ReceiveResponseメソッドに渡されるreporterインスタンスを利用して送受信の開始時と終了時に進捗状況を通知している。

■ 関連機能

- ◆ FA-03: 拡張フォーム機能
- ◆ FB-01: イベント処理機能
- ◆ FB-02: データセット変換機能
- ◆ FC-02: ファイルアップロード機能
- ◆ FC-03: ファイルダウンロード機能

FC-02 ファイルアップロード機能

■ 概要

本機能は、HTTP 通信を行い、TERASOLUNA Server Framework for Java/.NET で開発されたサーバ AP と連携してマルチパート形式またはバイナリ形式でファイルアップロードを行うための仕組みを提供する。

本機能の通信では、レスポンスは XML 形式のみである。

本機能を利用することで、イベントコントローラ経由、またはファイルアップロード用通信クラスを直接利用して、ファイルアップロードを実行できる。

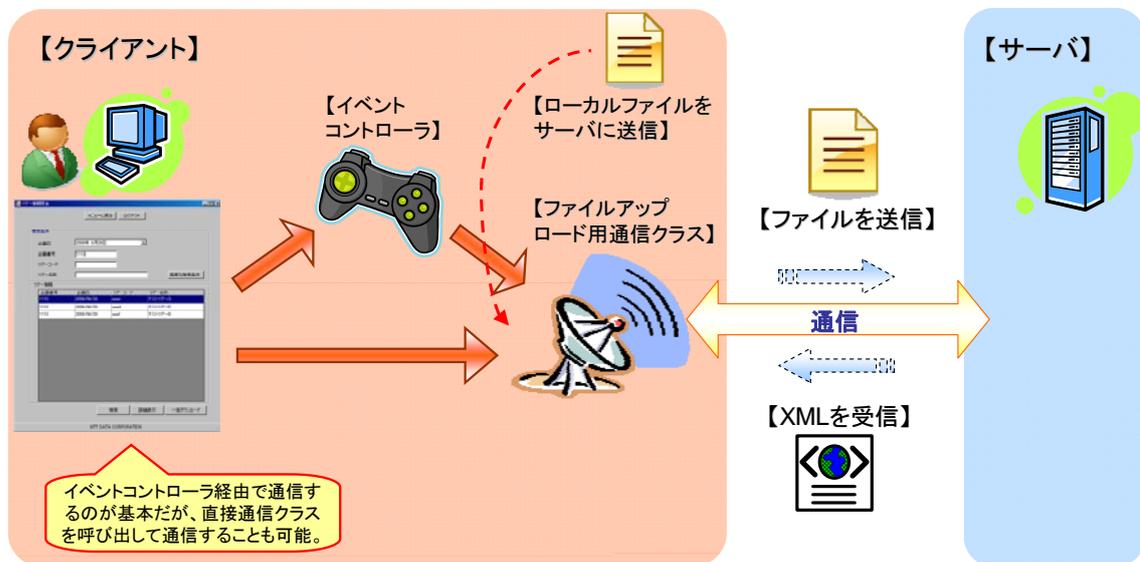


図 1 概念図

本資料では、イベントコントローラを使用するパターンと、直接通信クラスを使用するパターンの2種類に分けて説明を行う。

■ 使用方法(イベントコントローラ経由で、通信を行う場合)

ファイルアップロードを行う場合、各形式に対応したビジネスロジックを利用することで、イベントコントローラ経由でファイルアップロードを実現することができる。

以下に、形式に対応するビジネスロジックを示す。

表 1 ファイルアップロードを実現するビジネスロジック

項番	ファイルアップロード形式	ビジネスロジッククラス
1	マルチパート形式	MultipartUploadBLogic
2	バイナリ形式	BinaryFileUploadBLogic

◆ アプリケーション構成ファイル

(1) 接続先URLの設定

アプリケーション構成ファイル(App.config)に接続先 URL を設定する。接続先 URL は、アプリケーション構成ファイルの `appSettings/add` 要素に、`key` 属性を”BaseUrl”として、`value` 属性に通信先 URL を設定する。

以下に、接続先 URL を設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="BaseUrl" value="http://terasoluna.local/index.aspx"/>
  </appSettings>
</configuration>
```

リスト 1 アプリケーション構成ファイルに接続先 URL を設定する例

(2) リクエストタイムアウト時間の設定

リクエストタイムアウト時間を設定するには、アプリケーション構成ファイルの `appSettings/add` 要素に、`key` 属性を”RequestTimeout”として、`value` 属性にタイムアウト時間(ミリ秒)を設定する。

アプリケーション構成ファイルに設定したタイムアウト時間は、全リクエストに対して適用される。リクエストごとにタイムアウト時間を変更する場合の方法は、後述の「リスト 7 リクエストタイムアウト時間を変更する場合の実装例」および「リスト 13 リクエストごとにリクエストタイムアウト時間を変更する場合」を参照すること。

以下に、リクエストタイムアウト時間をアプリケーション構成ファイルに設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- 標準のリクエストタイムアウト時間を1時間(3600秒)とする -->
    <add key="RequestTimeout" value="3600000"/>
  </appSettings>
</configuration>
```

リスト 2 アプリケーション構成ファイルにリクエストタイムアウト時間を設定する例

(3) ビジネスロジック設定ファイルとデータセット変換設定ファイルのパスの設定

ファイルアップロードビジネスロジックを利用するには、利用するビジネスロジックをビジネスロジック設定ファイルに記述し、さらにビジネスロジック入力データセットへの変換設定をデータセット変換ファイルに記述する必要がある。

記述する各設定ファイルのパスは、アプリケーション構成ファイルで指定する。以下に、ビジネスロジック設定ファイルのパスとデータセット変換設定ファイルのパスを指定する例を示す。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
      type="TERASOLUNA.Fw.Common.Configuration.BLogic.
        BLogicConfigurationSection, TERASOLUNA.Fw.Common"/>
    <section name="conversionConfiguration"
      type="TERASOLUNA.Fw.Client.Configuration.Conversion.
        ConversionConfigurationSection, TERASOLUNA.Fw.Client"/>
  </configSections>

  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="Config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>

  <conversionConfiguration>
    <files>
      <!-- データセット変換定義ファイルのパス設定 -->
      <file path="Config¥ConverterConfiguration.config"/>
    </files>
  </conversionConfiguration>
</configuration>
```

リスト 3 アプリケーション構成ファイルにビジネスロジック設定ファイルのパスとデータセット変換設定ファイルのパスを記述する例

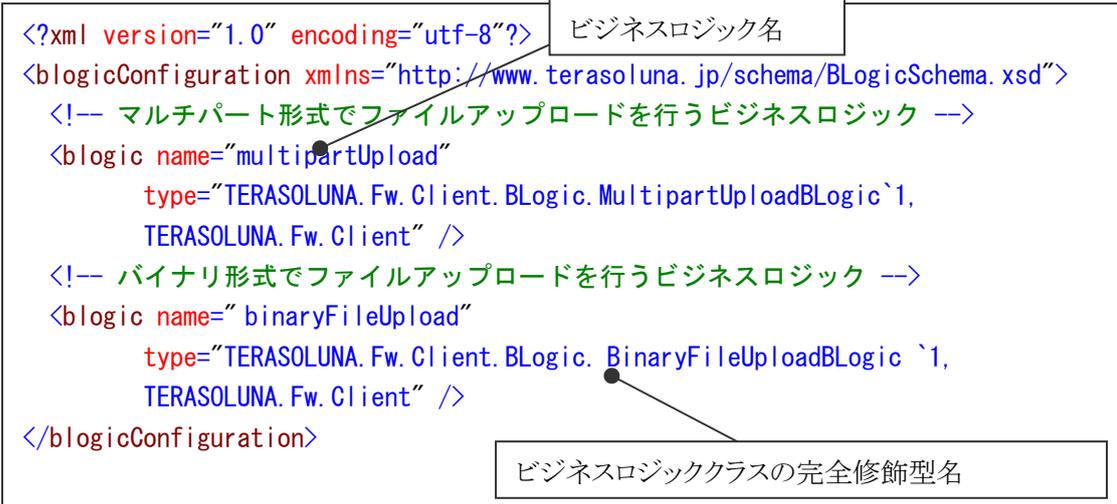
ビジネスロジック設定ファイルの詳細については『CM-04 ビジネスロジック生成機能』を、データセット変換設定ファイルの詳細については『FB-02 データセット変換機能』を参照すること。

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルにファイルアップロードビジネスロジックのビジネスロジック名とクラスの完全修飾型名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。

以下に、ビジネスロジック設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <!-- マルチパート形式でファイルアップロードを行うビジネスロジック -->
  <blogic name="multipartUpload"
    type="TERASOLUNA.Fw.Client.BLogic.MultipartUploadBLogic`1,
    TERASOLUNA.Fw.Client" />
  <!-- バイナリ形式でファイルアップロードを行うビジネスロジック -->
  <blogic name="binaryFileUpload"
    type="TERASOLUNA.Fw.Client.BLogic.BinaryFileUploadBLogic `1,
    TERASOLUNA.Fw.Client" />
</blogicConfiguration>
```



リスト 4 ビジネスロジック設定ファイルの記述例

ビジネスロジック設定ファイルの詳細に関しては『CM-04 ビジネスロジック実行機能』を参照すること。

◆ データセット変換設定ファイル

データセット変換設定ファイルに画面データセットからビジネスロジック入力データセットへの変換設定を記述する。

以下に、データセット変換設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<conversionConfiguration
xmlns="http://www.terasoluna.jp/schema/ConversionSchema.xsd">
  <convert id="multipartUploadConvert">
    <param>
      <column src="File.Name" />
      <column src="File.FilePath" />
      <column src="Text.Name" />
      <column src="Text.Value" />
    </param>
    <result>
      . . .省略. . .
    </result>
  </convert>
</conversionConfiguration>
```

リスト 5 データセット変換設定ファイルの記述例

データセット変換設定ファイルの詳細については『FB-02 データセット変換機能』を参照すること。

◆ 実装方法(マルチパートアップロード)

マルチパートアップロード処理を実装するには、(1)画面データセットの作成、(2)ビジネスロジック入出力データセットの作成、(3)イベントコントローラのプロパティの設定、(4)進捗状況通知イベントハンドラの実装、(5)イベントコントローラの実行の順番で実装する。

(1) ビジネスロジック入力データセットの作成

画面の情報を保持するデータセットを作成する。

マルチパートアップロードビジネスロジックを利用する場合、アップロードするファイルデータとテキストデータを指定するためのマルチパートアップロード用データセットを利用する。マルチパートアップロード用データセットには、**File** テーブルと **Text** テーブルを作成する。

File テーブルはアップロードするファイルの情報を格納するために利用し、**Text** テーブルはアップロードするテキストデータを格納するために利用する。

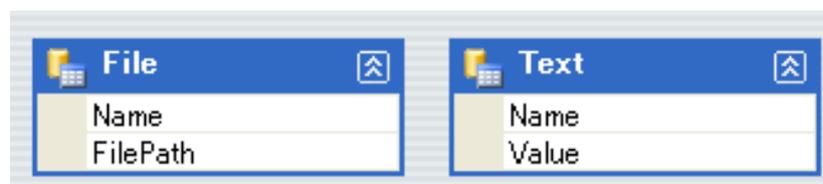


図 2 マルチパートアップロード用データセットに必要なテーブルとカラム

表 2 マルチパートアップロード用データセットに必要なテーブルとカラム一覧

項番	テーブル名	カラム名	DataType	説明
1	File	Name	System.String	重複しない任意のマルチパートの要素名を設定する。
2		FilePath	System.String	アップロードするファイルの絶対パス、またはアプリケーションルートからの相対パスを設定する。
3	Text	Name	System.String	重複しない任意のマルチパートの要素名を設定する。
4		Value	System.String	アップロードするテキストを設定する。

作成したマルチパートアップロード用データセットを、マルチパートアップロードを行う画面に追加する。

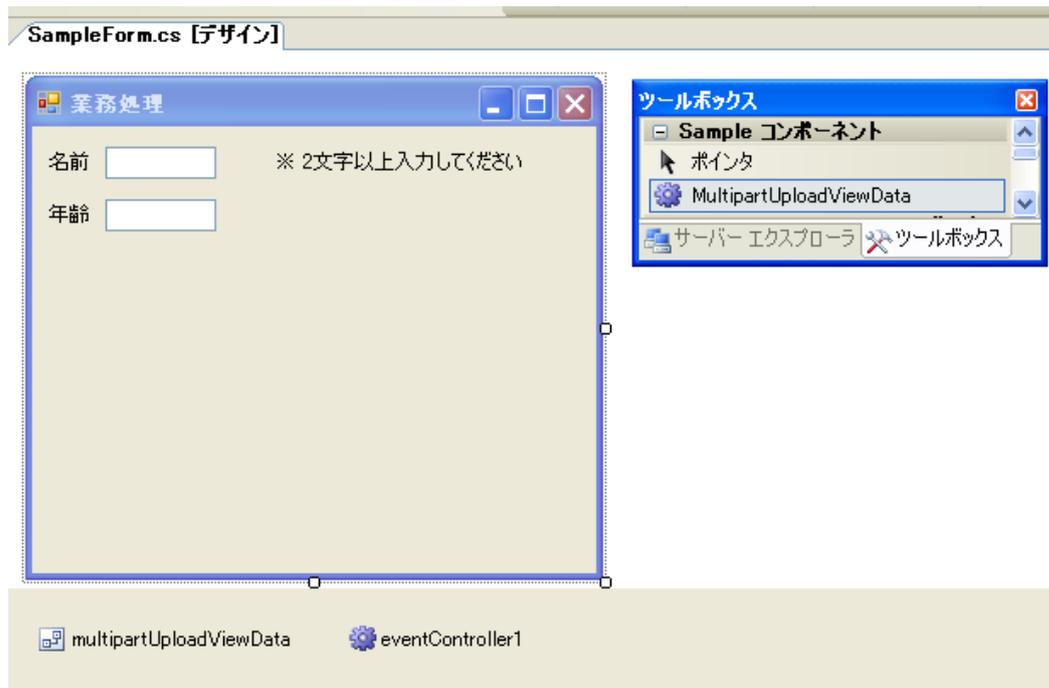


図 3 マルチパートアップロード用データセットを画面に追加する

(2) ビジネスロジック入出力データセットの作成

XML 電文としてサーバに送受信するデータを格納するためのデータセットを作成する。なお、画面データセットをそのまま利用することも出来る。その際は、後述のイベントコントローラのプロパティの設定において、“BLogicParamTypeName”、“BlogicResultTypeName”を空文字列にする。

(3) イベントコントローラのプロパティの設定

イベントコントローラのプロパティに、マルチパートアップロードビジネスロジックの実行に必要な設定を行う。設定しなければならないプロパティを以下に示す。

表 3 設定しなければならないイベントコントローラのプロパティ

項番	プロパティ	説明
1	BLogicName	ビジネスロジック設定ファイルに記述したマルチパートアップロードビジネスロジックのビジネスロジック名。
2	BLogicParamTypeName	ビジネスロジック入力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
3	BLogicResultTypeName	ビジネスロジック出力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
4	ConvertId	データセット変換設定ファイルに記述したコンバートID。
5	RequestName	サーバ側でマルチパートアップロード用のリクエストを受け付けるためのリクエスト名。
6	ViewData	画面データセットのインスタンス。

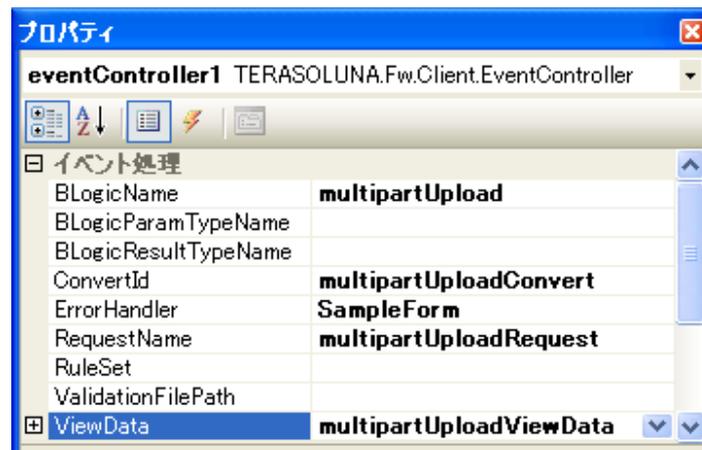


図 4 イベントコントローラのプロパティ設定例

(4) イベントコントローラの実行

イベントコントローラを実行し、実行結果の確認を行う実装例を以下に示す。設定内容は「表 1 マルチパートアップロード用データセットに必要なテーブルとカラム一覧」を参照のこと。実装例をリスト 6 に示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // ファイル情報の設定
    multipartUploadViewData.File.AddFileRow("File01", "マルチパートアップロード.bmp");
    // テキスト情報の設定
    multipartUploadViewData.Text.AddTextRow("Text01", "マルチパートテキスト");

    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();
    if (result.Success)
    {
        // ビジネスロジック成功に対する処理
    }
    else if ("communicationException".Equals(result.ResultString))
    {
        // 通信エラーに対するエラー処理
    }
}
```

リスト 6 マルチパートアップロードの実装例

イベントコントローラの実行後、**ExecutionResult** の **ResultString** プロパティにイベント処理が成功したかどうかを示す文字列が格納されるので、この値を確認する。失敗した場合は、エラー内容に応じた文字列(イベント処理を実行した結果)が格納されているので、適切にエラーハンドリング処理を実装する。

以下に、**ResultString** プロパティの値の一覧を示す。

表 4 ExecutionResult の ResultString 値一覧

項番	状態	値
1	単項目チェックエラー	“validationError”
2	カスタムチェックエラー	“validationError”
3	前処理エラー	“preprocessedError” または前処理イベントで設定する ResultString の値
4	ビジネスロジック実行失敗 (通信中に例外発生)	"communicationException"
5	ビジネスロジック実行失敗 (サーバ側で入力値検証エラー発生)	"serverValidateException"
6	ビジネスロジック実行成功	“success”
7	ビジネスロジック実行失敗 (サーバ側で業務エラー発生)	サーバでレスポンスヘッダに設定された任意の文字列 (例: "serviceException")

(5) その他

その他、マルチパートアップロード処理における設定を変更する方法を示す。

● リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更したい場合、イベントコントローラの Items プロパティに“RequestTimeout”をキーとして、リクエストタイムアウト時間を設定する。

以下に、実装例を示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // 省略

    // Itemsにリクエストタイムアウト時間 (5000ms)を設定する
    eventController1.Items["RequestTimeout"] = 5000;
    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();
}
```

リスト 7 リクエストタイムアウト時間を変更する場合の実装例

なお、Items プロパティとプリケーション構成ファイルの両方にリクエストタイムアウト時間を設定した場合、Items プロパティに設定した値が利用される。

● リクエストヘッダの設定

通信時のリクエストヘッダに値を格納する場合は、IDictionary ジェネリックインターフェイス実装クラスのインスタンスを利用して、イベントコントローラの Items プロパティに設定する。

なお、リクエスト名はイベントコントローラのプロパティに設定すると自動的にリクエストヘッダに格納されて送信される。

```
Dictionary<string, object> requestHeaders  
    = new Dictionary<string, object>();  
  
requestHeaders.Add("customData", "foo");
```

リスト 8 リクエストヘッダの設定

- 通信のキャンセル

通信中に送受信を中止したい場合は、イベントコントローラの **Cancel** メソッドを呼び出すことで処理をキャンセルすることができる。キャンセルは非同期処理実行時のみ行うことができる。詳細は、『FB-01 イベント処理機能』を参照すること。

◆ 実装方法(バイナリアップロード)

バイナリアップロード処理を実装するには、(1)画面データセットの作成、(2)ビジネスロジック出力データセットの作成、(3)イベントコントローラのプロパティの設定、(4)進捗状況通知イベントハンドラの実装、(5)イベントコントローラの実行の順番で実装する。

(1) 画面データセットの作成

画面の情報を保持するデータセットを作成する。

バイナリアップロードビジネスロジックを利用する場合、アップロードするファイルのパスを指定するためのバイナリアップロード用データセットを利用する。バイナリアップロード用データセットには、File テーブルを作成する。File テーブルはアップロードするファイルのパスを格納する。

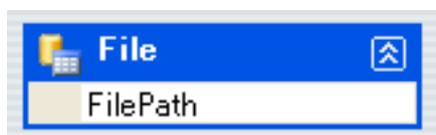


図 5 バイナリアップロード用データセットに必要なテーブルとカラム

表 5 バイナリアップロード用データセットに必要なテーブルとカラム一覧

項番	テーブル名	カラム名	Data Type	説明
1	File	FilePath	System.String	アップロードするファイルの絶対パス、またはアプリケーションルートからの相対パスを設定する。

作成したバイナリアップロード用データセットを、バイナリアップロードを行う画面に追加する。

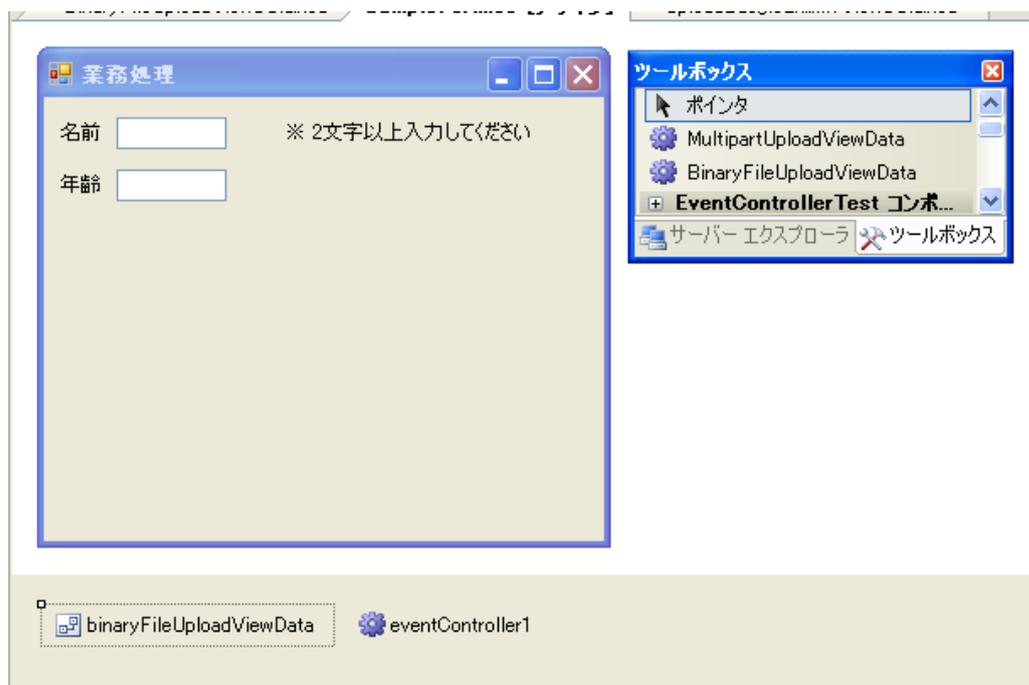


図 6 バイナリアップロード用データセットを画面に追加する

(2) ビジネスロジック入出力データセットの作成

XML 電文としてサーバに送受信するデータを格納するためのデータセットを作成する。なお、画面データセットをそのまま利用することも出来る。その際は、後述のイベントコントローラのプロパティの設定において、“BLogicParamTypeName”、“BlogicResultTypeName”を空文字列にする。

(3) イベントコントローラのプロパティの設定

イベントコントローラのプロパティに、バイナリアップロードビジネスロジックの実行に必要な設定を行う。設定しなければならないプロパティを以下に示す。

表 6 設定しなければならないイベントコントローラのプロパティ

項番	プロパティ	説明
1	BLogicName	ビジネスロジック設定ファイルに記述したバイナリアップロードビジネスロジックのビジネスロジック名。
2	BLogicResultTypeName	ビジネスロジック出力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
3	ConvertId	データセット変換設定ファイルに記述したコンバートID。
4	RequestName	サーバ側でバイナリアップロード用のリクエストを受け付けるためのリクエスト名。
5	ViewData	画面データセットのインスタンス。

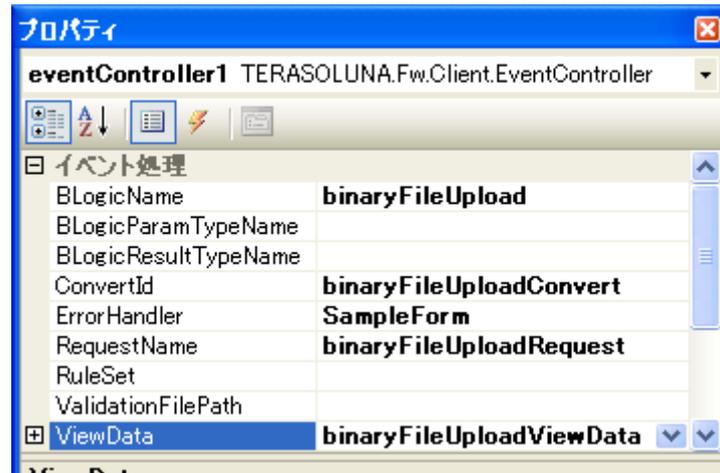


図 7 イベントコントローラのプロパティ設定例

(4) イベントコントローラの実行

イベントコントローラを実行し、実行結果の確認を行う実装例を以下に示す。設定内容は「表 5 設定しなければならないイベントコントローラのプロパティ」を参照のこと。実装例をリスト 6 に示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // ファイルパスの設定
    binaryFileUploadViewData.File.AddFileRow("バイナリアップロード.bmp");

    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();
    if (result.Success)
    {
        // ビジネスロジック成功に対する処理
    }
    else if ("communicationException".Equals(result.ResultString))
    {
        // 通信エラーに対するエラー処理
    }
}
```

リスト 9 バイナリアップロードの実装例

イベントコントローラの実行後、**ExecutionResult** の **ResultString** プロパティにイベント処理が成功したかどうかを示す文字列が格納されるので、この値を確認する。失敗した場合は、エラー内容に応じた文字列(イベント処理を実行した結果)が格納されているので、適切にエラーハンドリング処理を実装する。

以下に、**ResultString** プロパティの値の一覧を示す。

表 7 ExecutionResult の ResultString 値一覧

項番	状態	値
1	単項目チェックエラー	"validationError"
2	カスタムチェックエラー	"validationError"
3	前処理エラー	"preprocessedError" または前処理イベントで設定する ResultString の値
4	ビジネスロジック実行失敗 (通信中に例外発生)	"communicationException"
5	ビジネスロジック実行失敗 (サーバ側で入力値検証エラー発生)	"serverValidateException"
6	ビジネスロジック実行成功	"success"
7	ビジネスロジック実行失敗 (サーバ側で業務エラー発生)	サーバでレスポンスヘッダに設定された任意の文字列 (例："serviceException")

(5) その他

その他、バイナリアップロード処理における設定を変更する方法については、マルチパートアップロード処理と共通なので、「実装方法(マルチパートアップロード)(5)その他」を参照すること。

■ 使用方法(直接通信クラスを使用する場合)

マルチパートアップロードを行うには、`MultipartUploadCommunicator` クラスを直接利用して実現することも出来る。`MultipartUploadCommunicator` クラスは、イベントコントローラ経由で通信する場合に利用した `MultipartUploadBLogic` クラスの内部で利用している。

また、バイナリアップロードを行うには、`BinaryFileUploadCommunicator` クラスを直接利用して実現することも出来る。`BinaryFileUploadCommunicator` クラスは、イベントコントローラ経由で通信する場合に利用した `BinaryFileUploadBLogic` クラスの内部で利用している。

なお、`MultipartUploadCommunicator` クラス・`BinaryFileUploadCommunicator` クラスを直接利用して通信する場合は、例外処理などを自分で実装しなければならないことに注意すること。

◆ アプリケーション構成ファイル

本設定に関しては、「使用方法(イベントコントローラ経由で、通信を行う場合)」を参照すること。

◆ 実装方法

(1) マルチパートアップロード処理の実装

マルチパートアップロード処理を実装するには、①`MultipartUploadCommunicator` クラスのインスタンスの作成、②送信用パラメータの作成、③リクエストヘッダの作成、④進行状況通知イベントハンドラの登録、⑤実行とエラー処理の順番で実装する。

以下に、`MultipartUploadCommunicator` クラスで通信を実行する処理の実装例を示す。

```
// ①通信クラスのインスタンスを生成する
// Sample2Dsは受信するXMLを受けるデータセット
MultipartUploadCommunicator<Sample2Ds> communicator =
    new MultipartUploadCommunicator<Sample2Ds>();

// ②送信パラメータを作成
// マルチパートデータを設定する
MultipartValueElement textElement = new
MultipartValueElement("UploadText");
textElement.Value = "i was born";
MultipartFileElement fileElement = new MultipartFileElement("UploadFile");
fileElement.UploadFilePath = @"c:\temp\test.txt";

IList<MultipartElement> elementList = new List<MultipartElement>();
elementList.Add(textElement);
elementList.Add(fileElement);

// ③リクエストヘッダの作成(リクエスト名には"sampleRequest"を指定)
IDictionary<string, string> requestHeaders = new Dictionary<string, string>();
requestHeaders.Add("requestName", "sampleRequest");

// ※必要に応じてリクエストヘッダに値を設定
//...

// ④進行状況通知イベントハンドラを通信クラスへ登録する
communicator.ProgressChanged +=
    new ExecuteProgressChangedEventHandler(this.Sample_ExecuteProgressChanged);

// ⑤実行とエラー処理
try
{
    CommunicationResult result =
        communicator.Communicate(elementList, requestHeaders);
}
catch (ServerException e)
{
    // サーバで発生したエラーをハンドリングする必要がある場合、
    // ServerExceptionをキャッチする。
}
```

リスト 10 マルチパートアップロード処理の実行例

リスト 10では、MultipartElement型のインスタンスのリストを送信用パラメータとしてAPサーバに

送信し、リクエスト名”sampleRequest”で指定されたサーバ処理を実行している。

- **MultipartUploadCommunicator** クラスのインスタンスの作成

マルチパートアップロード処理を実装するには、**MultipartUploadCommunicator** クラスのインスタンスを利用する。最初に **MultipartUploadCommunicator** クラスのインスタンスを作成する。このとき、受信用データセットの型を型パラメータとして指定する。なお、指定できる型パラメータは **DataSet** クラスとその派生クラスのみである。

- 送信用パラメータの作成

マルチパートアップロード通信クラス(**MultipartUploadCommunicator**)を直接利用して通信する場合、**Communicate** メソッドの引数にマルチパート要素(**MultipartElement**)のコレクションを渡す。マルチパート要素にはファイル用マルチパート要素(**MultipartFileElement**)と、テキスト用マルチパート要素(**MultipartValueElement**)がある。

- リクエストヘッダの作成

送信時に **HTTP** リクエストのヘッダに情報を格納するために、**IDictionary** ジェネリックインターフェイス実装クラスのインスタンスを利用する。ここには、サーバ処理を一意に識別するための文字列であるリクエスト名を格納する必要がある。リクエスト名を格納する際のキーとして”requestName”を指定する。

- 進行状況通知イベントハンドラの登録

必要に応じて、進行状況通知イベントハンドラを登録する。このイベントを用いることで、プログレスバーなどに通信の進行状況を表示することができる。なお、進行状況を表す値は、百分率で表された整数値で、送信終了で 50、受信終了で 100 となる。実装に関しては、『**FB-01 イベント処理機能**』を参照すること。

- 実行とエラー処理

通信を実行するには、**MultipartUploadCommunicator** クラスのインスタンスに対して、**Communicate** メソッドを呼び出す。このとき、引数に送信用パラメータと、リクエストヘッダ情報を格納した **IDictionary** ジェネリックインターフェイス実装クラスのインスタンスを渡す。

通信が失敗した場合や、サーバでエラーが発生してエラー電文が返却された場合、**MultipartUploadCommunicator** は例外をスローする。以下にスローされる例外の種類を示す。

表 8 例外の種類

項番	例外クラス	発生原因
1	ServerException	サーバ処理において例外が発生し、エラー電文が返却された場合。
2	CommunicationException	サーバが見つからないなど、通信に失敗した場合。

ServerException のインスタンスには、エラー種別を示す文字列(**ErrorType** プロパティ)と、エラー情報のリスト(**Errors** プロパティ)が格納される。以下にエラー種別とその内容について示す。

表 9 エラー種別とエラーの内容

項番	ErrorType の値	説明	備考
1	“serverException”	サーバで例外が発生したことを示す。	
2	“serverValidationException”	サーバで入力値検証エラーが発生したことを示す。	Errors プロパティは、ValidationMessageInfo クラスのインスタンスのリストとなる。
3	(その他の文字列)	その他、サーバでクライアントにエラーを通知したいときに、任意の文字列をサーバで格納する。	

(2) バイナリアップロード処理の実装

バイナリアップロード処理を実装するには、①BinaryFileUploadCommunicator クラスのインスタンスの作成、②送信用パラメータの作成、③リクエスト ヘッダの作成、④進行状況通知イベントハンドラの登録、⑤実行とエラー処理の順番で実装する。

以下に、BinaryFileUploadCommunicator クラスで通信を実行する処理の実装例を示す。

```
// ①通信クラスのインスタンスを生成する
// Sample2Dsは受信するXMLを受けるデータセット
BinaryFileUploadCommunicator<Sample2Ds> communicator =
    new BinaryFileUploadCommunicator<Sample2Ds>();

// ②送信パラメータを作成
// ファイルパスを設定する
string filepath = "sampleFile.txt";

// ③リクエストヘッダの作成(リクエスト名には"sampleRequest"を指定)
IDictionary<string, string> requestHeaders = new Dictionary<string, string>();
requestHeaders.Add("requestName", "sampleRequest");

// ※必要に応じてリクエストヘッダに値を設定
//...

// ④進行状況通知イベントハンドラを通信クラスへ登録する
communicator.ProgressChanged +=
    new ExecuteProgressChangedEventHandler(this.Sample_ExecuteProgressChanged);

// ⑤実行とエラー処理
try
{
    CommunicationResult result =
        communicator.Communicate(filepath, requestHeaders);
}
catch (ServerException e)
{
    // サーバで発生したエラーをハンドリングする必要がある場合、
    // ServerExceptionをキャッチする。
}
```

リスト 11 バイナリアップロード処理の実行例

リスト 21 では、ファイルパスを送信用パラメータとして AP サーバに送信し、リクエスト名"sampleRequest"で指定されたサーバ処理を実行している。

- BinaryFileUploadCommunicator クラスのインスタンスの作成

バイナリアップロード処理を実装するには、BinaryFileUploadCommunicator クラスのインスタンスを利用する。最初に BinaryFileUploadCommunicator クラスのインスタンスを作成する。このとき、受信用データセットの型を型パラメータとして指定する。なお、指定できる型パラメータは DataSet クラスとその派生クラスのみである。

- 送信用パラメータの作成

バイナリアップロード通信クラス(BinaryFileUploadCommunicator)を直接利用して通信する場合、Communicate メソッドの引数にファイルパスを渡す。

- リクエストヘッダの作成

送信時に HTTP リクエストのヘッダに情報を格納するために、IDictionary ジェネリックインターフェイス実装クラスのインスタンスを利用する。ここでは、サーバ処理を一意に識別するための文字列であるリクエスト名を格納する必要がある。リクエスト名を格納する際のキーとして”requestName”を指定する。

- 進行状況通知イベントハンドラの登録

必要に応じて、進行状況通知イベントハンドラを登録する。このイベントを用いることで、プログラクサーなどに通信の進行状況を表示することができる。なお、進行状況を表す値は、百分率で表された整数値で、送信終了で 50、受信終了で 100 となる。実装に関しては、『FB-01 イベント処理機能』を参照すること。

- 実行とエラー処理

通信を実行するには、BinaryFileUploadCommunicator クラスのインスタンスに対して、Communicate メソッドを呼び出す。このとき、引数に送信用パラメータと、リクエストヘッダ情報を格納した IDictionary ジェネリックインターフェイス実装クラスのインスタンスを渡す。

通信が失敗した場合や、サーバでエラーが発生してエラー電文が返却された場合、BinaryFileUploadCommunicatorは例外をスローする。例外の種類については「表 8 例外の種類」を参照すること。

ServerExceptionのインスタンスには、エラー種別を示す文字列(ErrorTypeプロパティ)と、エラー情報のリスト(Errorsプロパティ)が格納される。エラー種別とその内容については「表 9 エラー種別とエラーの内容」を参照すること。

(3) リクエストごとに通信先URLを変更する場合

リクエストごとに通信先URLを変更する場合は、`MultipartUploadCommunicator`クラスのインスタンスの`Address`プロパティにURLを設定する。実装例をリスト 12に示す。

```
communicator.Address = "http://terasoluna.local/index.aspx";
```

リスト 12 リクエストごとに通信先 URL を変更する場合

なお、両方指定していた場合は、`Address` プロパティに指定した方が優先される。

(4) リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更する場合は、`MultipartUploadCommunicator`クラスのインスタンスの`RequestTimeout`プロパティにリクエストタイムアウト時間を設定する。実装例をリスト 13に示す。

```
// リクエストタイムアウト時間を5秒に設定  
communicator.RequestTimeout = 5000;
```

リスト 13 リクエストごとにリクエストタイムアウト時間を変更する場合

なお、両方指定していた場合は、`RequestTimeout` プロパティに指定した方が優先される。

(5) 通信のキャンセル

通信中に送受信を中止したい場合は、`Cancel` メソッドを呼び出すことで処理をキャンセルすることができる。

```
communicator.Cancel();
```

リスト 14 通信のキャンセル

ただし、`Cancel` メソッドを呼び出すのは、`Communicate` メソッドを呼び出して通信処理を実行したスレッドとは別のスレッドでなければならない。

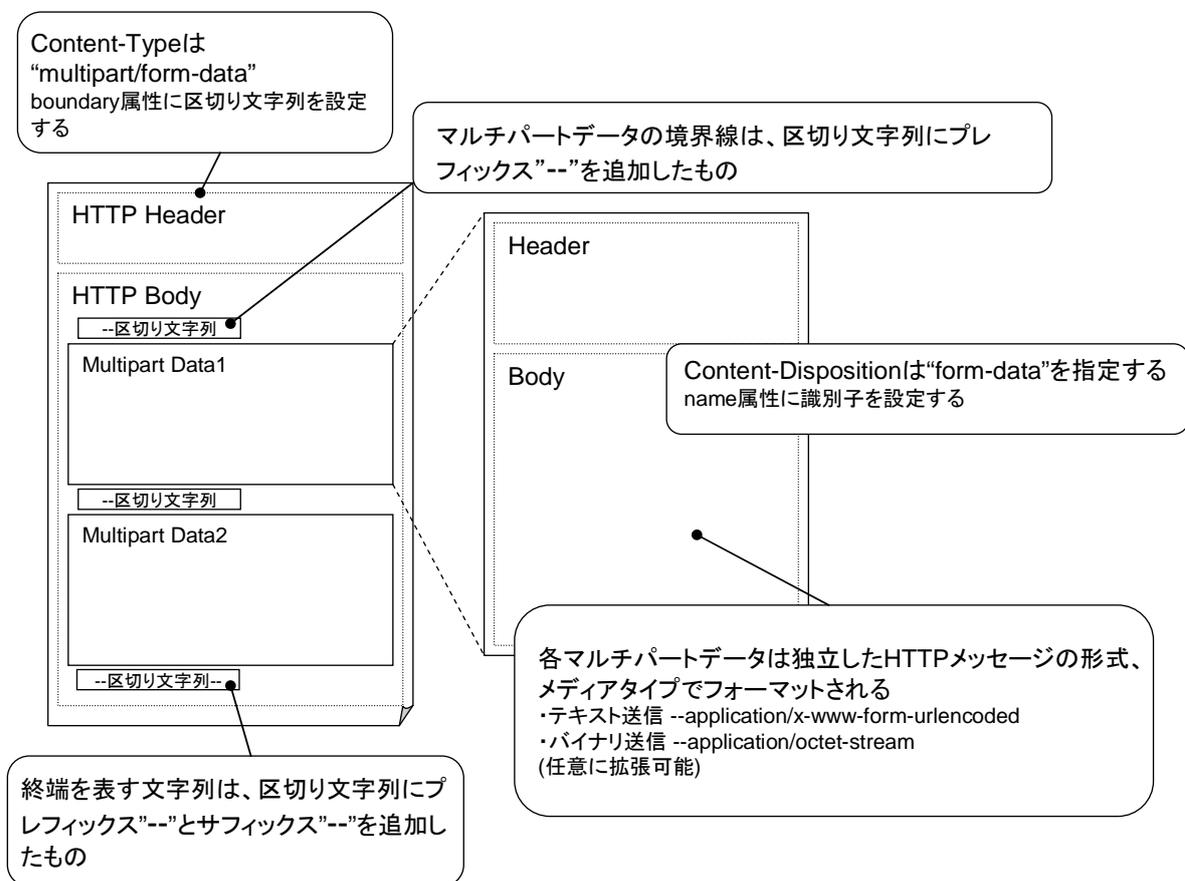
■ 内部構造

◆ Communicateメソッドの処理フロー(共通)

Communicate メソッドの処理フローは、全ての通信クラスで共通なので、『FC-01 XML 通信機能』の内部構造を参照のこと。

◆ マルチパートアップロードの仕様

MultipartUploadCommunicator は、RFC1867 の仕様に従い、マルチパート形式でアップロードを行う。



MultipartUploadCommunicator は、HTTP ヘッダの Content-Type に「multipart/form-data; + boundary=区切り文字列; charset=文字コード」の形式で値を設定する。区切り文字列にはリクエスト電文ごとにグローバル一意識別子 (GUID)を用いて重複する可能性の低いランダムな文字列を設定し、HTTP ボディの各マルチパートデータの区切り文字列として使用する。文字コードには UTF-8 を設定する。

MultipartUploadCommunicator は、HTTP ボディに、送信用パラメータとして設定された MultipartElement 派生クラス (MultipartFileElement および MultipartValueElement) の

数だけ生成したマルチパートデータを設定する。各マルチパートデータは、HTTP ヘッダに設定された区切り文字列にプレフィックス"--とサフィックス"--を追加した文字列で区切られる。マルチパートデータは、それぞれ独立した HTTP メッセージ形式でフォーマットされ、ヘッダとボディを持つ。

MultipartFileElement から生成するマルチパートデータの、Content-Type ヘッダには「application/octet-stream」を設定する。Content-Disposition ヘッダには「form-data; name=識別子名; filename=ファイル名」の形式で値を設定する。また、ファイル名は Base64 エンコードして設定する。

MultipartValueElement から生成するマルチパートデータの Content-Type ヘッダには「application/x-www-form-urlencoded」を設定する。また、本文は ISO-2022-JP でエンコードして設定する。

■ 拡張ポイント

TERASOLUNA Client Framework for .NET が標準で提供するファイルアップロード用通信クラスは、レスポンスとして受信できるのは XML 形式のみだが、拡張を行うことにより、レスポンスとして XML 形式以外の電文(ファイルなど)を受信することも可能となる。

拡張ポイントの詳細に関しては、『FC-01 XML 通信機能』を参照のこと。

■ 関連機能

- ◆ CM-04: ビジネスロジック生成機能
- ◆ FC-01: イベント処理機能

FC-03 ファイルダウンロード機能

■ 概要

本機能は、HTTP 通信を行い、TERASOLUNA Server Framework for Java/.NET で開発されたサーバ AP と連携してファイルダウンロードを行うための仕組みを提供する。

本機能の通信では、リクエストは XML 形式のみである。また、ファイルダウンロードした際の受信データはファイルとして保存する。

本機能を利用することで、イベントコントローラ経由、またはファイルダウンロード用通信クラスを直接利用して、ファイルダウンロードを実行できる。

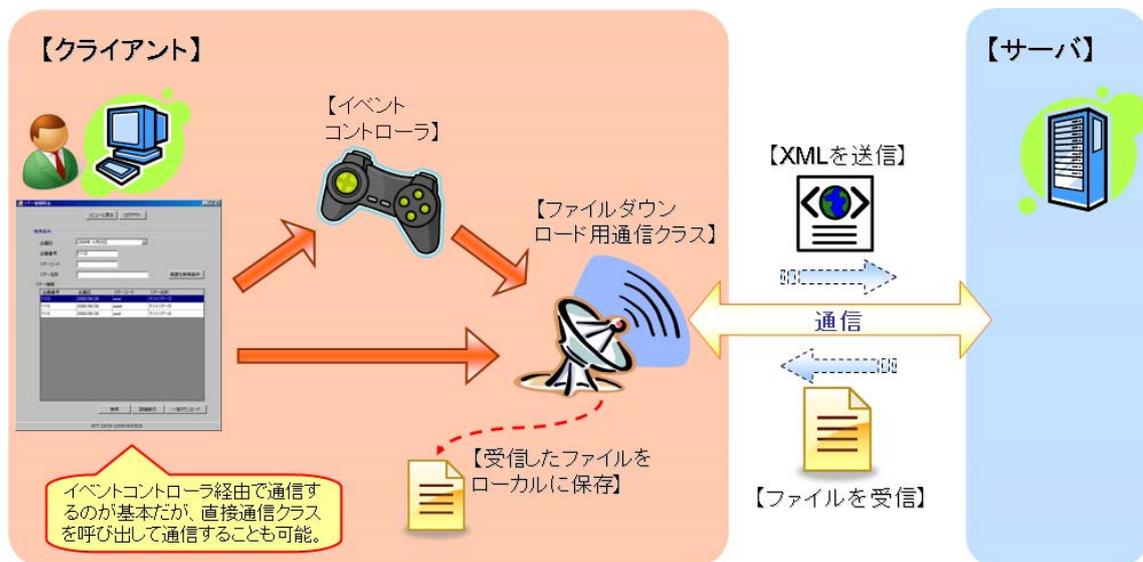


図 1 概念図

本資料では、イベントコントローラを使用するパターンと、直接通信クラスを使用するパターンに分けて説明を行う。

■ 使用方法(イベントコントローラ経由で、通信を行う場合)

ファイルダウンロードを行う場合、ビジネスロジッククラスとして BinaryFileDownloadBLogic クラスを利用することで、イベントコントローラ経由で処理を実現することができる。

◆ アプリケーション構成ファイル

(1) 接続先URLの設定

アプリケーション構成ファイル(App.config)に接続先 URL を設定する。接続先 URL は、アプリケーション構成ファイルの appSettings/add 要素に、key 属性を”BaseUrl”として、value 属性に通信先 URL を設定する。

以下に、接続先 URL を設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="BaseUrl" value="http://terasoluna.local/index.aspx"/>
  </appSettings>
</configuration>
```

リスト 1 アプリケーション構成ファイルに接続先 URL を設定する例

(2) リクエストタイムアウト時間の設定

リクエストタイムアウト時間を設定するには、アプリケーション構成ファイルの appSettings/add 要素に、key 属性を”RequestTimeout”として、value 属性にタイムアウト時間(ミリ秒)を設定する。

アプリケーション構成ファイルに設定したタイムアウト時間は、全リクエストに対して適用される。リクエストごとにタイムアウト時間を変更する場合の方法は、後述の「リスト 8 リクエストタイムアウト時間を変更する場合の実装例」および「リスト 12 リクエストごとにリクエストタイムアウト時間を変更する場合」を参照すること。

以下に、リクエストタイムアウト時間をアプリケーション構成ファイルに設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- 標準のリクエストタイムアウト時間を1時間(3600秒)とする -->
    <add key="RequestTimeout" value="3600000"/>
  </appSettings>
</configuration>
```

リスト 2 アプリケーション構成ファイルにリクエストタイムアウト時間を設定する例

(3) ビジネスロジック設定ファイルとデータセット変換設定ファイルのパスの設定

ファイルアップロードビジネスロジック(BinaryFileDownloadBLogic)を利用するには、利用するビジネスロジックをビジネスロジック設定ファイルに記述し、さらにビジネスロジック入力データセットへの変換設定をデータセット変換ファイルに記述する必要がある。

記述する各設定ファイルのパスは、アプリケーション構成ファイルで指定する。以下に、ビジネスロジック設定ファイルのパスとデータセット変換設定ファイルのパスを指定する例を示す。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
      type="TERASOLUNA.Fw.Common.Configuration.BLogic.
        BLogicConfigurationSection, TERASOLUNA.Fw.Common"/>
    <section name="conversionConfiguration"
      type="TERASOLUNA.Fw.Client.Configuration.Conversion.
        ConversionConfigurationSection, TERASOLUNA.Fw.Client"/>
  </configSections>

  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="Config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>

  <conversionConfiguration>
    <files>
      <!-- データセット変換定義ファイルのパス設定 -->
      <file path="Config¥ConverterConfiguration.config"/>
    </files>
  </conversionConfiguration>
</configuration>
```

リスト 3 アプリケーション構成ファイルにビジネスロジック設定ファイルのパスとデータセット変換設定ファイルのパスを記述する例

ビジネスロジック設定ファイルの詳細については『CM-04 ビジネスロジック生成機能』を、データセット変換設定ファイルの詳細については『FB-02 データセット変換機能』を参照すること。

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルにファイルダウンロードビジネスロジックのビジネスロジック名とクラスの完全修飾型名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。

以下に、ビジネスロジック設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8"?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <blogic name="download" type="TERASOLUNA.Fw.Client.BLogic.
    BinaryFileDownloadBLogic, TERASOLUNA.Fw.Client" />
</blogicConfiguration>
```



リスト 4 ビジネスロジック設定ファイルの記述例

ビジネスロジック設定ファイルの詳細に関しては『CM-04 ビジネスロジック実行機能』を参照すること。

◆ データセット変換設定ファイル

データセット変換設定ファイルには、画面データセットからビジネスロジック入力データセットへの変換設定を記述する。

以下に、データセット変換設定ファイルの記述例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<conversionConfiguration
  xmlns="http://www.terasoluna.jp/schema/ConversionSchema.xsd">
  <convert id="downloadBLogic">
    <param>
      . . . 省略 . . .
    </param>
    <result>
    </result>
  </convert>
</conversionConfiguration>
```

● result 属性は設定しない。

リスト 5 データセット変換設定ファイルの記述例

データセット変換設定ファイルの詳細に関しては『FB-02 データセット変換機能』を参照すること。

◆ 実装方法

ファイルダウンロード処理を実装するには、(1)画面データセットの作成、(2)ビジネスロジック入力データセットの作成、(3)イベントコントローラのプロパティの設定、(4)進捗状況通知イベントハンドラの実装、(5)イベントコントローラの実行の順番で実装する。

(1) 画面データセットの作成

画面の情報を保持するデータセットを作成する。画面データセット作成方法の詳細に関しては、『FB-01 イベント処理機能』の「画面項目とデータセットのバインド(関連付け)」を参照すること。画面データセットから XML に変換する方法の詳細に関しては、『FC-01 XML 通信機能』の「TODO」を参照すること。

(2) ビジネスロジック入力データセットの作成

XML 電文としてサーバに送信するデータを格納するためのデータセットを作成する。なお、画面データセットをそのまま利用することも出来る。その際は、後述のイベントコントローラのプロパティの設定において、“BLogicParamTypeName”を空文字列にする。

(3) イベントコントローラのプロパティの設定

イベントコントローラのプロパティに、ファイルアップロードビジネスロジックの実行に必要な設定を行う。設定しなければならないプロパティを以下に示す。

表 1 設定しなければならないイベントコントローラのプロパティ

項番	プロパティ	説明
1	BLogicName	ビジネスロジック設定ファイルに記述したファイルダウンロードビジネスロジックのビジネスロジック名。
2	BLogicParamTypeName	ビジネスロジック入力データセットの完全修飾型名。省略した場合、ViewData に設定したインスタンスと同じ型となる。
3	ConvertId	データセット変換設定ファイルに記述したコンバートID。
4	RequestName	サーバ側でファイルダウンロード用のリクエストを受け付けるためのリクエスト名。
5	ViewData	画面データセットのインスタンス。

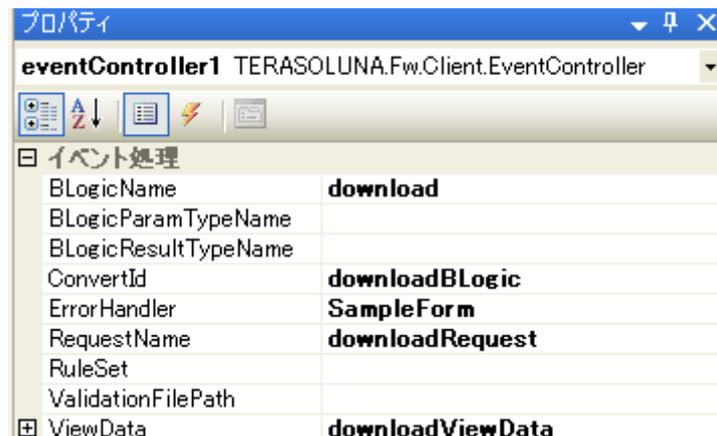


図 2 イベントコントローラのプロパティ設定例

(4) 進行状況通知イベントハンドラの実装

ダウンロードしたファイルの保存先の指定は、イベントコントローラの進行状況通知イベント (ExecuteProgressChanged) のイベントハンドラに実装する。

以下に、イベントハンドラの登録方法と、ダウンロード先ファイル名を指定するイベントハンドラの実装例を示す。

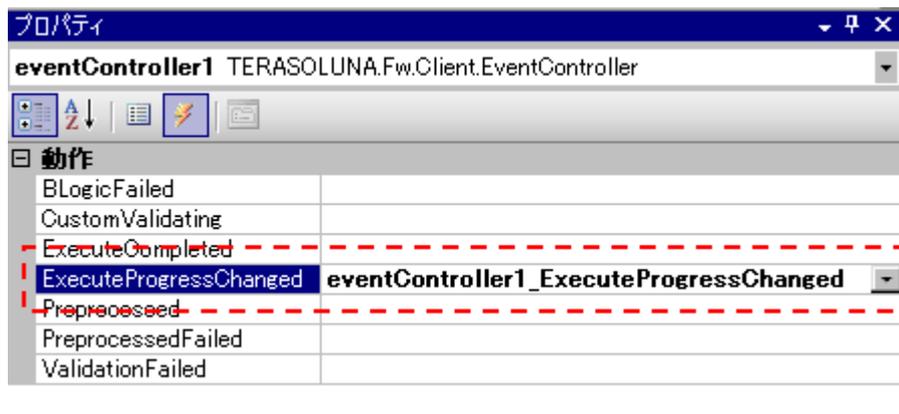


図 3 進行状況通知イベントハンドラの登録

```
private void Sample_ExecuteProgressChanged(object sender,
                                           ExecuteProgressChangedEventArgs e)
{
    // "DownloadReady"キーが存在する場合に、
    // ダウンロードしたファイルを保存するパスを指定する
    if (e.Items.Contains("DownloadReady"))
    {
        string contentFileName = e.Items["ContentFileName"] as string;
        // ダイアログボックスを表示してユーザにファイルパスを指定させる
        SaveFileDialog saveFileDialog = new SaveFileDialog();
        saveFileDialog.Title = "ダウンロードファイルパスの指定";
        saveFileDialog.FileName = contentFileName;

        if (DialogResult.OK == saveFileDialog.ShowDialog())
        {
            // ダウンロードしたファイルを保存するパスをItemsに設定する
            e.Items.Add("DownloadFilePath", saveFileDialog.FileName);
        }
    }
}
```

サーバから返却されたファイル名を取得する。

"DownloadFilePath"をキーとして、Items プロパティにファイルパスを格納する。

リスト 6 進行状況通知イベントハンドラでファイルパスを指定する実装例

進行状況通知イベントハンドラの引数 `ExecuteProgressChangedEventArgs` の `Items` プロパティに "DownloadReady" キーが存在するタイミングで、ダウンロードしたファイルを保存するパスを `Items` プロパティの "DownloadFilePath" キーの値に設定する。受信開始時に実行される `ProgressChanged` イベントにのみ、`Items` に "DownloadReady" が設定される。それ以外のタイミングで実行される `ProgressChanged` イベントに "DownloadReady" が設定されることはない。指定したファイルが既に存在する場合には、ダウンロードしたデータで上書きされる。

引数 `ExecuteProgressChangedEventArgs` に格納される `Items` の詳細に関しては、「表 2 `ExecuteProgressChangedEventArgs` の `Items` プロパティに格納される値」を参照すること。

表 2 ExecuteProgressChangedEventArgs の Items プロパティに格納される値

項番	キー	説明	値
1	DownloadReady	ダウンロード準備を通知するキー。 ※ 進行状況割合を通知する際は、DownloadReady キーを設定しない	true (固定)
2	ContentFileName	サーバから返却されたファイル名。 Content-Disposition ヘッダに filename 属性でファイル名が指定されていた場合のみ格納される。ファイル名がエンコード方式”B” (Base64) でエンコードされていた場合、デコードしたファイル名となる。 ※必ずしも本プロパティを保存するファイル名に利用する必要はない(必ずしもサーバで Content-Disposition ヘッダにファイル名を設定する必要はない)。	DownloadSample.doc など

(5) イベントコントローラの実行

イベントコントローラを実行し、実行結果の確認を行う実装例を以下に示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();

    if (result.Success)
    {
        // 成功時の処理
        // ダウンロードが成功したことを画面に表示する
        MessageBox.Show("ダウンロードに成功しました。ファイルパス→" +
            eventController1.Items["DownloadFilePath"]);
    }
    else
    {
        // 失敗時の処理
        MessageBox.Show("ダウンロードに失敗しました。");
    }
}
```

Items プロパティの"DownloadFilePath"キーから、ダウンロードされたファイルが保存されているパスを取得できる。

リスト 7 ファイルダウンロードの実装例

イベントコントローラの実行後、**ExecutionResult** の **ResultString** プロパティにイベント処理が成功したかどうかを示す文字列が格納されるので、この値を確認する。失敗した場合は、エラー内容に応じた文字列(イベント処理を実行した結果)が格納されているので、適切にエラーハンドリング処理を実装する。

以下に、**ResultString** プロパティの値の一覧を示す。

表 3 ExecutionResult の ResultString 値一覧

項番	状態	値
1	単項目チェックエラー	“validationError”
2	カスタムチェックエラー	“validationError”
3	前処理エラー	“preprocessedError” または前処理イベントで設定する ResultString の値
4	ビジネスロジック実行失敗 (通信中に例外発生)	"communicationException"
5	ビジネスロジック実行失敗 (サーバ側で入力値検証エラー発生)	"serverValidateException"
6	ビジネスロジック実行成功	“success”
7	ビジネスロジック実行失敗 (サーバ側で業務エラー発生)	サーバでレスポンスヘッダに設定された任意の文字列 (例：“serviceException”)

(6) その他

その他、ファイルダウンロード処理における設定を変更する方法を示す。

● リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更したい場合、イベントコントローラの Items プロパティに“RequestTimeout”をキーとして、リクエストタイムアウト時間を設定する。

以下に、実装例を示す。

```
private void button1_Click(object sender, EventArgs e)
{
    // 省略

    // Itemsにリクエストタイムアウト時間 (5000ms)を設定する
    eventController1.Items["RequestTimeout"] = 5000;
    // イベントコントローラの実行
    ExecutionResult result = eventController1.Execute();
}
```

リスト 8 リクエストタイムアウト時間を変更する場合の実装例

なお、Items プロパティとプリケーション構成ファイルの両方にリクエストタイムアウト時間を設定した場合、Items プロパティに設定した値が利用される。

● リクエストヘッダの設定

通信時のリクエストヘッダに値を格納する場合は、IDictionary ジェネリックインターフェイス実装クラスのインスタンスを利用して、イベントコントローラの Items プロパティに設定する。

なお、リクエスト名はイベントコントローラのプロパティに設定すると自動的にリクエストヘッダに格納されて送信される。

```
Dictionary<string, object> requestHeaders
    = new Dictionary<string, object>();

requestHeaders.Add("customData", "foo");
```

リスト 9 リクエストヘッダの設定

- 通信のキャンセル

通信中に送受信を中止したい場合は、イベントコントローラの **Cancel** メソッドを呼び出すことで処理をキャンセルすることができる。キャンセルは非同期処理実行時のみ行うことができる。詳細は、『FB-01 イベント処理機能』を参照すること。

■ 使用方法(直接通信クラスを使用する場合)

ファイルダウンロードを行うには、BinaryFileDownloadCommunicator クラスを直接利用して実現することも出来る。BinaryFileDownloadCommunicator クラスは、イベントコントローラ経由で通信する場合に利用した BinaryFileDownloadBLogic クラスの内部で利用している。

なお、BinaryFileDownloadCommunicator クラスを直接利用して通信する場合は、例外処理などを自分で実装しなければならないことに注意すること。

◆ アプリケーション構成ファイル

本設定に関しては、イベントコントローラ経由で通信を行う場合と同様なので、「(1)接続先URLの設定」、「(2)リクエストタイムアウト時間の設定」を参照すること。

◆ 実装方法

(1) ファイルダウンロード処理の実装

ファイルダウンロード処理を実装するには、①BinaryFileDownloadCommunicator クラスのインスタンスの作成、②送信用データセットの作成、③リクエストヘッダの作成、④進行状況通知イベントハンドラの登録、⑤実行とエラー処理、⑥ファイル保存先パスの取得の順番で実装する。

以下に、BinaryFileDownloadCommunicator クラスで通信を実行する処理の実装例を示す。

```
// ①通信クラスのインスタンスを生成する
BinaryFileDownloadCommunicator communicator = new
BinaryFileDownloadCommunicator ();

// ②送信パラメータを作成
Sample1Ds inputDataSet = new Sample1Ds ();

// ※必要に応じてリクエストヘッダに値を設定
//...

// ③リクエストヘッダの作成(リクエスト名には"sampleRequest"を指定)
IDictionary<string, string> requestHeaders = new Dictionary<string, string>();
requestHeaders.Add("requestName", "sampleRequest");

// ※必要に応じてリクエストヘッダに値を設定
//...

// ④進行状況通知イベントハンドラを通信クラスへ登録する
communicator.ProgressChanged += new
ExecuteProgressChangedEventHandler (this. Sample_ExecuteProgressChanged);

// ⑤実行とエラー処理
try
{
    DownloadResult result = communicator.Download(paramData, requestHeaders);

    // ダウンロードしたファイルの内容をコンソールに出力する
    using (StreamReader sr = new StreamReader(result.DownloadFilePath))
    {
        Console.WriteLine(sr.ReadToEnd());
    }
}
catch (ServerException e)
{
    // サーバで発生したエラーをハンドリングする必要がある場合、
    // ServerExceptionをキャッチする。
}
}
```

リスト 10 ファイルダウンロード処理の実行例

リスト 10では、Sample1Dsクラスのインスタンスを送信用データセットとしてAPサーバに送信し、リクエスト名"sampleRequest"で指定されたサーバ処理を実行している。また、Sample_ExecuteProgressChengedメソッドでダウンロードしたファイルの保存処理を実装している

(実装例は省略。「進行状況通知イベントハンドラの実装」を参照すること)。

- **BinaryFileDownloadCommunicator** クラスのインスタンスの作成

ファイルダウンロード処理を実装するには、**BinaryFileDownloadCommunicator** クラスのインスタンスを利用する。最初に **BinaryFileDownloadCommunicator** クラスのインスタンスを作成する。このとき、受信用データセットの型を型パラメータとして指定する。なお、指定できる型パラメータは **DataSet** クラスとその派生クラスのみである。

- 送信用データセットの作成

サーバに送信するための送信用データセットを作成する。作成した後、データセットにデータを格納する。

- リクエストヘッダの作成

送信時に **HTTP** リクエストのヘッダに情報を格納するために、**IDictionary** ジェネリックインターフェイス実装クラスのインスタンスを利用する。ここでは、サーバ処理を一意に識別するための文字列であるリクエスト名を格納する必要がある。リクエスト名を格納する際のキーとして”requestName”を指定する。

- 進行状況通知イベントハンドラの登録

ダウンロードしたファイルの保存先を指定する場合は、進行状況通知イベントハンドラを登録する。また、このイベントを用いることで、プログレスバーなどに通信の進行状況を表示することもできる。なお、進行状況を表す値は、百分率で表された整数値で、送信終了で 50、受信終了で 100 となる。実装に関しては、「進行状況通知イベントハンドラの実装」を参照すること。

- 実行とエラー処理

通信を実行するには、**BinaryFileDownloadCommunicator** クラスのインスタンスに対して、**Download** メソッドを呼び出す。このとき、引数に送信用データセットと、リクエストヘッダ情報を格納した **IDictionary** ジェネリックインターフェイス実装クラスのインスタンスを渡す。

通信が失敗した場合や、サーバでエラーが発生してエラー電文が返却された場合、**BinaryFileDownloadCommunicator** は例外をスローする。以下にスローされる例外の種類を示す。

表 4 例外の種類

項番	例外クラス	発生原因
1	ServerException	サーバ処理において例外が発生し、エラー電文が返却された場合。
2	CommunicationException	サーバが見つからないなど、通信に失敗した場合。

ServerException のインスタンスには、エラー種別を示す文字列(**ErrorType** プロパティ)と、エラー情報のリスト(**Errors** プロパティ)が格納される。以下にエラー種別とその内容について示す。

表 5 エラー種別とエラーの内容

項番	ErrorType の値	説明	備考
1	“serverException”	サーバで例外が発生したことを示す。	
2	“serverValidationException”	サーバで入力値検証エラーが発生したことを示す。	Errors プロパティは、ValidationMessageInfo クラスのインスタンスのリストとなる。
3	(その他の文字列)	その他、サーバでクライアントにエラーを通知したいときに、任意の文字列をサーバで格納する。	

- ファイル保存先パスの取得

Communicate メソッドの戻り値となる DownloadResult インスタンスの DownloadFilePath プロパティには、ダウンロードしたファイルの保存に成功した場合、保存したファイルのパスが格納される。なお、ファイルを保存するパスは、進行状況通知イベントハンドラで設定したものである。

以下に、DownloadResult のプロパティの一覧を示す。

表 6 DownloadResult のプロパティ一覧

項番	プロパティ名	説明
1	ResultHeaders	HTTP レスポンスヘッダが格納される。
2	ResultData	データセットのインスタンスが格納される。本データセットのインスタンスが XML となり、リクエスト電文として送信される。
3	DownloadFilePath	ダウンロードしたデータの保存に成功した場合にはファイルパスが格納される。ファイルの保存に失敗した場合、DownloadFilePath プロパティには null が格納される。

(2) リクエストごとに通信先URLを変更する場合

リクエストごとに通信先URLを変更する場合は、BinaryFileDownloadCommunicatorクラスのインスタンスのAddressプロパティにURLを設定する。実装例をリスト 11に示す。

```
communicator.Address = "http://terasoluna.local/index.aspx";
```

リスト 11 リクエストごとに通信先 URL を変更する場合

なお、両方指定していた場合は、Address プロパティに指定した方が優先される。

(3) リクエストごとにタイムアウト時間を変更する場合

リクエストごとにリクエストタイムアウト時間を変更する場合は、BinaryFileDownloadCommunicatorクラスのインスタンスのRequestTimeoutプロパティにリクエストタイムアウト時間を設定する。実装例をリスト 12に示す。

```
// リクエストタイムアウト時間を5秒に設定  
communicator.RequestTimeout = 5000;
```

リスト 12 リクエストごとにリクエストタイムアウト時間を変更する場合

なお、両方指定していた場合は、RequestTimeout プロパティに指定した方が優先される。

(4) 通信のキャンセル

通信中に送受信を中止したい場合は、Cancel メソッドを呼び出すことで処理をキャンセルすることができる。

```
communicator.Cancel ();
```

リスト 13 通信のキャンセル

ただし、Cancel メソッドを呼び出すのは、Download メソッドを呼び出して通信処理を実行したスレッドとは別のスレッドでなければならない。

■ 内部構造

◆ Communicateメソッドの処理フロー(共通)

Downloadメソッドは内部で Communicate メソッドを呼び出している。Communicate メソッドの処理フローは、全ての通信クラスで共通なので、『FC-01 XML 通信機能』の内部構造を参照のこと。

◆ 一時ファイルの保存先

ファイルダウンロード用通信クラスは、受信したダウンロードファイルのストリームを一旦一時ファイルに保存してから、進行状況通知イベントハンドラで設定されたファイルパスに一時ファイルの内容をコピーするような内部動作を行う。

一時ファイルは、Path クラスの GetTempFileName メソッドを実行して取得できる Windows で指定された一時フォルダの一意のパスに保存される。一時ファイルは、イベントで設定されたファイルパスの場所にファイルが保存された後に自動的に削除される。

◆ Content-Dispositionヘッダ

サーバ側のアプリケーションは、ファイルダウンロードのレスポンスをクライアントへ返却する際に、Content-Dispositionヘッダ¹にファイル名を設定することができる。Content-DispositionヘッダにBase64形式でエンコードされたファイル名が格納された場合のみ、デコードされたファイル名が受信開始時("DownloadReady"の状態)に実行される進行状況通知イベントハンドラに渡される。具体的には、進捗状況通知イベントハンドラの引数ExecuteProgressChangedEventArgsのItemsプロパティの"ContentFileName"キーにContent-Dispositionに設定されたファイル名が格納される。

なお、必ずサーバで Content-Disposition ヘッダを設定する必要はない。省略した場合、ExecuteProgressChangedEventArgs にファイル名は設定されず、Items プロパティの"ContentFileName"キーが存在しない状態となる。

¹ HTTP ヘッダの一つ

■ 拡張ポイント

TERASOLUNA Client Framework for .NET が標準で提供するファイルダウンロード用通信クラスは、リクエストとして送信できるのは XML 形式のみだが、拡張を行うことにより、リクエストとして XML 形式以外の電文(ファイルなど)を送信することも可能となる。

拡張ポイントの詳細に関しては、『FC-01 XML 通信機能』を参照のこと。

■ 関連機能

- ◆ CM-04: ビジネスロジック生成機能
- ◆ FC-01: イベント処理機能