↑Top

目次

改訂情報

変更年月日	変更内容
2019-08-01	初版
2019-12-01	第2版 下記を追加・変更しました。 ▪ 「*JavaScript（スクリプト開発モデル）による前処理の実装*」を追加
2020-04-01	第3版 下記を追加・変更しました。 ▪ 「*エレメント・アクションアイテムの作成*」を追加 ▪ 「*エラー処理*」を追加
2020-08-01	第4版 下記を追加・変更しました。 ▪ 「*hichee.d.ts*」に 2020 Summer(Zephirine) の hichee.d.ts を追加
2020-12-01	第5版 下記を追加・変更しました。 ▪ 「*hichee.d.ts*」に 2020 Winter(Azalea) の hichee.d.ts を追加
2021-04-01	第6版 下記を追加・変更しました。 ▪ 「*実装上の補足*」を追加 ▪ 「*hichee.d.ts*」に 2021 Spring(Bergamot) の hichee.d.ts を追加 ▪ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを修正
2021-08-01	第7版 下記を追加・変更しました。 ▪ 「*hichee.d.ts*」に 2021 Summer(Cattleya) の hichee.d.ts を追加
2021-12-01	第8版 下記を追加・変更しました。 ▪ 「*エレメント・アクションアイテムの無効化*」を追加 ▪ 「*付録*」を追加 ▪ 「*hichee.d.ts*」に 2021 Winter(Dandelion) の hichee.d.ts を追加
2022-06-01	第9版 下記を追加・変更しました。 ▪ 「*付録*」に新規アクションを追加 　▪ 変数○に○の各キー名を配列にして代入する 　▪ 変数○に○の各要素の値を配列にして代入する 　▪ 音声または動画○を再生する 　▪ 音声または動画○を一時停止する 　▪ 音声または動画○を再生・一時停止する ▪ 「*hichee.d.ts*」に 2022 Spring(Eustoma) の hichee.d.ts を追加 ▪ 「*エレメント・アクションアイテムの作成*」を 2022 Spring(Eustoma) の資材に基づいた情報に修正

変更年月日	変更内容
2022-12-01	第10版 下記を追加・変更しました。 ■ 「*付録*」に新規エレメントを追加 　■ 時刻入力 　■ 時刻入力（Bulma） 　■ 数値入力（フォーマット） 　■ 数値入力（フォーマット）（Bulma） ■ 「*エレメント・アクションアイテムの作成*」-「*エレメントを実装する*」に「*エレメントの非表示方法を実装する*」を追加 ■ 「*エレメント・アクションアイテムの作成*」-「*エレメントを実装する*」の「*固有プロパティを追加する*」に、rerender および deepObserve 設定の説明を追加 ■ 「*hichee.d.ts*」に 2022 Winter(Freesia) の hichee.d.ts を追加 ■ 「*事前準備*」の Node.js, npm のバージョンを更新
2023-04-01	第11版 下記を追加・変更しました。 ■ 「*hichee.d.ts*」に 2023 Spring(Gerbera) の hichee.d.ts を追加
2023-10-01	第12版 下記を追加・変更しました。 ■ 「*hichee.d.ts*」に 2023 Autumn(Hollyhock) の hichee.d.ts を追加
2024-04-01	第13版 下記を追加・変更しました。 ■ 「*実装上の補足*」に以下を追加 　■ *エレメントのインスタンスを作成* 　■ *親エレメントの取得* 　■ *子エレメントの取得* ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを修正 ■ 「*IM-BloomMakerが提供する標準のエレメントのクラス名と提供開始バージョン一覧*」にコンテンツ種別 imds の情報を追加 ■ 「*IM-BloomMakerが提供する標準のアクションアイテムのクラス名と提供開始バージョン一覧*」にコンテンツ種別 imds の情報を追加 ■ 「*hichee.d.ts*」に 2024 Spring(Iris) の hichee.d.ts を追加
2024-06-01	第14版 下記を追加・変更しました。 ■ 「*事前準備*」の Node.js, npm のバージョンを更新 ■ 「*エレメントを実装する*」の内容を 2024 Spring(Iris) 版に更新 ■ 「*エレメントのサンプル実装*」で実装する郵便番号入力エレメントに、placeholder プロパティを追加 ■ 「*エレメントのサンプル実装*」-「*レイアウトモードで使用可能にする*」を追加 ■ 「*カスタムツールバーの実装*」を追加。 ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを 2024 Spring(Iris) 版に更新 ■ 「*hichee.d.ts*」- 2024 Spring(Iris) の hichee.d.ts を更新

変更年月日	変更内容
2024-10-01	第15版 下記を追加・変更しました。 ■ 「*付録*」に新規エレメントを追加 　■ マークダウン 　■ トグルスイッチ（imds） 　■ 進捗サークル（imds） 　■ 水平罫線（imds） 　■ ステッパー（imds） 　■ アコーディオングループ（imds） 　■ 空状態コンテナ（imds） ■ 「*付録*」に新規アクションを追加 　■ エレメント○の位置へ遷移する 　■ メッセージ○を確認ダイアログで表示する（imds） 　■ ページ○をダイアログで開く（imds） 　■ ダイアログを閉じる（imds） ■ 「*付録*」を最新の状態に更新 ■ 「*IM-BloomMakerが提供する標準のエレメントのクラス名と提供開始バージョン一覧*」に「*「IM-Copilot」カテゴリ*」の情報を追加 ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを 2024 Autumn(Jasmine) 版に更新 ■ 「*hichee.d.ts*」- 2024 Autumn(Jasmine) の hichee.d.ts を追加
2025-04-01	第16版 下記を追加・変更しました。 ■ 「*アクションアイテムを実装する*」にパラメータの必須・任意について追記 ■ 「*付録*」に新規アクションを追加 　■ エレメント○にフォーカスを設定する ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを 2025 Spring(Kamille) 版に更新 ■ 「*hichee.d.ts*」- 2025 Spring(Kamille) の hichee.d.ts を追加
2025-10-01	第17版 下記を追加・変更しました。 ■ 「*付録*」に新規エレメントを追加 　■ 日付入力（カレンダー）（Bulma） 　■ 日付入力（カレンダー）（imds） ■ 「*付録*」を最新の状態に更新 ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを 2025 Autumn(Lilac) 版に更新 ■ 「*hichee.d.ts*」- 2025 Autumn(Lilac) の hichee.d.ts を追加
2026-04-01	第18版 下記を追加・変更しました。 ■ 「*付録*」に新規エレメントを追加 　■ タブセット（Bulma） 　■ タブセット（imds） ■ 「*モジュールプロジェクトについて*」のモジュールプロジェクトを 2026 Spring(Mimosa) 版に更新 ■ 「*hichee.d.ts*」- 2026 Spring(Mimosa) の hichee.d.ts を追加

はじめに

本書の目的

本書は、IM-BloomMaker for Accel Platform（以下 IM-BloomMaker）の前処理の実装方法とサンプル実装を説明します。

説明範囲は以下のとおりです。

- 前処理プログラムの実装方法
- Java によるサンプルプログラム
- JavaScript（スクリプト開発モデル）によるサンプルプログラム
- IM-LogicDesigner のフロー定義でのサンプルプログラム

対象読者

本書では以下のユーザを対象としています。

- IM-BloomMakerを利用して前処理を実装したいユーザ

また、次のドキュメントを読了していると、より理解が深まります。

- IM-BloomMaker ユーザ操作ガイド

Java で前処理プログラムを実装する場合は、Java によるプログラムの開発方法を理解している必要があります。

JavaScript で前処理プログラムを実装する場合は、スクリプト開発モデルによるプログラムの開発方法を理解している必要があります。

ロジックフローで前処理プログラムを実装する場合は、IM-LogicDesigner の仕様、操作方法を理解している必要があります。

サンプルコードについて

本書に掲載されているサンプルコードは可読性を重視しており、性能面や保守性といった観点において必ずしも適切な実装ではありません。 開発においてサンプルコードを参考にされる場合には、上記について十分に注意してください。

本書の構成

- *前処理を実装する*

  前処理プログラムの実装方法について説明します。

- *前処理のサンプル実装*

  Java、 JavaScript および IM-LogicDesigner のフロー定義による前処理プログラムについて説明します。

- *エレメント・アクションアイテムを作成する*

  エレメント、アクションアイテムを実装する際の流れや必要な準備について説明します。

- *エレメントを実装する*

  エレメントの実装方法について説明します。

- *エレメントのサンプル実装*

  添付のサンプルに沿って実装の流れを説明します。

- *エレメント・アクションアイテムの無効化*

  エレメント・アクションアイテムを無効化する方法について説明します。

前処理を実装する

前処理を実装するには、デザイナで作成する画面で必要となる情報はなにか？を決めなければいけません。 必要な情報は、デザイナの変数タブ「入力」($input) で定義します。 $input にはキー名、型、そして構造をプロパティとして定義します。

前処理では、このプロパティにセットする値を生成する処理を実装していきます。 一つの前処理だけでは処理が複雑になる場合、複数の前処理プログラムに分割して実装します。

前処理に外部から値を渡したい場合、送信元からクエリパラメータやリクエストボディとして送信してください。クエリパラメータの例を「*リクエストパラメータの解析*」で説明します。

前処理の実装方式

前処理には以下の3つの実装方式があります。

- Java の前処理クラスを実装する
- JavaScript（スクリプト開発モデル）の前処理スクリプトを実装する
- IM-LogicDesigner のフローを定義する

前処理はルーティングに紐付けられます。

前処理はコンテンツに依存しないので、複数のコンテンツに共通の前処理を指定できます。 一方で、コンテンツを編集する機能であるデザイナやプレビュー画面で前処理を実行することはできません。

> **コラム**
> JavaScript（スクリプト開発モデル）による前処理プログラムの実装は、2019 Winter(Xanadu)から可能です。

前処理の実行順序と引数

ルーティングに指定した URL にアクセスすると、前処理が指定された順に実行されます。 前処理が実行されると、引数としていくつかの値が渡されます。

- パス
  - リクエストのパス
- パス変数
  - スクリプト開発モデル プログラミングガイド - ルーティング - PathVariables を参照してください。
- コンテンツ情報
  - コンテンツの情報
- 解析済みリクエストパラメータ情報
  - 「*リクエストパラメータの解析*」で説明します。
- リクエストオブジェクト
  - 生のリクエストオブジェクト（Java の前処理クラス、JavaScript の前処理スクリプトで取得できます。IM-LogicDesigner のフロー定義では取得できません）

リクエストパラメータの解析

ルーティングに指定した URL に対してパラメータを送信すると、前処理で受信できます。 単純な key-value 形式だけでなく、構造を持ったパラメータも送信できます。

キーを .（ピリオド）でつなげると、Map として解析されます。 [] をつけると、配列として解析されます。

```
http://<host>:<port>/<contextPath>/<ルーティングに定義したURL>?parameter1=value1
&parameter2.property1=prop_value1&parameter2.property2=prop_value2&array1[0]=foo&array1[1]=bar

（幅の都合上改行していますが、本来は1行です。）
```

のようなリクエストは、

```
{
 "parameter1": "value1",
 "parameter2": {
  "property1": "prop_value1",
  "property2": "prop_value2"
 },
 "array1": [
  "foo",
  "bar"
 ]
}
```

のような形に変換され、解析済みリクエストパラメータとして取得できます。

Java の前処理プログラムの場合は

```
public Map<String, Object> execute(final BMContentPreprocessorContext context) throws BloomMakerException {
   final String parameter1 = (String) context.getParsedRequestParameters().get("parameter1");
   final Map<String, String> parameter2 = (Map<String, String>) context.getParsedRequestParameters().get("parameter2");
   final String[] array1 = (String[]) context.getParsedRequestParameters().get("array1");

   // :
}
```

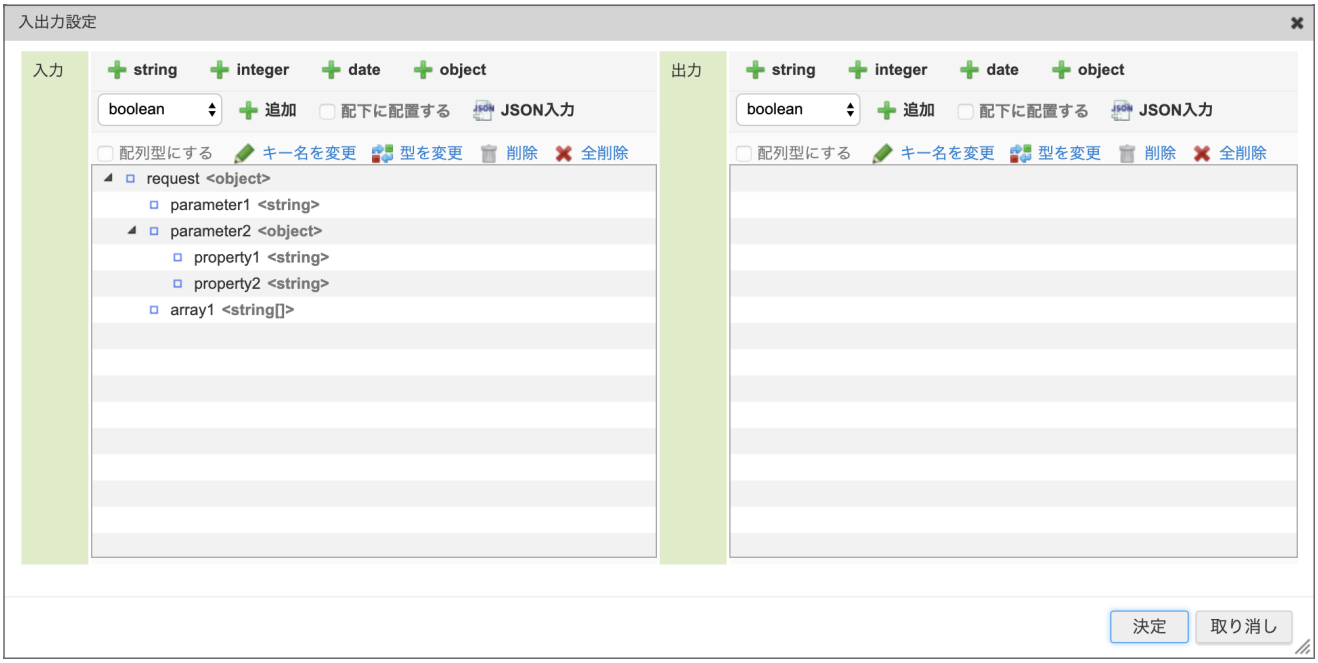
のように取得できます。

JavaScript の前処理プログラムの場合は

```
function execute(context) {
   const parameter1 = context.parsedRequestParameters.parameter1;
   const parameter2 = context.parsedRequestParameters.parameter2;
   const array1 = context.parsedRequestParameters.array1;

   // :
}
```

のように取得できます。

IM-LogicDesigner のフロー定義の場合は



のように定義すると、後続の処理で入力から値を取得できます。 入力のルートにある *request* は、「*フロー定義でパラメータを受けとる*」で説明する解析済みリクエストパラメータ情報を表します。

フロー定義でパラメータを受けとる

フロー定義で様々な入力を取得するには、入出力定義の入力に次のようなキーを持つ object や string を定義します。

- パス
  - キー名：path
  - 型：string
- パス変数
  - キー名：variables
  - 型：object
- コンテンツ情報
  - キー名：content
  - 型：object
- 解析済みリクエストパラメータ情報
  - キー名：request
  - 型：object

型が object のものは、 *リクエストパラメータの解析* のフロー定義のように必要なプロパティを定義します。

> **コラム**
>
> フロー定義の出力を定義する際、デザイナの変数タブの「入力」($input) で JSON エディタの値をコピーし、フロー定義の JSON 入力に貼り付けるとキー名と構造を正確に定義できます。
>
> キー名の誤字は見つけづらい場合がありますので、ぜひ JSON エディタ、JSON 入力をご利用ください。

複数の前処理

複数の前処理が指定された場合、同じキーに対して値をセットすることがあります。 その場合、ルーティングに指定された順に前処理が実行され、同じキーに対して値を上書きしていきます。 後に実行された前処理の結果が最終的な結果として扱われます。

前処理の返却値

前処理の結果は、Java では Map<String, Object> の形で、JavaScript では Object の形で、IM-LogicDesigner のフローでは object として返します。 上述の通り、すべての前処理の結果がまとめられ、コンテンツの実行画面に渡されます。 コンテンツの実行画面では、変数の入力 ($input) として取得できます。

エラー処理

前処理の実行中にエラーが発生した場合、処理を中断し、エラー画面へ遷移させることができます。

前処理専用のエラー画面に遷移する

以下のような実装を行います。

- Java
  - jp.co.intra_mart.foundation.bloommaker.exception.BloomMakerPreprocessException をスローする
- JavaScript
  - Error をスローする
- IM-LogicDesigner
  - エラー終了で終了する

それぞれのエラーにメッセージを指定すると、そのメッセージがエラー画面に表示されます。

> **コラム**
>
> 前処理専用のエラー画面に遷移させるのは、2020 Spring(Yorkshire)から可能です。

> **注意**
>
> 前処理専用のエラー画面に遷移できるようにするため、2020 Spring(Yorkshire)で仕様の変更を行いました。
>
> 以下の場合に該当する場合、指定したエラーメッセージが画面に表示されます。 エラーメッセージに内部情報などを指定していると、意図せぬ情報が漏洩する可能性がありますので注意してください。
>
> - 2019 Winter(Xanadu)以前のバージョンで JavaScript または IM-LogicDesigner で前処理を実装した
> - JavaScript で Error をスローする、IM-LogicDesigner のエラー終了で終了する際に、エラーメッセージを指定した

標準のエラー画面に遷移する

Java に限り、以下のような実装を行うと intra-mart Accel Platform 標準のエラーページへ遷移します。この場合、エラーメッセージを画面に表示することはできません。

- Java
  - jp.co.intra_mart.foundation.bloommaker.exception.BloomMakerException をスローする

> **注意**
>
> 2019 Winter(Xanadu)以前のバージョンでは、JavaScript で Error をスローする、IM-LogicDesigner のエラー終了で終了するように実装した場合でも、標準のエラー画面に遷移します。

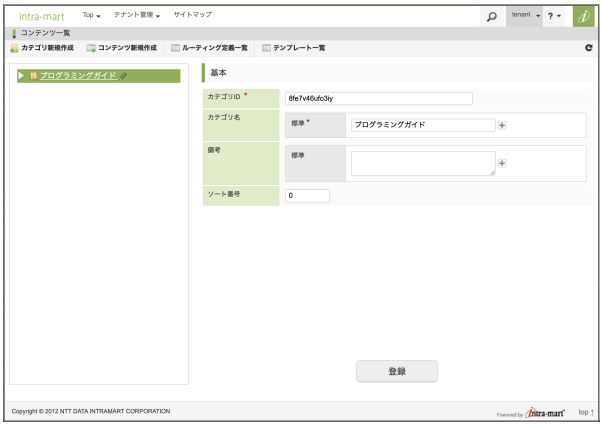
前処理のサンプル実装

ここでは3つの前処理の実装方式でサンプル実装を作成します。前述の通り、前処理はルーティング定義に紐づくものなので、3つの前処理の実装方式に1つずつ、計3つのルーティング定義を作成します。 一方で、コンテンツは1つだけ作成します。このコンテンツは3つのルーティング定義に共通で使用します。

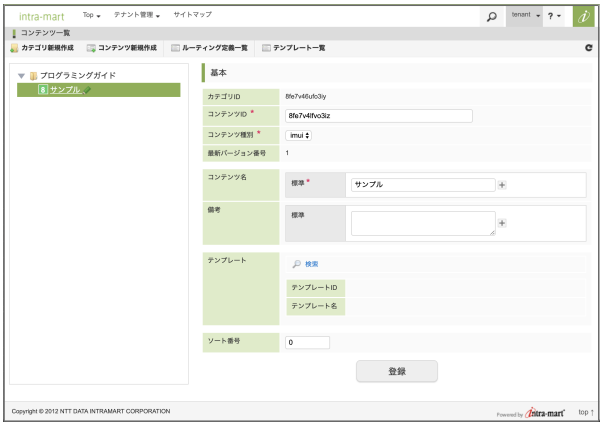
コンテンツの作成

まず共通に使うコンテンツを作成します。

「サイトマップ」→「IM-BloomMaker」→「コンテンツ一覧」で、「コンテンツ一覧」画面を表示し、「カテゴリ新規作成」リンクをクリックします。 画面右側の「カテゴリ名」に「プログラミングガイド」と入力し、「登録」ボタンをクリック、「登録確認」ダイアログで「決定」ボタンをクリックします。

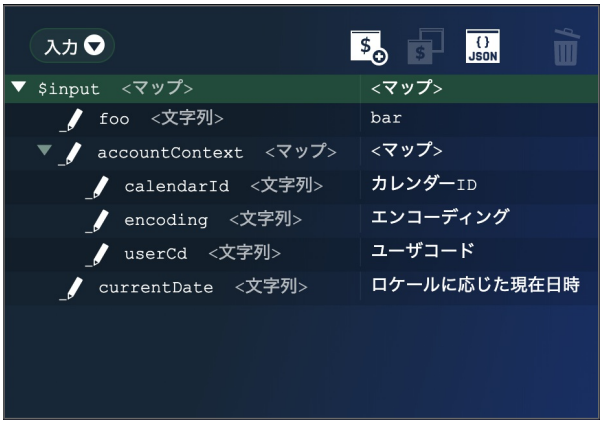


次に作成した「プログラミングガイド」カテゴリを選択した状態で「コンテンツ新規作成」リンクをクリックします。 画面右側の「コンテンツ名」に「サンプル」と入力し、「登録」ボタンをクリック、「登録確認」ダイアログで「決定」ボタンをクリックします。

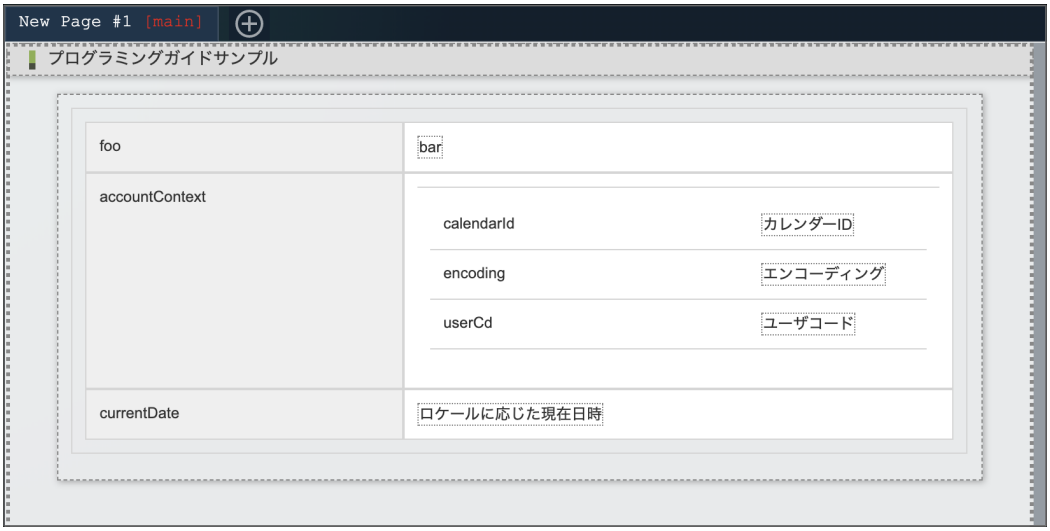


「デザイン編集」ボタンをクリックし、デザイナ画面を開きます。

実装した前処理の結果を受け取るためには、Im-BloomMakerのデザイナ画面で入力を設定する必要があります。右側の「変数」タブをクリック、変数のドロップダウンで「入力」を選択し、入力を設定します。設定内容は次の通りです。



エレメントは以下のように配置します。



foo の右側のラベルの textContent には *$input.foo* を指定します。 calendarId の右側のラベルの textContent には *$input.accountContext.calendarId* を指定します。

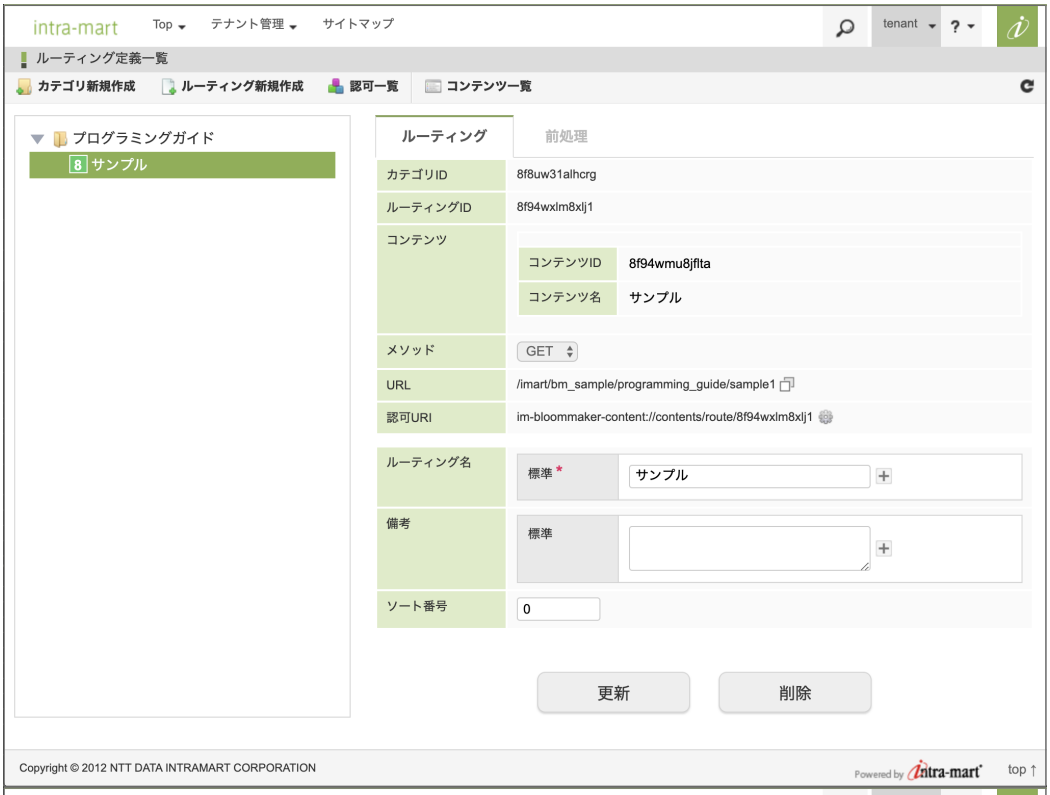
他の項目も同様に、テーブルの左側の文字列と同じキーの変数を右側のラベルのプロパティ textContent に指定します。

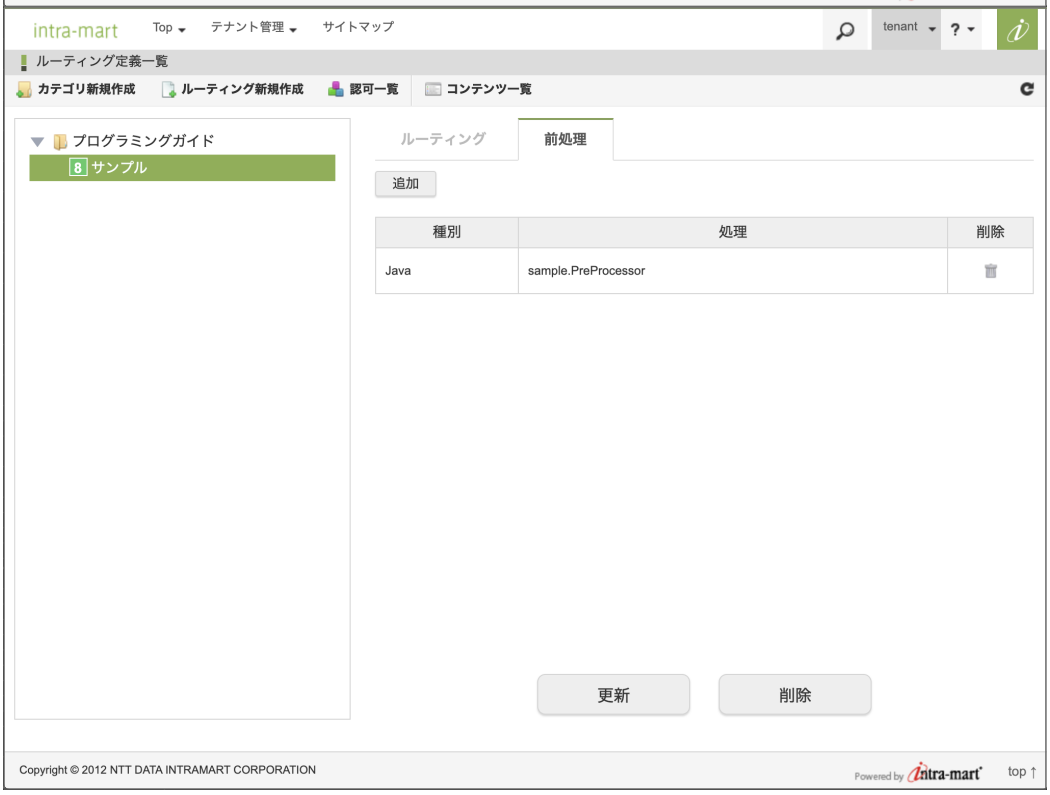
Java による前処理の実装

Java で前処理を実装するには、 *jp.co.intra_mart.foundation.bloommaker.route.preprocess.BMContentPreprocessor* を実装したクラスを作成してください。

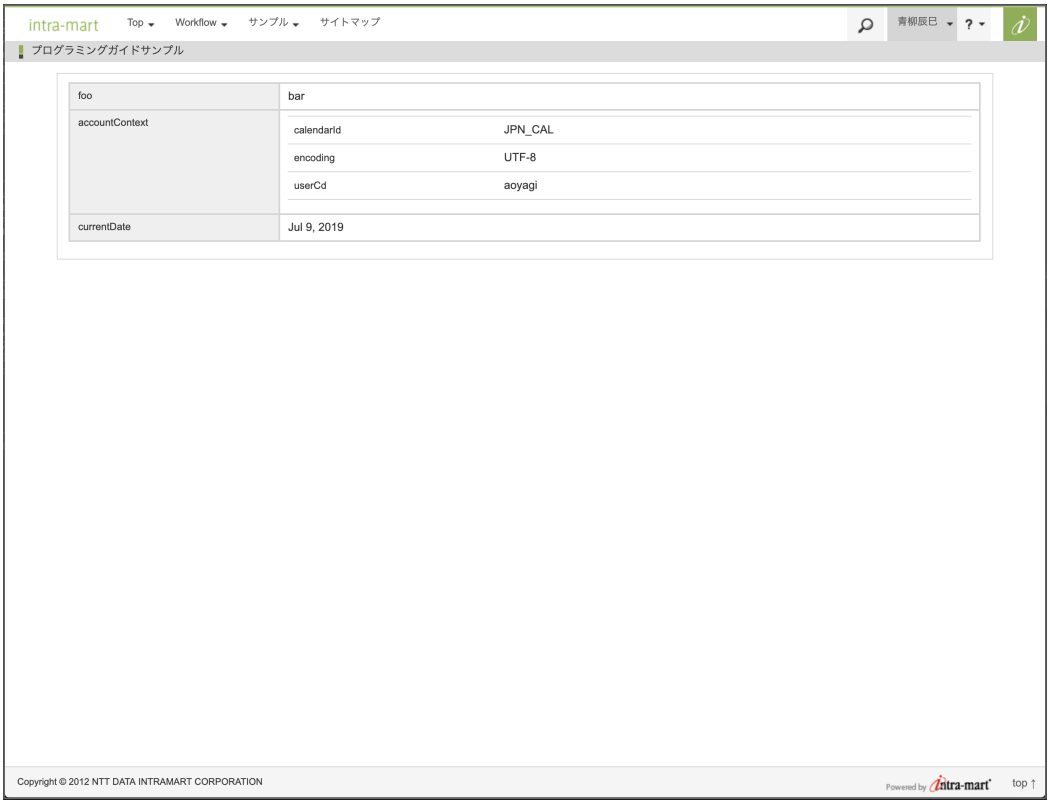
```
public class PreProcessor implements BMContentPreprocessor {

    @Override
    public Map<String, Object> execute(final BMContentPreprocessorContext context) throws BloomMakerException {
        // 返却するマップ
        final Map<String, Object> result = new HashMap<>();

        // 単純なkey-valueをセットします。
        result.put("foo", "bar");

        // アカウントコンテキストをセットします。
        final Map<String, Object> accountContextMap = new HashMap<>();
        final AccountContext accountContext = Contexts.get(AccountContext.class);
        accountContextMap.put("calendarId", accountContext.getCalendarId());
        accountContextMap.put("encoding", accountContext.getEncoding());
        accountContextMap.put("userCd", accountContext.getUserCd());
        result.put("accountContext", accountContextMap);

        // リクエストパラメータを取得します。
        final String targetLocale = (String) context.getParsedRequestParameters().get("locale");

        // 取得したロケールに応じたフォーマットで現在日時をフォーマットします。
        final DateTimeFormatSetInfo[] formats = SystemDateTimeFormat.getFormatSets();
        final String formatsetId = Arrays.asList(formats).stream()
                .filter(format -> format.getLocale().toString().equals(targetLocale))
                .findFirst().map(format -> format.getFormatSetId())
                .orElse(SystemDateTimeFormat.getDefaultFormatSet().getFormatSetId());
        final String format = SystemDateTimeFormat.getFormats(formatsetId)
                .get("IM_DATETIME_FORMAT_DATE_STANDARD");
        final DateTimeFormatter formatter = DateTimeFormatter.withPattern(format);
        result.put("currentDate", formatter.format(new Date()));

        // 何らかのエラーが発生したことを示すフラグ。
        // 本来は API の返り値を格納したり try-catch 構文の中で例外をスローしたりします。
        final boolean someError = false;
        if (someError) {
            throw new BloomMakerPreprocessException("Some error has occured.");
        }
        // 結果として、次のようなオブジェクトを返します。
        // {
        //   "foo": "bar",
        //   "accountContext": {
        //     "calendarId": "カレンダーID",
        //     "encoding": "エンコーディング",
        //     "userCd": "ユーザコード"
        //   },
        //   "currentDate": "ロケールに応じた現在日時"
        // }

        return result;
    }
}
```

ルーティングは以下のように定義します。

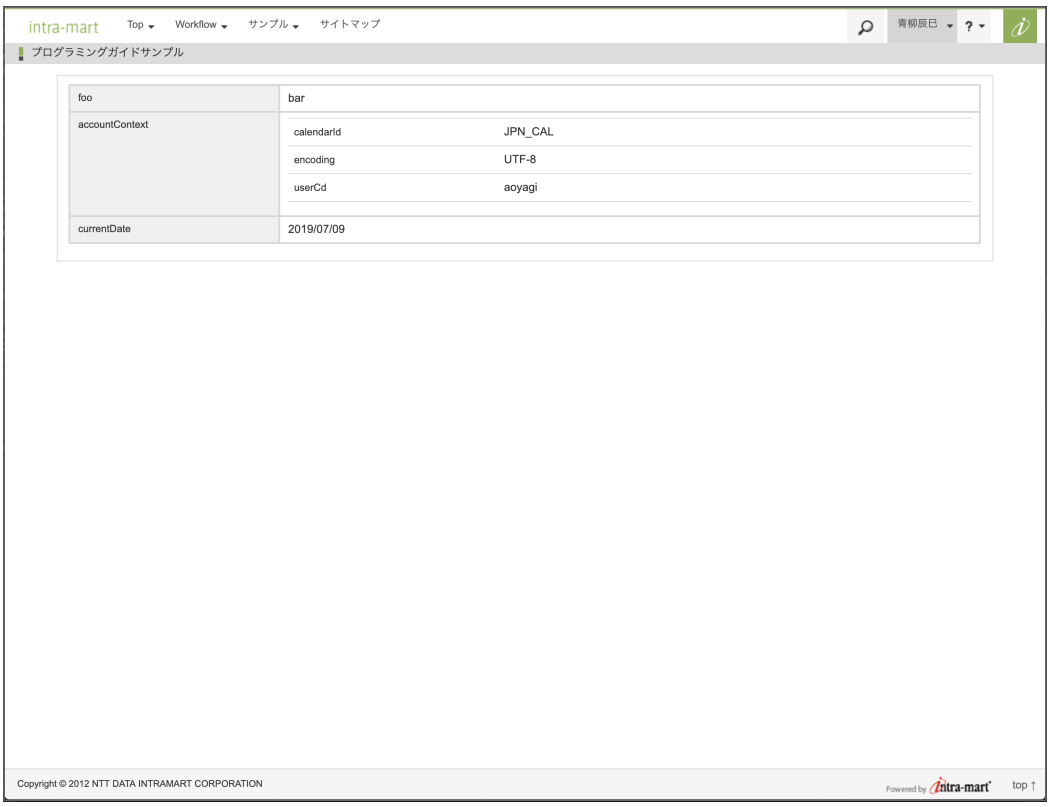




ルーティングに指定した URL にアクセスすると、以下のように表示されます。



URLに *?locale=ja* を追加すると、currentDate の表示が変化します。



JavaScript（スクリプト開発モデル）による前処理の実装

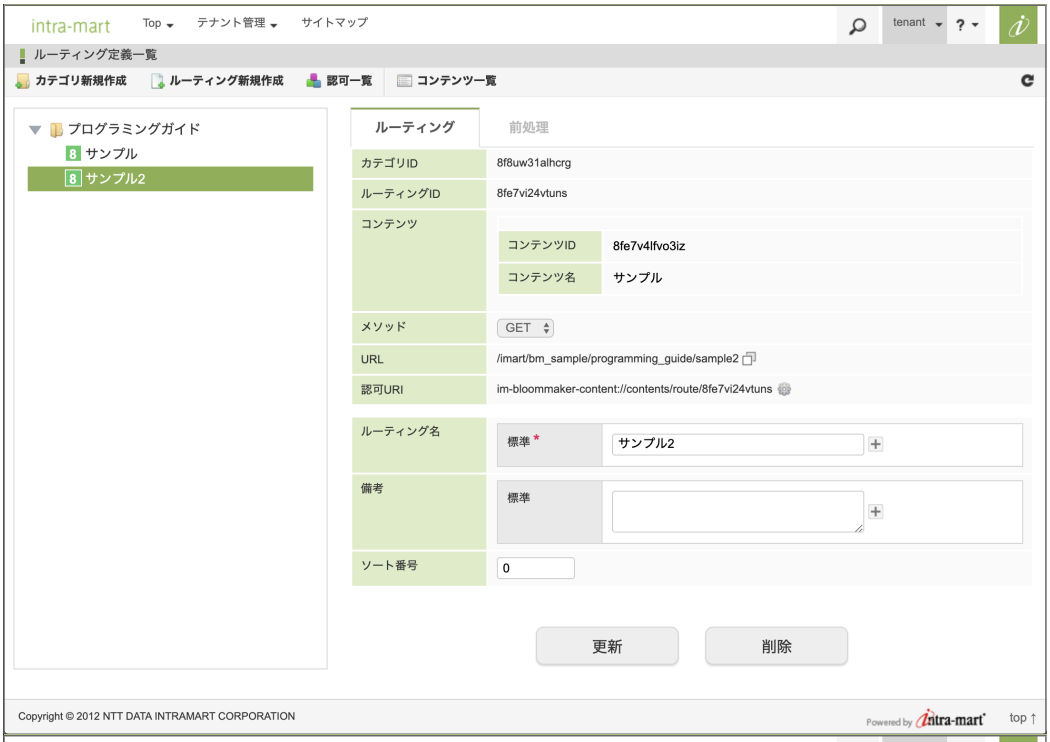
上記の Java による前処理プログラムを JavaScript（スクリプト開発モデル）でも実装してみます。

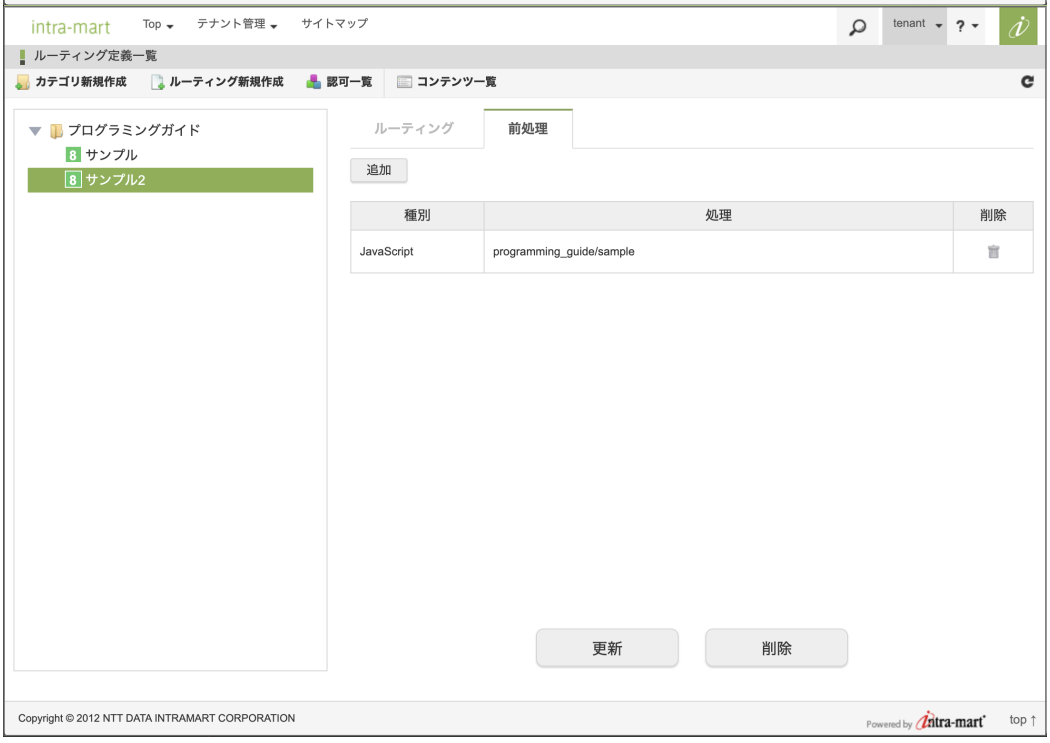
JavaScript で前処理を実装するには、 *execute* 関数を実装したスクリプトを作成してください。

```
function execute(context) {

  // 引数の context は以下のような構造のオブジェクトです。
  //
  // {
  //   path: 'リクエストのパス',
  //   pathvariables: {}, // パス変数
  //   content: {}, // コンテンツの情報
  //   parsedRequestParameters: {}, // 解析済みリクエストパラメータ情報
  //   request: {} // 生のリクエストオブジェクト
  // }

  // 返却するマップ
  let result = {};

  // 単純なkey-valueをセットします。
  result.foo = "bar";

  // アカウントコンテキストをセットします。
  let accountContext = Contexts.getAccountContext();
  let accountContextMap = {
    calendarId: accountContext.calendarId,
    encoding: accountContext.encoding,
    userCd: accountContext.userCd
  };
  result.accountContext = accountContextMap;

  // リクエストパラメータを取得します。
  let targetLocale = context.parsedRequestParameters.locale;

  // 取得したロケールに応じたフォーマットで現在日時をフォーマットします。
  let formats = SystemDateTimeFormat.getFormatSets().data;
  let formatsetId = SystemDateTimeFormat.getDefaultFormatSet().data.formatSetId;
  for (let i = 0, len = formats.length; i < len; i++) {
    let format = formats[i];
    if (format.locale === targetLocale) {
      formatsetId = format.formatSetId;
      break;
    }
  }
  let format = SystemDateTimeFormat.getFormats(formatsetId).IM_DATETIME_FORMAT_DATE_STANDARD;
  result.currentDate = DateTimeFormatter.format(format, new Date())

  // 何らかのエラーが発生したことを示すフラグ。
  // 本来は API の返り値を格納したり try-catch 構文の中で例外をスローしたりします。
  let someError = false;
  if (someError) {
     throw new Error("Some error has occured.");
  }
  // 結果として、次のようなオブジェクトを返します。
  // {
  //   "foo": "bar",
  //   "accountContext": {
  //     "calendarId": "カレンダーID",
  //     "encoding": "エンコーディング",
  //     "userCd": "ユーザコード"
  //   },
  //   "currentDate": "ロケールに応じた現在日時"
  // }

  return result;
}
```

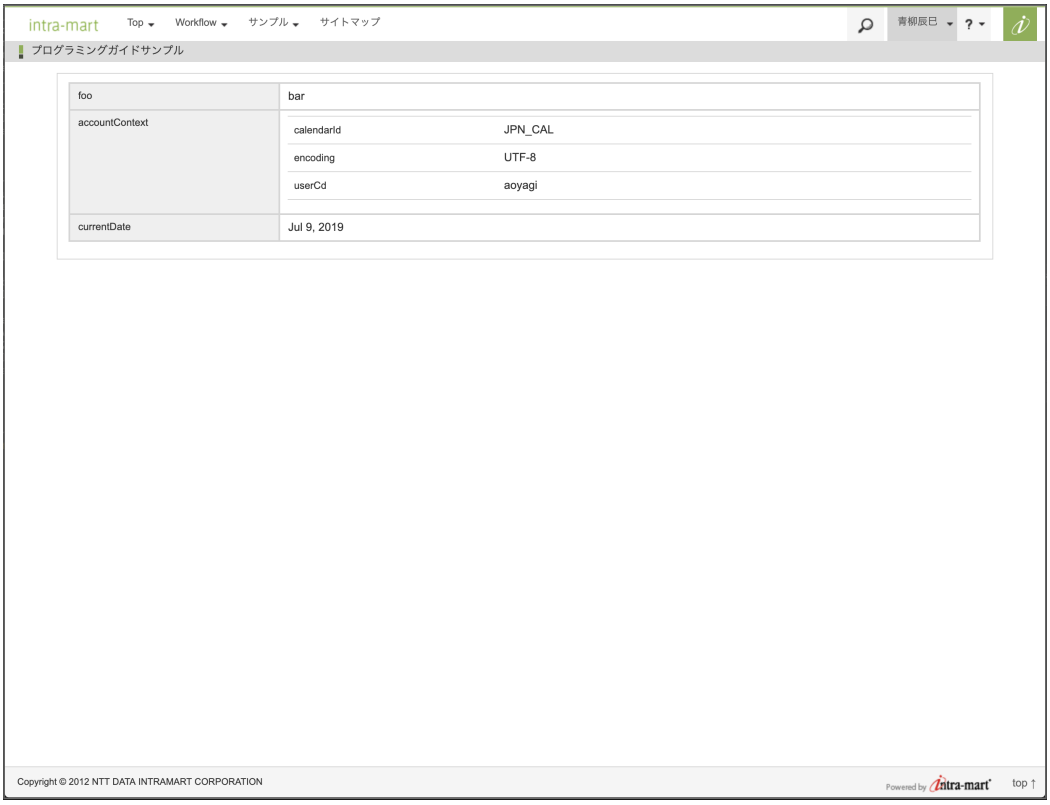
作成したスクリプトは、 *programming_guide/sample.js* として保存します。

ルーティングは以下のように定義します。

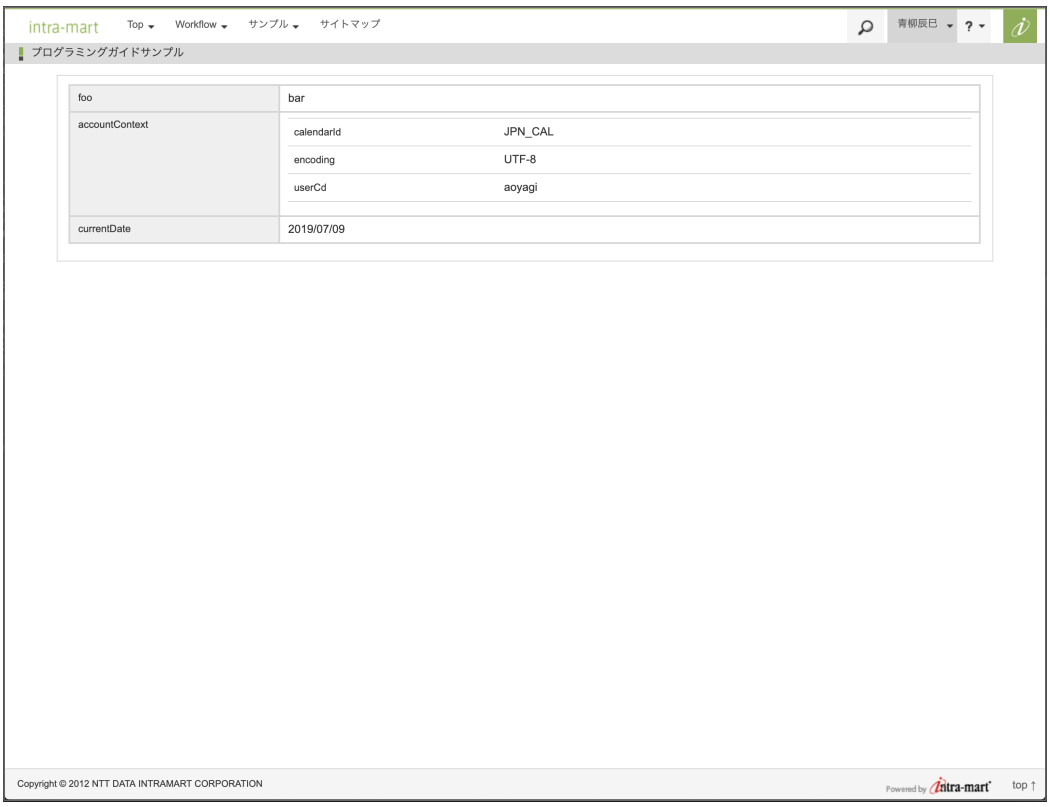




ルーティングに指定した URL にアクセスすると、以下のように表示されます。



URLに *?locale=ja* を追加すると、currentDate の表示が変化します。



IM-LogicDesigner のフロー定義による前処理の実装

上記の Java による前処理プログラムを IM-LogicDesigner でも実装してみます。

IM-LogicDesigner の JavaScript 定義を新規に作成し、以下のような実装を行います。

```
function run(input) {
 // 返却するマップ
 const result = {};

 // 単純なkey-valueをセットします。
 result.foo = "bar";

 // アカウントコンテキストをセットします。
 const accountContext = Contexts.getAccountContext();
 const accountContextMap = {
   calendarId: accountContext.calendarId,
   encoding: accountContext.encoding,
   userCd: accountContext.userCd
 }
 result.accountContext = accountContextMap;

 // リクエストパラメータを取得します。
 const targetLocale = input.locale;

 // 取得したロケールに応じたフォーマットで現在日時をフォーマットします。
 let formatsetId = SystemDateTimeFormat.getDefaultFormats()['format-set-id'];
 const formats = SystemDateTimeFormat.getFormatSets();
 if (!formats.error) {
   const formatsData = formats.data;
   for (let i = 0, len = formatsData.length; i < len; i++) {
     if (formatsData[i].locale === targetLocale) {
       formatsetId = formatsData[i].formatSetId;
       break;
     }
   }
 }
 const format = SystemDateTimeFormat.getFormats(formatsetId).IM_DATETIME_FORMAT_DATE_STANDARD;
 result.currentDate = DateTimeFormatter.format(format, new Date());

 // 結果として、次のようなオブジェクトを返します。
 // {
 //   "foo": "bar",
 //   "accountContext": {
 //     "calendarId": "カレンダーID",
 //     "encoding": "エンコーディング",
 //     "userCd": "ユーザコード"
 //   },
 //   "currentDate": "ロケールに応じた現在日時"
 // }

 return result;
}
```

入力値は以下のように定義します。

JSON入力に以下の JSON をペーストし、全ての項目を置き換えることでも定義できます。

入力

```
{
 "request": {
   "locale": ""
 }
}
```

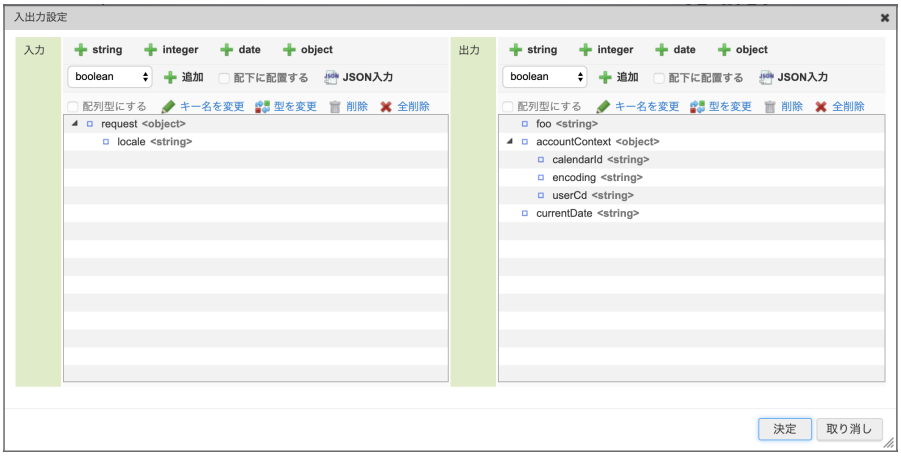
出力

```
{
 "foo": "",
 "accountContext": {
  "calendarId": "",
  "encoding": "",
  "userCd": ""
 },
 "currentDate": ""
}
```

ユーザ定義IDなど、他の項目は適当な値を入力、選択してください。今回はユーザ定義IDを *preprocessor* とします。

次にフロー定義を作成します。

入出力設定を以下のように定義します。



次に先ほど作成したユーザ定義を配置し、開始と終了に接続します。

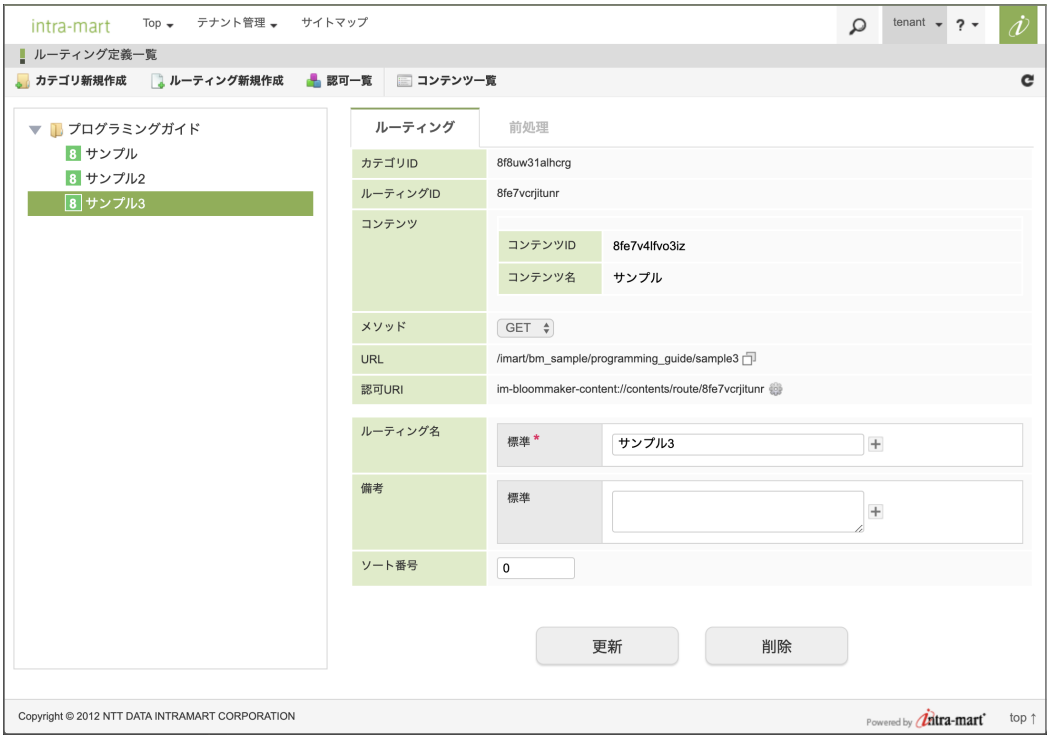
最後にマッピング設定を行います。

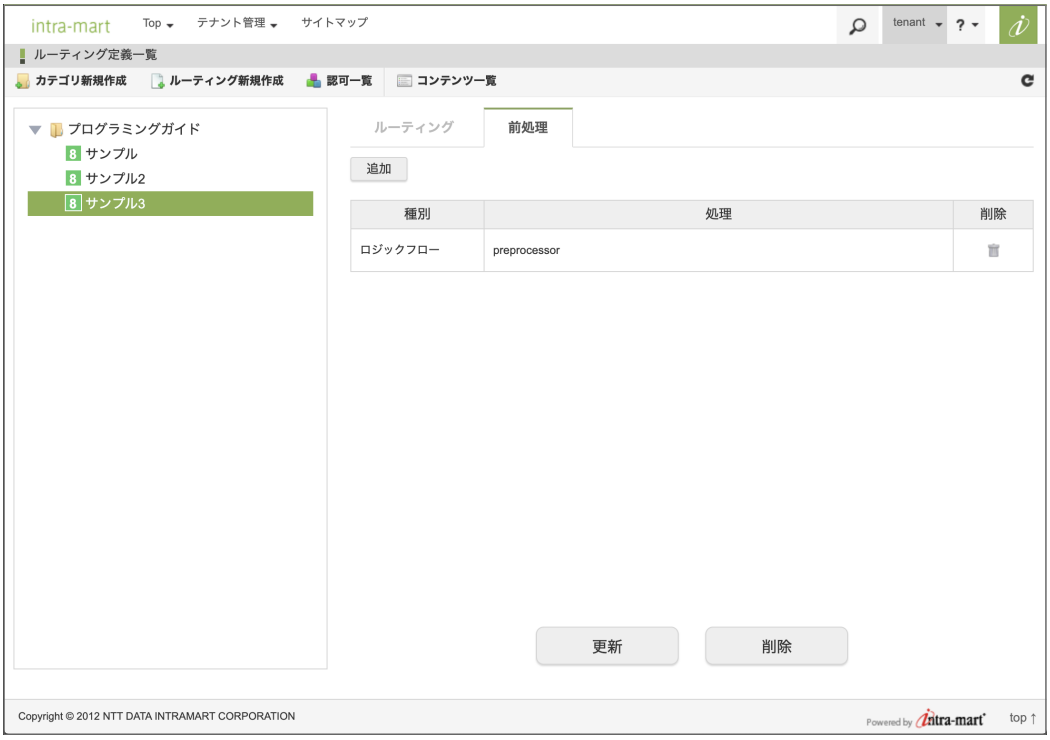
preprocessor のマッピング定義は以下のように定義します。

終了のマッピング定義は以下のように定義します。

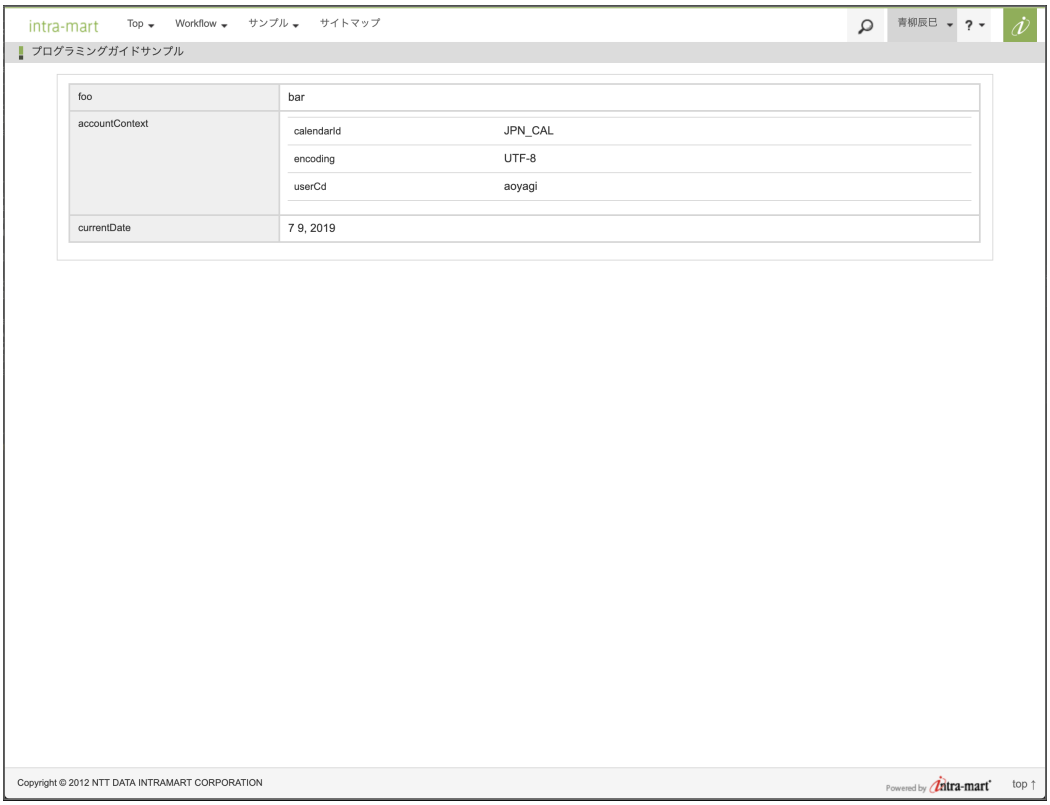
新規保存します。フロー定義IDなど適当な値を入力してください。今回はフローIDなどを preprocessor とします。

コンテンツは上記のものを再度利用します。 ルーティングは以下のように定義します。

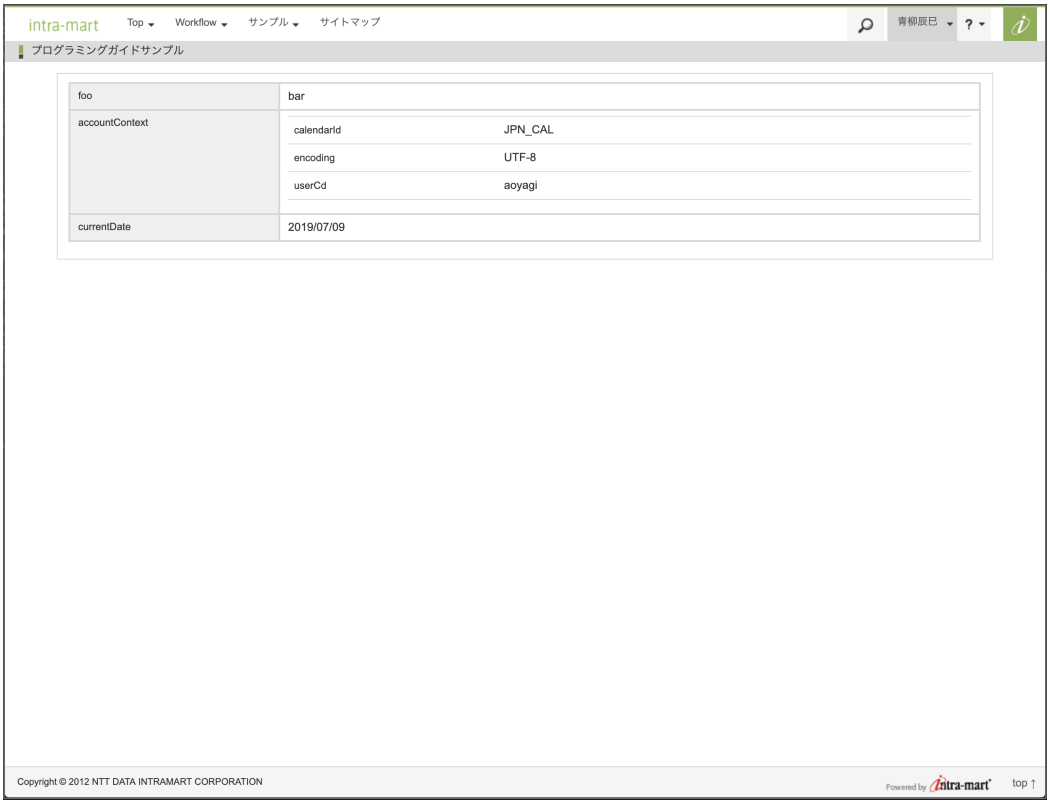




ルーティングに指定した URL にアクセスすると、以下のように表示されます。



URLに *?locale=ja* を追加すると、currentDate の表示が変化します。



サンプル実装の資材

- **Java、JavaScript による前処理の実装のユーザモジュール**
  - IM-Juggling でユーザモジュールとして追加してください。
  - ソース
- **IM-LogicDesigner による前処理の実装**
  - LogicDesigner のインポートからインポートしてください。
  - 以下のフローが定義されます
    - フローカテゴリ：BloomMaker
    - フロー定義ID：preprocessor
    - フロー定義名：preprocessor
- **IM-BloomMaker のコンテンツ・ルーティング定義**
  - BloomMaker のインポートからインポートしてください。
  - 以下のコンテンツ、ルーティング定義が定義されます。
    - コンテンツ
      - プログラミングガイド
        - サンプル
    - ルーティング定義
      - プログラミングガイド
        - サンプル
        - サンプル2
        - サンプル3

注意
上記ファイルのインポート後、IM-BloomMaker ルーティング定義の認可の設定を行ってください。

エレメント・アクションアイテムを作成する

IM-BloomMaker では、任意のエレメントやアクションアイテムを作成し、利用することが可能です。

この章では、TypeScript というプログラミング言語を用いてエレメントやアクションアイテムを実装し、IM-BloomMaker 上でそれらを利用するまでの流れを説明します。

> **注意**
> エレメント・アクションアイテムの作成は、2020 Spring(Yorkshire) 以降のバージョンで可能です。

事前準備

エレメントやアクションアイテムの作成にあたって必要な開発環境を準備します。

Node.js のインストール

Node.js はサーバサイドで動作する JavaScript実行環境です。
以下から v20.8 の最新版をダウンロードし、インストールしてください。

https://nodejs.org/en

正しくインストール出来ている場合、以下のようにコマンドを実行することでバージョンを確認できます。

```
node -v
>> v20.8.1
npm -v
>> 10.1.0
```

エレメント・アクションアイテムは静的ファイルとして実装し、デプロイされます。
Node.js はこの静的ファイルをビルドするために必要です。

Visual Studio Code のインストール

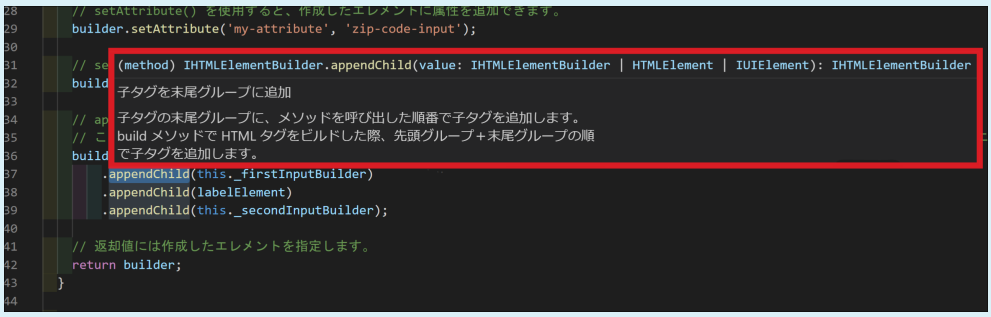
エレメント・アクションアイテムの実装にあたって、コードエディタには Visual Studio Code （以降 VSCode と表記します）の利用を強く推奨します。

インストーラを以下からダウンロードし、インストールしてください。

https://azure.microsoft.com/ja-jp/products/visual-studio-code/

VSCode は、TypeScript を記述する際のコード補完機能 (IntelliSense) や型チェックがとても強力です。

> **コラム**
> エレメント・アクションアイテムの実装には、後述する hichee.d.ts で定義されている様々なメソッドを利用します。
> hichee.d.ts 内に定義されているメソッドや型は、VSCode上でマウスポインターを重ねると、画像のように説明が表示されます。
> また、この状態で Ctrl キーを押しながらクリックすると、そのメソッドや型が定義されている箇所へジャンプします。
> 実装の際には、これらの機能を利用して、メソッドや型の定義や説明を参照することが可能です。



e Builder のインストール

モジュールプロジェクトをimmファイルとしてエクスポートするために、intra-mart e Builder for Accel Platform（以降 e Builder と表記します）を利用します。

e Builder のインストールについての詳細は、「intra-mart e Builder for Accel Platform セットアップガイド」を参照してください。

エレメント・アクションアイテムの作成・追加の流れ

エレメント・アクションアイテムは以下の流れで作成・追加します。

1. コマンドプロンプトで src/main/webpack に移動し、npm install を実行する。

   npm コマンドについては「*npm install*」で説明します。

2. エレメント・アクションアイテムを VSCode で実装する。

   TypeScript を利用してエレメント・アクションアイテムを実装します。

3. コマンドプロンプトで npm run build を実行する。

   npm run build コマンドにより、実装したコードを JavaScript へトランスパイル（変換）し、bundle とよばれるファイルにまとめて出力します。

4. e Builder を利用してユーザモジュール化する。

   e Builder を用いて、bundle を含んだプロジェクトを、imm ファイル形式で出力します。

5. 出力したユーザモジュール を IM-Juggling のプロジェクトに取り込み、warファイルを出力する。

6. warファイルをアプリケーションサーバにデプロイする。

モジュールプロジェクトについて

モジュールプロジェクトを以下のリンクからダウンロードしてください。
im_bloommaker_programming_sample_2026Spring.zip ［2026 Spring(Mimosa) 版］

ダウンロードした im_bloommaker_programming_sample.zip の中には *template* ディレクトリと *implemented* ディレクトリが存在しています。
それぞれのディレクトリの配下にそれぞれ *im_bloommaker_programming_sample* ディレクトリが存在します。これがモジュールプロジェクトです。

- **template**
  エレメントおよびアクションアイテムを実装する上での雛形となる、モジュールプロジェクトのテンプレートです。
  エレメントを作成する際は、このテンプレートを利用してください。
- **implemented**
  上のテンプレートに、サンプルのエレメント・アクションアイテムを実装したモジュールプロジェクトです。
  実装については、「*エレメントのサンプル実装*」と「*アクションアイテムを実装する*」で説明します。

以降、簡略化のため、ディレクトリのパスを、以下のように記載する場合があります。

- im_bloommaker_programming_sample を {EBUILDER_HOME} と記載します。
- im_bloommaker_programming_sample/src/main/webpack を {VSCODE_HOME} と記載します。

モジュールプロジェクトは、以下のようなディレクトリ構成です。

```
im_bloommaker_programming_sample 【このディレクトリを e Builder を利用し immファイルとしてエクスポートします。】
│
├── message.properties
├── message_en.properties
├── message_ja.properties
├── message_zh_CN.properties
├── module.xml 【IM-BloomMaker への依存関係が記述されています。】
└── src
    └── main
        ├── conf
        │   └── bloommaker-config
        │       └── im_bloommaker_sample.xml 【追加コンポーネントの差し込みに必要です。】
        ├── jssp
        │   └── src
        │       └── bloommaker
        │           └── maintenance
        │               └── designer
        │                   └── resources
        │                       ├── im_bloommaker_sample_resource.html 【追加コンポーネントの差し込みに必要です。】
        │                       └── im_bloommaker_sample_resource.js 【追加コンポーネントの差し込みに必要です。】
        ├── public 【Node.js でビルドすると、このディレクトリに静的ファイルが生成されます。】
        └── webpack 【このディレクトリを VSCode で開きます。 このディレクトリは warファイルには含まれません。】
            ├── package.json
            ├── src
            │   ├── d.ts
            │   │   └── hichee.d.ts 【型定義ファイルです。 IM-BloomMaker のバージョンアップに伴い更新されていきます。】
            │   ├── index.ts 【実装したエレメントやアクションアイテムを登録します。】
            │   └── public
            │       ├── actions 【サンプルでは、ここにアクションアイテムを実装します。】
            │       ├── elements 【サンプルでは、ここにエレメントを実装します。】
            │       ├── layouter 【サンプルでは、ここにツールバー拡張を実装します。】
            │       └── messages 【サンプルでは、メッセージをこのディレクトリ配下に定義します。】
            │           ├── component.properties
            │           ├── component_en.properties
            │           ├── component_ja.properties
            │           └── component_zh_CN.properties
            ├── tsconfig.json 【TypeScript の設定ファイルです。】
            ├── webpack.config-local.js 【Node.js の開発時ビルド設定です。】
            └── webpack.config.js 【Node.js のビルド設定です。】
```

package.json

必要なパッケージを package.json へ記載しておくと、後述するnpmコマンドで、プロジェクトの管理や環境構築を行うことができます。

```
{
  "name": "im_bloommaker_programming_sample",
  "version": "1.0.0",
  "description": "IM-BloomMakerにおける、エレメントのサンプル実装です。",
  "main": "",
  "directories": {},
  "dependencies": { ... },
  "devDependencies": { ... },
  "scripts": {
     "build": "cross-env NODE_OPTIONS=--openssl-legacy-provider webpack --mode production"
  },
  "author": "NTT DATA INTRAMART CORPORATION"
}
```

- “dependencies”

  依存するパッケージは `"dependencies"` に記述します。
  ここに記述したパッケージ群は、成果物となる静的ファイルに含まれます。

  `npm install` 実行時には、`"dependencies"` と `"devDependencies"` の記述にしたがって必要なモジュールがインストールされます。

- “devDependencies”

  開発時（ビルド時）にのみ利用するモジュールは、`"devDependencies"` に記述します。
  ここに記述したパッケージ群は、成果物となる静的ファイルに含まれません。

- “scripts”

“scripts” にコマンドを記述することで、エイリアスとして登録することが可能です。
登録したコマンドは、`npm run {エイリアス名}` のように入力すれば実行できます。

例えば上の例のように package.json を記述した場合、そのプロジェクトのディレクトリで `npm run build` と入力して実行すると、実際には `cross-env NODE_OPTIONS=--openssl-legacy-provider webpack --mode production` というコマンドが実行されます。

webpack.config

webpack.config は webpackコマンドの実行時に参照される設定ファイルです。

webpack は、Node.js 上で動作し、HTML や JavaScript, TypeScript , CSS といった静的ファイルを 1 ファイルにまとめることが可能なツールです。 webpack によってまとめて出力されたファイルを bundle と呼び、まとめることを bundle する、と言います。

webpack.config には、bundle する対象のファイルや出力先などを記述します。

> **コラム**
>
> bundle の出力先は、テンプレートでは、`{EBUILDER_HOME}/src/main/public/im_hichee` に設定されています。
> このまま e Builder から ユーザモジュール（immファイル）化して warファイルに取り込めば、Webサーバ上の適切なディレクトリに bundle が配置されるため、変更する必要はありません。

hichee.d.ts

IM-BloomMaker のエレメントやアクションアイテムを TypeScript で実装する上で必要となるインタフェースや型情報が記載された、型定義ファイルです。
VSCode でのコード補完機能 (IntelliSense) や型チェックに利用される他、メソッドの説明なども記載されています。

> **コラム**
>
> このファイルは IM-BloomMaker のアップデートに伴い更新される場合があります。
> ご利用のバージョンに合わせた hichee.d.ts を以下からダウンロードし、`{VSCODE_HOME}/src/d.ts` 配下に配置して利用してください。
>
> **hichee.d.ts （日本語 2026 Spring）**
> **hichee.d.ts （日本語 2025 Autumn）**
> **hichee.d.ts （日本語 2025 Spring）**
> **hichee.d.ts （日本語 2024 Autumn）**
> **hichee.d.ts （日本語 2024 Spring）**
> **hichee.d.ts （日本語 2023 Autumn）**
> **hichee.d.ts （日本語 2023 Spring）**
> **hichee.d.ts （日本語 2022 Winter）**
> **hichee.d.ts （日本語 2022 Spring）**
> **hichee.d.ts （日本語 2021 Winter）**
> **hichee.d.ts （日本語 2021 Summer）**
> **hichee.d.ts （日本語 2021 Spring）**
> **hichee.d.ts （日本語 2020 Winter）**
> **hichee.d.ts （日本語 2020 Summer）**
> **hichee.d.ts （日本語 2020 Spring）**
> **hichee.d.ts （英語 2020 Spring）**

npm install

テンプレートとなるモジュールプロジェクトである `template/im_bloommaker_programming_sample` を利用し開発を進めていきます。

添付の sample.zip を展開後、コマンドプロンプトを起動し、以下のコマンドを実行してください。

```
cd template/im_bloommaker_programming_sample
cd src/main/webpack
npm install
```

> **コラム**
>
> npm は、Node.js をインストールした際に同時にインストールされる、Node.js のパッケージ管理ツールです。
>
> npm コマンドを実行することで、Node.js 上で利用する様々なパッケージを管理することが可能です。

> **コラム**
>
> npm install は、package.json に記載された情報を元に、プロジェクトで必要となるパッケージを node_modules ディレクトリ配下にインストールするコマンドです。
>
> プロキシ環境下で npm install に失敗する場合は、npm のプロキシの設定を確認してください。

なお、前述した webpack も、上記の操作後、{VSCODE_HOME}/node_modules 配下にインストールされています。

実装作業

「*エレメントを実装する*」・「*アクションアイテムを実装する*」の説明にしたがって、エレメント・アクションアイテムを実装します。

bundleの生成

エレメントおよびアクションアイテムの実装が終了したら、bundle を生成します。

エレメントやアクションアイテムを実装した TypeScript ファイル、メッセージプロパティを記述したプロパティファイル、index.ts などを bundle します。 bundle されたファイルは、webpack.config で指定したディレクトリに出力されます。

コマンドプロンプトを起動して、以下のコマンドを実行してください。

```
cd template/im_bloommaker_programming_sample
cd src/main/webpack
npm run build
```

ユーザモジュールの作成・利用

生成したbundleがプロジェクトの {EBUILDER_HOME}/src/main/public/im_hichee ディレクトリ配下に存在することを確認して、プロジェクトをユーザモジュール化します。
e Builder を利用して、プロジェクトをimmファイルとしてエクスポートしてください。

エクスポートの詳細については、e Builderの「アプリケーション開発ガイド」を参照してください。

エクスポートしたimmファイルは、IM-Juggling からプロジェクトに追加することで、warファイルに追加して利用することが可能です。 詳細は、「intra-mart Accel Platform セットアップガイド」を参照してください。

> **コラム**
> プロジェクトをe Builderにインポートすると、作成した bundle にエラーマーカーが表示される場合がありますが、immファイルのエクスポートには問題ありません。

エレメントを実装する

この章では、エレメントの実装方法について解説します。

エレメント本体のファイルの実装

エレメント本体のクラスの実装

まずは、エレメント本体のファイルを実装していきます。
{VSCODE_HOME}/src/public/elements に、エレメント本体を実装するファイル（.tsファイル）を作成します。
ここでは、MySampleElement.ts というファイル名で作成しています。

このファイルに、IUIElementCore インタフェースを実装した、エレメントのクラスを定義していきます。

```
export class MySampleElement implements IUIElementCore {

  // ラッパークラスを返却します。
  // 繰り返さないシンプルなエレメントを作成する場合は `SimpleUIElement`、
  // 手動で繰り返すエレメントを作成する場合は `IndexableUIElement`、
  // 自動的に繰り返すエレメントを作成する場合は `RepeatableUIElement` を返却してください。
  public get wrapperClass(): UIElementWrapperClass {
    return 'SimpleUIElement';
  }
```

```
// エレメント名を返却します。
// { エレメントのクラス名 } .name のように、エレメントのクラス名を返却してください。
// エレメント名が他のエレメント名と重複すると、正常に動作しない恐れがあります。
public get elementTypeName(): string {
    return MySampleElement.name;
}

// エレメントの表示形式を返却します。
public get displayType(): UIElementDisplayType {
    // TODO: 後述します
}

// エレメントを非表示状態にするときの非表示方法を返却します。
// 実装がない場合は REMOVE_NODE_TREE を返却した場合と同じ動作です。
public get hiddenType(): UIElementHiddenType {
    // TODO: 後述します
}

// エレメントの配置制約を定義したクラスのクラス名を返却します。
public get constraintClass(): UIComponentConstraintClass {
    // TODO: 後述します
}

// エレメント固有プロパティの定義を返却します。
public get uniquePropertyDefinition(): IUniquePropertyDefinition {
    // TODO: 後述します
}

// 共通プロパティの定義を返却します。
public get commonPropertyDefinition(): ICommonPropertyDefinition {
    return {};
}

// createChildren(), updateChildren(), cloneChildren() で
// 子エレメントとして自身以外のエレメントを使用する場合に、そのクラス名を返却します。
// ここで返却されたエレメントのクラスは、IM-BloomMaker 側によって自動的にクラスが登録されるため、
// 上記の3メソッドの中で使用できます。
// 子エレメントを実装する必要がない場合は、空の配列を返却します。
public get dependElements(): UIElementCoreClass[] {
    return [];
}

// エレメントが作成された際に一度だけ呼び出されるメソッドです。
// このメソッド内で、このエレメントで使用するルートとなる HTML エレメントを、HTMLElementBuilder を経由して返却します。
// updateElement() によって都度変更されない常時適用する属性値やイベントの設定は、このメソッドに実装します。
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    // TODO: 後述します
}

// レンダリングが必要になった際に呼び出されるメソッドです。
// プロパティで紐づけている変数の値が変更された際など、エレメント側を更新する必要がある場合は、このメソッドに実装します。
public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // TODO: 後述します
}

// レンダリングが行われた後に、レンダリングが不要になった際に呼び出されるメソッドです。
// document.body や window などグローバルに対してのイベントの設定を解除する処理を、このメソッドに実装します。
// 特に実装がない場合は、このメソッドの実装は不要です。
public hiddenElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // TODO: 後述します
}

// エレメントが削除された際に呼び出されるメソッドです。
```

```
    // エレメントが削除された際に呼び出されるメソッドです。
    // 外部ライブラリを使用しており、ライブラリの解放処理が必要な場合に、このメソッドに実装します。
    // そのため、エレメントの実装のみで表示処理が完結している場合は、このメソッドの実装は不要です。
    public destroyElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): void {
        return;
    }

    // エレメントが作成された際に一度だけ呼び出されるメソッドです。 createElement() の後に呼び出されます。
    // 自身のエレメント配下に子エレメントを作成する必要がある場合、このメソッドでエレメントを作成し、配列として返却します。
    // 子エレメントが不要な場合は、空配列を返却します。
    // IHTMLElementBuilder.appendChild() で子要素を追加する場合と異なり、
    // デザイナ上でコンテナに配置した際に、子要素はそれぞれ別のエレメントとして扱われます。
    public createChildren(self: IUIElement, container: IUIContainer): IUIElement[] {
        return [];
    }

    // レンダリングが必要になった際に呼び出されるメソッドです。 updateElement() の後に呼び出されます。
    // プロパティの状態によって子エレメントの数に変更がある場合、新しい子エレメントの配列を返却することで、子エレメントを再配置します。
    // 子エレメントの数に変更がない場合でも、子エレメントのプロパティ値を変更する場合は、このメソッドに実装します。
    // createChildren() で空配列を返却しており、子エレメントを追加する予定がない場合は、このメソッドは実装しません。
    public updateChildren(
        children: IUIElement[],
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return children;
    }

    // エレメントが複製された際に呼び出されるメソッドです。 createElement() の後に呼び出されます。
    // children 引数には、同様に複製されたエレメントが格納されています。
    // プロパティの状態によって子エレメントの数に変更がある場合、新しい子エレメントの配列を返却することで、子エレメントを再配置します。
    // 子エレメントの数に変更がない場合でも、子エレメントのプロパティ値を変更する場合は、このメソッドに実装します。
    // 子エレメントに対して何も操作しない場合は、children 引数をそのまま返却します。
    public clonedChildren(
        children: IUIElement[],
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return children;
    }
}
```

制約を実装する

エレメントには制約を実装する必要があります。 制約とは、具体的にはエレメントの移動・削除・複製等が可能であるかどうかの定義です。

以下の手順で実装します。

1. エレメント本体のクラスの外で、 IUIComponentConstraint を実装した制約クラスを定義します。

   ここで定義した制約クラスをエレメント本体のクラスで利用します。特に理由がなければ、エレメント本体のクラスと同じファイルに記述することを推奨します。
   制約クラスを別ファイルに分けて定義する場合は、エレメント本体のクラスのファイルで制約クラスをimportして利用します。

   制約クラス内では、以下の6つのメソッドを実装します。いずれのメソッドも、真偽値（`true` か `false`）を返却します。

制約の内容	対応するプロパティ/メソッド
エレメントの移動	movable()
エレメントの削除	removable()
エレメントの複製	copyable()
子エレメントの配置	extremity() acceptableChild()
親エレメントの配置	acceptableParent()

各メソッドの実装方法については、以下に示す実装例を参考にしてください。

```
// 制約クラスで IUIComponentConstraint を実装します。
class Constraint implements IUIComponentConstraint {

    // エレメントの移動可否を指定するメソッドです。
    // true を返却する場合、デザイナ画面上に配置したエレメントを移動できます。
    public get movable(): boolean {
        return true;
    }

    // エレメントの削除可否を指定するメソッドです。
    // true を返却する場合、デザイナ画面上に配置したエレメントを削除できます。
    public get removable(): boolean {
        return true;
    }

    // エレメントの複製可否を指定するメソッドです。
    // true を返却する場合、デザイナ画面上に配置したエレメントを複製できます。
    public get copyable(): boolean {
        return true;
    }

    // エレメントが末端かどうかを指定するメソッドです。
    // true を返却する場合、デザイナ画面上に配置したエレメントの配下に子エレメントは配置できません。
    // このメソッドの返却値に true を指定した場合は、acceptableChild() の返却値は常に false とします。
    public get extremity(): boolean {
        return true;
    }

    // 親要素に対して、自身の配置可否を指定するメソッドです。
    // true を返却した場合、デザイナ画面上に配置したエレメントを親として、自身を子エレメントに配置できます。
    // parent 引数には親エレメントが渡されますので、親エレメントによって配置可能とするかどうか指定できます。
    public acceptableParent(parent: ParentUIElement): boolean {
        return true;
    }

    // 自身に対して、子エレメントの配置可否を指定するメソッドです。
    // true を返却した場合、デザイナ画面上に配置した自身のエレメントに対して、子エレメントを配置できます。
    // child 引数には子エレメントが渡されますので、子エレメントによって配置可能とするかどうか指定できます。
    // 常に false を返却する場合は、extremity() の返却値は常に true とします。
    public acceptableChild(child: IUIElement): boolean {
        return false;
    }
}
```

2. エレメント本体のクラス内の constraintClass メソッドで返却するように実装します。

   エレメント本体のクラス内では、以下のように constraintClass メソッドを実装します。

```
// 制約クラスのクラス名を返却します。
public get constraintClass(): UIComponentConstraintClass {
    return Constraint;
}
```

エレメントの表示形式を実装する

エレメントの表示形式を実装する必要があります。
表示形式とは、実行画面上でエレメントがどのように配置・表示されるかの定義です。

エレメント本体のクラス内に、 displayType メソッドを実装し、設定したい表示形式を文字列で指定します。

displayType メソッドの実装例を示します。以下は、表示形式に `INPUT` を設定する場合です。

```
public get displayType(): UIElementDisplayType {
    return 'INPUT';
}
```

> コラム

指定できる表示形式は、以下の通りです。

- **BLOCK**
  ブロック要素。
  コンテナページの横いっぱいに幅を取って配置されるため、デザイナ画面上に連続でブロック要素のエレメントを配置した場合、縦並びに要素が配置されます。
  見出しレベルエレメントやリストエレメントなどで使用されています。
- **INLINE**
  インラインブロック要素。
  連続でデザイナ画面上にエレメントを配置した場合、横並びに要素が配置されます。
  ラベルエレメントや画像埋め込みエレメントなどで使用されています。
- **INLINE_FLEX**
  インラインフレックス要素。
  一定の間隔で縦や横に子エレメントを配置させたいインライン要素に指定します。
  インラインフレックスエレメントで使用されています。
- **INPUT**
  インライン入力要素。
  横並びに要素が配置されます。
  INLINE とほぼ同様ですが、入力系エレメントの場合は INPUT を選択してください。
  テキスト入力エレメントやボタンエレメントなどで使用されています。
- **SEPARATOR**
  セパレータ要素。
  水平の横線を引く際に利用される <hr> タグ専用の要素です。
  水平罫線エレメントで使用されています。
- **TABLE**
  テーブル要素。
  テーブルのエレメントを形成する <table> タグ専用の要素です。
  テーブルエレメントで使用されています。
- **TABLE_CELL**
  テーブルセル要素。
  テーブルのセル部分を形成する <th>, <td> タグ専用の要素です。
  テーブルエレメントのヘッダ部分やそれ以外のセル部分を構成するエレメントで使用されています。
- **FLEX_BOX**
  フレックス要素。
  一定の間隔で縦や横に子エレメントを配置させたり、BLOCK や INLINE など、どちらか一方向のみ配置可能である子エレメントの並び方をレイアウトしたいブロック要素に指定します。
  フレックスコンテナエレメントで使用されています。
- **FLEX_ITEM**
  フレックスアイテム要素。
  FLEX_BOX のエレメントを親にもち、テーブルセル要素に似た動作をさせたいブロック要素に指定します。
  フレックスコンテナの子エレメントで使用されています。
- **FIXED**
  無デザイン要素。
  デザイナ上で選択されたくない要素に指定します。
  この要素を指定したエレメントはデザイナ上での移動・削除やエレメントの固有プロパティの変更ができません。
  リッチテキストボックスエレメントで使用されています。
- **SELECTABLE_FIXED**
  選択可能な無デザイン要素。
  FIXED と似ていますが、FIXED の機能のうち、選択のみ可能になった要素です。
- **FUNCTIONAL**
  機能的な無デザイン要素。
  デザイナ上でのみ表示させ、プレビュー画面や実行画面上では表示させたくないエレメントに指定します。
  タイマーエレメントで使用されています。
- **CANVAS**
  デザイナ上と実行画面上で表示方法が異なるインライン要素。
  デザイナ上ではイメージ図を表示させ、プレビュー画面や実行画面では実際の動作を行うエレメントに指定します。
  IM共通マスタ系のエレメントで使用されています。
- **PURE**
  IM-BloomMaker による表示制御を行わない要素。
  エレメントの表示デザインを IM-BloomMaker によって変更されたくない場合に指定します。

Bulma 系のエレメントで使用されています。

エレメントの非表示方法を実装する

エレメントが非表示状態になったとき、どのように非表示にするか方法を指定できます。
非表示方法とは、実行画面上でエレメントをどのように隠すかどうかの定義です。

エレメント本体のクラス内に、 hiddenType メソッドを実装し、設定したい非表示方法を文字列で指定します。
実装しない場合は `REMOVE_NODE_TREE` を設定した場合と同一です。

hiddenType メソッドの実装例を示します。以下は、非表示方法に `DISPLAY_CSS` を設定する場合です。

```
public get hiddenType(): UIElementHiddenType {
  return 'DISPLAY_CSS';
}
```

コラム

指定できる非表示方法は、以下の通りです。

- **REMOVE_NODE_TREE**
  ノードツリーから削除することで非表示にします。
  ノードツリーから削除するため、document.querySelector などでタグの探索ができません。
- **DISPLAY_CSS**
  ノードツリーに HTML タグを残したまま、display CSS に none を指定することで非表示にします。
  ノードツリーからは削除しないため、document.querySelector などでタグの探索が可能ですが、ブラウザの動作が遅くなる可能性があります。

createElement メソッドを実装する

エレメント本体の createElement メソッド内でエレメントのビルダを作成して返却します。

例として、`<input>` タグだけのエレメントを作成する場合の実装を以下に示します。

```
public createElement(
  container: IUIContainer,
  properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
  // エレメントのビルダを作成するには、IHTMLElementBuilder.createElement() を利用します。
  const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);

  // 返却値には作成したビルダを指定します。
  return builder;
}
```

コラム

エレメントのビルダを作成する方法について

createElement() の返却値は、純粋な HTML エレメント（HTMLElement 型）ではなく、エレメントのビルダ（IHTMLElementBuilder 型）にする必要があります。
子要素を追加する場合、追加される子要素に関しても同様です。（参考： *子エレメントを追加する* ）
ビルダを作成するには、上の例のように IHTMLElementBuilder.createElement() でタグ名からビルダを作成するか、IHTMLElementBuilder.fromElement() を利用して純粋な HTML のエレメントをビルダ化してください。
fromElement() の使用例は、「*固有プロパティを追加する*」を参照してください。

ここまでの実装例に沿って実装することで、`<input>` タグだけのエレメントを作成できます。

スタイル（CSS）を設定する

IM-BloomMaker では、デザイナ画面から各エレメントにスタイル（CSS）を適用することが可能です。
エレメントを作成する場合は、あらかじめスタイルが適用されているエレメントを作成することも可能です。

スタイルが適用されたエレメントを作成するには、createElement メソッド内で、 setCSS メソッドを利用してスタイルを追加します。

例えば、「*createElement メソッドを実装する*」の例に追加で padding: 10px を適用する場合、実装は以下の通りです。

```
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);

    // setCSS() を使用すると、作成したエレメントにstyleを追加できます。
    builder.setCSS('padding', '10px');

    return builder;
}
```

属性を設定する

エレメントを作成する際は、任意の属性を付与することも可能です。

createElement メソッド内で、setAttribute メソッドを利用して属性を追加します。

例えば、「*createElement メソッドを実装する*」の例に追加で name 属性に sample-element という値を付与する場合、実装は以下の通りです。

```
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);

    // setAttribute() を使用すると、作成したエレメントにstyleを追加できます。
    builder.setAttribute('name', 'sample-element');

    return builder;
}
```

独自の属性を付与することも可能です。

```
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);

    // 独自の属性名は、最初に data- を付与します。
    builder.setAttribute('data-my-attribute', 'my-attribute-value');

    return builder;
}
```

固有プロパティを追加する

エレメントのプロパティには、固有プロパティと共通プロパティがあります。

共通プロパティは、すべてのエレメントが持っているプロパティで、「ID」「表示/非表示」「ツールチップ」の3項目です。
エレメントを作成する際に実装する必要はなく、自動的に上の3項目が付与されます。

固有プロパティは、エレメント毎に異なるプロパティで、入力系エレメントであれば入力値、繰り返し系エレメントであれば繰り返しの回数などを持っています。
固有プロパティを付与したい場合は、エレメントのファイル内で実装する必要があります。

以下の手順で実装します。

1. PropertyDefinition 型を定義し、その中で、必要なプロパティを UniquePropertyDefinitionType 型で宣言します。

   例えば sampleValue というプロパティが必要な場合、まずは以下のように宣言します。

```
type PropertyDefinition = {
    sampleValue: UniquePropertyDefinitionType;
};
```

2. uniquePropertyDefinition という変数を作成し、UniquePropertyDefinitionType に従ってプロパティの定義を記述します。

   上に続いて、以下のように sampleValue プロパティの定義を記述します。

```
// エレメント固有カテゴリのプロパティは uniquePropertyDefinition で設定できます。
const uniquePropertyDefinition: IUniquePropertyDefinition & PropertyDefinition = {
    sampleValue: {
        displayName: 'sampleValue',
        definition: {
            required: true,
            rerender: true,
        },
        type: 'string',
    },
};
```

3. エレメント本体のクラスに、uniquePropertyDefinition メソッドを実装します。メソッド内で、プロパティの定義を宣言した変数を返却します。

   以下のように実装します。

```
public get uniquePropertyDefinition(): IUniquePropertyDefinition {
    return uniquePropertyDefinition;
}
```

4. エレメント本体のクラスの createElement メソッド内に、エレメントに値が入力されたときにプロパティの sampleValue を更新する処理を記述します。

   以下のように実装します。

```
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
    ): IHTMLElementBuilder {

        // まず、<input>タグを作成します。
        const inputElement = document.createElement('input');

        // <input>タグに入力が行われたときに、sampleValueプロパティを更新するようにイベントリスナを付与します。
        inputElement.addEventListener('input', (e) => {
            properties.setProperty('sampleValue', inputElement.value, false);
        });

        // IHTMLElementBuilder.fromElement() を利用して、作成したエレメントからビルダを生成します。
        const builder = window.imHichee.HTMLElementBuilder.fromElement(inputElement);

        return builder;
    }
```

5. エレメント本体のクラスの updateElement メソッド内に、sampleValue が変更されたときにエレメント側の値を更新する処理を記述します。

   以下のように実装します。

```
public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
    ): void {
        // プロパティの値を取得するには、properties.getProperty() でプロパティ名を文字列で指定します。
        const sampleValue = properties.getProperty('sampleValue').toString();

        // builder.setValue() で、実際に描画される<input>タグのvalue属性に、sampleElementの値を反映します。
        builder.setValue(sampleValue);
    }
```

コラム

uniquePropertyDefinition で記述できるプロパティの定義について

UniquePropertyDefinitionType の実装は以下の通りです。
uniquePropertyDefinition では、この形式に従って、必要な定義を記述してください。
? がついている定義は省略可能です。

```
type UniquePropertyDefinitionType = {
  /** 表示名 */
  displayName: string;

  /** 省略可能な定義 */
  definition: {
    /** バリデーションルール - 必須（未指定の場合：`false`） */
    required?: boolean;

    /** バリデーションルール - データ種別（未指定の場合：`ANY`） */
    dataType?: UIComponentPropertyDataType;

    /** バリデーションルール - 数値の最小値 */
    min?: number;

    /** バリデーションルール - 数値の最大値 */
    max?: number;

    /** 読み取り専用（未指定の場合：`false`） */
    readonly?: boolean;

    /** 値変更時、再レンダリング（未指定の場合：`false`） */
    rerender?: boolean;

    /** エレメント繰り返し使用可否（未指定の場合：`false`） */
    repeatable?: boolean;

    /** プレビュー時バリデーション実行（未指定の場合：`false`） */
    validateValue?: boolean;

    /** 固定値許可（未指定の場合：`true`） */
    acceptStatic?: boolean;

    /** 変数値許可（未指定の場合：`true`） */
    acceptDynamic?: boolean;

    /** 複数行文字列の入力許可（未指定の場合：`false`） */
    acceptMultipleLines?: boolean;

    /** 深い階層にある値の変更を検知（未指定の場合：`false`） */
    deepObserve?: boolean;

    /** 変数値で指定可能な型 */
    allowedVariableTypes?: UIComponentPropertyVariableType[];

    /** テキストの直接編集（レイアウトモードのみ有効、未指定の場合：`none`） */
    directInput?: UIComponentPropertyDirectInputType;

    /** アイコンの直接編集（レイアウトモードのみ有効、未指定の場合：`none`） */
    iconClass?: UIComponentPropertyIconClassType;

    /** 値の選択候補（セレクトボックスの場合などに指定） */
    candidate?: {
      /** 表示名 */
      displayName: string;

      /** 値 */
      value: UIComponentPropertyValueType;
    }[];
  };

  /** プロパティタイプ名 */
  type: UIComponentPropertyType;

  /** プロパティ値 */
  value?: {};

  /** カテゴリID */
  categoryId?: string;
};
```

コラム

双方向バインディングについて

IM-BloomMaker の標準で用意されている入力系エレメントは、プロパティに変数を紐付けた際、双方向バインディングされるように実装されています。
双方向バインディングとは、エレメントのプロパティとそれに結びついている変数が、互いに更新し合って同期されるような結びつきの仕組みです。
具体的には以下のように動作します。

- エレメントの入力値が変更された際に、その入力値のプロパティと結びついている変数の値が変更される。
- アクション等、エレメントへの入力以外で変数の値が変更されると、エレメントのプロパティが変更される。

これを独自のエレメントで実現するためには、変数の値が変更されたときに IM-BloomMaker からの再レンダリング要求を送信するように設定が必要です。
再レンダリング要求は、標準では送信されません。以下の rerender および deepObserve プロパティを使用して、再レンダリング要求が送信されるようにしてください。

rerender について

rerender を `true` に設定したプロパティに変数値を指定した場合、その変数値に変化があると updateElement, updateChildren メソッドが呼び出されます。
この仕組みを利用して、変数値が変化したときにエレメントを再レンダリングすると、双方向バインディングのうち、変数→エレメントの流れを実現できます。

deepObserve について

deepObserve を `true` に設定したプロパティに変数値を指定した場合、その変数値が配列やマップ型のときに配列・マップ配下も含めた値の変化も検知します。
rerender と共に設定することで、複数の変数値を同時に使用するようなエレメント（表、リスト、ツリー等）の再レンダリングを支援します。

グローバルなイベントを登録する

document.body や window など、エレメントの範疇を超えてイベントを設定する必要がある場合、updateElement メソッドでイベントを登録し、hiddenElement で解除します。
このように実装することで、エレメントがレンダリングされたときにイベントが登録され、エレメントが非表示状態になったときにイベントが解除されます。

```
type ResizeEventCallback = (this: Window, e: Event) => void;
private _attachedResizeCallback: ResizeEventCallback | null;

constructor() {
    this._attachedResizeCallback = null;
}

public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // window に対して resize イベントを登録します。
    // 重複登録にならないよう、登録済みのコールバックの存在を確認します。
    if (!this._attachedResizeCallback) {
        const callback = this.resizeWindow.bind(this);
        this._attachedResizeCallback = callback;
        window.addEventListener('resize', callback);
    }
}

public hiddenElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // window に対して resize イベントを解除します。
    if (this._attachedResizeCallback) {
        window.removeEventListener('resize', this._attachedResizeCallback);
        this._attachedResizeCallback = null;
    }
}

private resizeWindow(): void {
    // ウィンドウがリサイズした際の処理を実装します。
}
```

コラム

updateElement メソッド内でのイベント登録について

上の例のように、updateElement メソッド内でのイベント登録時は、すでにイベントが登録されているかどうかを確認してください。
エレメントが非表示状態に変更された場合は hiddenElement メソッドが1度だけ呼ばれますが、エレメントが画面上に表示されている場合は変数値の変更などによって、複数回呼ばれる可能性があるためです。
本来不要なイベントが複数登録されてしまうと、画面表示が遅くなるなどのパフォーマンスに影響します。

子エレメントを追加する

1つ、ないし複数の子エレメントを持ったエレメントを作成したい場合には、createElement メソッド内で IHTMLElementBuilder.appendChild() を利用して、1つのエレメントの配下に他のエレメントを追加することが可能です。
プレビュー画面・実行画面で動作する際に実際に描画される HTML では、親要素の中に子要素が配置されます。

「*createElement メソッドを実装する*」の実装例では、`<input>` タグだけのエレメントを作成していましたが、これを子要素にもつ `<div>` タグのエレメントを作成する場合、createElement メソッドの実装例は以下です。

```
export class MySampleElement implements IUIElementCore {

    // 子要素のビルダを、あらかじめメソッドの外で宣言しておきます。理由は後述します。
    private _inputElementBuilder: IHTMLElementBuilder;
```

```
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {

    const inputElement = document.createElement('input');
    inputElement.addEventListener('input', (e) => {
        properties.setProperty('value', inputElement.value, false);
    });

    // ビルダを生成し、あらかじめ宣言している変数に代入する
    this._inputElementBuilder = window.imHichee.HTMLElementBuilder.fromElement(inputElement);

    // <div> のエレメントビルダを生成
    const builder = window.imHichee.HTMLElementBuilder.createElement('div', HTMLDivElement);

    // 子要素を追加
    builder.appendChild(this._inputElementBuilder);

    return builder;
}
```

あわせて、updateElement メソッドの実装も変更する必要があります。
ここでは、sampleValue プロパティの値が変更された際に、子要素の `<input>` タグの `value` 属性の値も変更されるように、処理を実装します。

```
public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // プロパティの値を文字列で取得します。
    const sampleValue = properties.getProperty('sampleValue').toString();

    // 子要素のエレメントに値をセットします。
    // 子ノードの IHTMLElementBuilder は自動で build() が呼ばれない仕様であるため、独自で build() を実行します。
    this._inputElementBuilder.setValue(sampleValue).build();
}
```

コラム

updateElement メソッド内での各子要素へのアクセスについて

上の例のように、createElement メソッドと updateElement メソッドなど、複数のメソッド内でエレメントの子要素にアクセスする必要がある場合があります。
このような場合、上の例のように子要素のビルダをメソッドの外で変数としてあらかじめ宣言しておくことで、メソッド内で子要素のビルダそれぞれにアクセスすることが可能です。

コラム

エレメントの子要素について

appendChild メソッドを利用して子要素を追加した場合、IM-BloomMaker のデザイナ上で1つ1つの子要素を動かしたり、子要素のプロパティを設定したりすることはできません。
全体で1つのエレメントとして扱われます。
createChildren メソッドを利用して子要素を追加した場合は、デザイナ上で1つ1つの子要素を動かしたり、削除したり、子要素のプロパティを設定したりすることが可能です。

実装したクラスの登録

エレメント本体のクラスを登録する処理を `{VSCODE_HOME}/src/index.ts` に実装します。
カテゴリを新規に登録する場合、IUIElementRepository.registerCategory メソッドを利用します。

```
import {MySampleElement} from './public/elements/MySampleElement';

// エレメントのリポジトリは以下のように取得します。
const elementRepository = window.imHichee.UIElementRepository;

// エレメントのカテゴリを登録します。
// 新たにカテゴリを追加したい場合には IUIElementRepository.registerCategory() を使用して、カテゴリを追加してください。
// id にはカテゴリのID（任意）、sortNumber にはカテゴリを表示させる際の優先順位を指定してください。
// カテゴリは sortNumber を基準に昇順で表示されます。
elementRepository.registerCategory('programming-sample', {sortNumber: 3000});

// 作成したエレメントをカテゴリに登録します。
// categoryId には登録先のカテゴリ名、sortNumber はカテゴリ内でエレメントを表示させる際の優先順位を指定できます。
// sortNumber を基準に昇順で表示されます。
elementRepository.registerClass(MySampleElement, {
    categoryId: 'programming-sample',
    sortNumber: 0,
});

// メッセージプロパティを JSON 化したものが以下に置換され、メッセージとして登録されます。
// そのため、この記述は削除しないでください。
window.imHichee.MessageBuilder.register('6549e165-925d-4de2-8c52-ec3cda282ed2');
```

コラム

registerCategory の sortNumber について

elementRepository.registerCategory の引数に指定する sortNumber は、IM-BloomMaker が標準で提供するカテゴリのソート番号と共用です。
独自のエレメントカテゴリを指定する場合は、3000 以上の値を指定することを推奨しますが、以下の値の近くに設定することで、標準提供カテゴリの近くに独自カテゴリを表示できます。

標準提供カテゴリ名	sortNumber
レイアウト	110
繰り返し	210
フォーム部品	310
共通マスタ	360
汎用	410
パーツ	510
その他	2010

プロパティファイルの実装

新たに追加したカテゴリの名称、作成したエレメントの名称、ヘルプを画面上に表示するために、プロパティファイルを実装する必要があります。
必要な言語に応じて、{VSCODE_HOME}/src/public/messages 配下にプロパティファイルを作成します。
日中英の3ロケール（言語）に対応する場合は、以下の4ファイルが必要です。不要なロケールはプロパティファイルを作成する必要はありません。

- component_ja.properties
  日本語のプロパティファイルです。
- component_en.properties
  英語のプロパティファイルです。
- component_zh_CN.properties
  中国語のプロパティファイルです。
- component.properties
  対応するロケールのプロパティが欠落している場合に参照されるプロパティファイルです。

それぞれのプロパティは以下のように記述します。

- エレメントのカテゴリ名： CAP.Z.IWP.HICHEE.COMPONENT.CATEGORY.NAME.{index.ts内で指定したカテゴリID}={表示したいカテゴリ名}

- エレメントの名称： CAP.Z.IWP.HICHEE.COMPONENT.NAME.{エレメントのクラス名}={表示したいエレメント名}
- エレメントのヘルプ： CAP.Z.IWP.HICHEE.COMPONENT.DESCRIPTION.{エレメントのクラス名}={表示したいヘルプの内容}

日本語のプロパティファイルを記述する際の例を以下に示します。（先頭に#をつけて、コメントを記述することも可能です。）

```
# エレメント - カテゴリ名
CAP.Z.IWP.HICHEE.COMPONENT.CATEGORY.NAME.programming-sample=プログラミングガイドサンプル

# エレメント - 名称
CAP.Z.IWP.HICHEE.COMPONENT.NAME.MySampleElement=サンプルエレメント

# エレメント - ヘルプ
CAP.Z.IWP.HICHEE.COMPONENT.DESCRIPTION.MySampleElement=サンプルエレメントです。
```

コラム

プロパティファイル内に記述する日本語・中国語の Unicode エスケープについて

intra-mart Accel Platform の多くの製品では、Java の仕様上、プロパティファイルに記述する日本語や中国語の文字は *\u3042* のような **Unicode エスケープ形式** で記述する必要があります。
しかし、IM-BloomMaker でエレメントやアクションを作成する際には、そのようなエスケープ処理は必要ありません。
日本語や中国語をそのままプロパティファイルに記述してください。

注意

エレメントの命名について

エレメントのクラス名が（IM-BloomMaker 標準のエレメントも含めて）複数のエレメントで重複すると、正常に動作しないため、一意なクラス名を付けるようにしてください。

2020 Spring(Yorkshire) 時点の、既存のエレメントのクラス名の一覧は、「*付録) 2020 Spring(Yorkshire) 時点のエレメントのクラス名一覧*」を参照してください。
また、2020 Spring(Yorkshire) 以降のリリースで追加されるエレメントに関しては、すべてクラス名の先頭に *Im* が付与されます。
独自のエレメントを実装する際は、独自の接頭辞を付与するなどの方法で、クラス名の重複を回避してください。

エレメントの作成に必要な作業は以上です。
「*bundleの生成*」に戻って残りの作業を行うことで、実装したエレメントを利用することが可能です。

スーパークラスの利用（任意）

似た挙動をするエレメントを複数実装する場合は、実装を共通化することで効率的な開発が可能です。
エレメントのクラスでは IUIElementCore インタフェースを実装する必要がありますが、共通の実装はスーパークラスに実装しておくことで処理を共通化できます。
各エレメントのクラスでは、IUIElementCore インタフェースを実装する代わりに、そのスーパークラスを継承します。

例えば、以下のような条件を持つエレメントを実装する際に、共通で利用されるスーパークラスを考えてみます。

- インラインで表示するエレメント
  - → displayType() は INLINE を返却する
- 繰り返しエレメントでは無い
  - → wrapperClass() は SimpleUIElement を返却する
- 子エレメントは持たない
  - → createChildren(), clonedChildren() では何も処理を行わない
- その他メソッドでは、サブクラスで任意の実装を行う

このような場合のスーパークラスの実装例を以下に示します。

```
export abstract class SampleBaseElementCore implements IUIElementCore {

    public abstract get elementTypeName(): string;

    public get displayType(): UIElementDisplayType {
        return 'INLINE';
    }

    public abstract get constraintClass(): UIComponentConstraintClass;

    public get wrapperClass(): UIElementWrapperClass {
        return 'SimpleUIElement';
    }

    public abstract get uniquePropertyDefinition(): IUniquePropertyDefinition;

    public abstract get commonPropertyDefinition(): ICommonPropertyDefinition;

    public get dependElements(): UIElementCoreClass[] {
        return []; // NO DEPENDENCIES
    }

    public abstract createElement(
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IHTMLElementBuilder;

    public updateElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): void {
        // DO NOTHING
    }

    public destroyElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): void {
        // DO NOTHING
    }

    public createChildren(
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return []; // DO NOTHING
    }

    public clonedChildren(
        children: IUIElement[],
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return children; // DO NOTHING
    }
}
```

各エレメントのクラスでは以下のように継承して利用します。

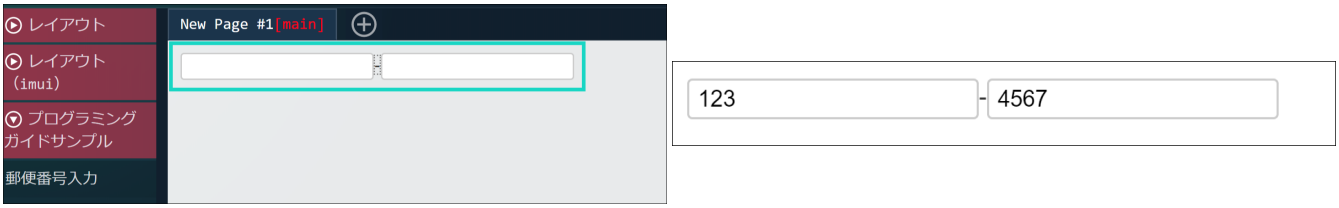
```
export class SampleElementClass extends SampleBaseElementCore { ... }
```

エレメントのサンプル実装

この章では、添付のサンプル（sample.zip の implemented ディレクトリ）の郵便番号入力エレメント（MyZipCodeField.ts）の実装について、説明します。

郵便番号入力エレメントは、以下のようなエレメントです。

- 全体の親要素(<*div*>)と、その子要素である2つのテキストボックス(<*input*>)と1つのラベル(<*label*>)から構成されています。
- 1つめのテキストボックスに3桁、2つめのテキストボックスに4桁まで値を入力できます。
- 全体でvalueという固有プロパティを持っています。valueの値は、2つのテキストボックスに入力された値を連結したものです。既存の多くのコンポーネントと同様に、双方向バインディングで実装します。（双方向バインディングについては、「*固有プロパティを追加する*」を参照してください。）



本体のクラスを実装する

まずは、 {VSCODE_HOME}/src/public/elements に、MyZipCodeField.ts を作成します。

MyZipCodeField クラスで IUIElementCore インタフェースを実装します。 // TODO の部分はこれから実装していきます。

```
export class MyZipCodeField implements IUIElementCore {

  // 繰り返しのないエレメントなので、SimpleUIElement を指定します。
  public get wrapperClass(): UIElementWrapperClass {
    return 'SimpleUIElement';
  }

  // エレメント名を返却します。
  // { エレメントのクラス名 } .name 形式で、エレメントのクラス名を返却します。
  public get elementTypeName(): string {
    return MyZipCodeField.name;
  }

  // エレメントの表示形式を返却します。
  public get displayType(): UIElementDisplayType {
    // TODO
  }

  // エレメントの配置制約を定義したファイルのクラス名を返却します。
  public get constraintClass(): UIComponentConstraintClass {
    // TODO
  }

  // エレメント固有プロパティの定義を返却します。
  // 固有プロパティがない場合は、空のオブジェクトを返却します。
  public get uniquePropertyDefinition(): IUniquePropertyDefinition {
    // TODO
  }

  // 共通プロパティの定義を返却します。
  // 固有プロパティがない場合は、空のオブジェクトを返却します。
  public get commonPropertyDefinition(): ICommonPropertyDefinition {
    return {};
  }

  // createChildren(), updateChildren(), cloneChildren() で
  // 子エレメントとして自身以外のエレメントを使用する場合に、そのクラス名を返却します。
  // このサンプルでは子エレメントを作成しませんので、空配列を返却します。
  public get dependElements(): UIElementCoreClass[] {
    return [];
  }

  // エレメント作成時に一度だけ呼び出されるメソッドです。
  // このメソッド内で、このエレメントで使用するルートとなる HTML エレメントを、HTMLElementBuilder を経由して返却します。
  // このサンプルでは、<div> タグやその子要素の <input> タグを作成する処理を記述していきます。
  public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
  ): IHTMLElementBuilder {
    // TODO
```

```
    }

    // レンダリングが必要になった際に呼び出されるメソッドです。
    // プロパティで紐づけている変数の値が変更された際など、エレメント側を更新する必要がある場合は、このメソッドに実装します。
    // このサンプルでは、プロパティである value が変更された際に、子要素の <input> タグに value の値を反映させる処理を記述していきます。
    public updateElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
    ): void {
        // TODO
    }

    // エレメントが作成された際に一度だけ呼び出されるメソッドです。 createElement() の後に呼び出されます。
    // 自身のエレメント配下に子エレメントを作成する必要がある場合、このメソッドでエレメントを作成し、配列として返却します。
    // このサンプルでは子エレメントを作成しませんので、空配列を返却します。
    public createChildren(self: IUIElement, container: IUIContainer): IUIElement[] {
        return [];
    }

    // エレメントが複製された際に呼び出されるメソッドです。 createElement() の後に呼び出されます。
    // このサンプルでは子エレメントを作成しませんので、children 引数をそのまま返却します。
    public clonedChildren(
        children: IUIElement[],
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return children;
    }

}
```

制約を実装する

制約クラス Constraint で IUIComponentConstraint を実装します。

ここでは、以下のように制約を定義します。

制約の内容	可否	対応するプロパティ/メソッド
エレメントの移動	可能	movable()
エレメントの削除	可能	removable()
エレメントの複製	可能	copyable()
子エレメントの配置	不可能	extremity() acceptableChild()
親エレメントの配置	可能	acceptableParent()

この場合の実装は以下です。

```
class Constraint implements IUIComponentConstraint {

   // エレメントの移動可否を指定するメソッドです。
   // このサンプルでは、エレメントは移動可能にするため、true を返却します。
   public get movable(): boolean {
      return true;
   }

   // エレメントの削除可否を指定するメソッドです。
   // このサンプルでは、エレメントは削除可能にするため、true を返却します。
   public get removable(): boolean {
      return true;
   }

   // エレメントの複製可否を指定するメソッドです。
   // このサンプルでは、エレメントは複製可能にするため、true を返却します。
   public get copyable(): boolean {
      return true;
   }

   // エレメントが末端かどうかを指定するメソッドです。
   // このサンプルでは、子エレメントを作成しませんので、true を返却します。
   public get extremity(): boolean {
      return true;
   }

   // 親要素に対して、自身の配置可否を指定するメソッドです。
   // このサンプルでは、どの親エレメントに対しても配置可能としますので、常に true を返却します。
   public acceptableParent(parent: ParentUIElement): boolean {
      return true;
   }

   // 自身に対して、子エレメントの配置可否を指定するメソッドです。
   // このサンプルでは、自身より子階層にエレメントを移動させたくないため、常に false を返却します。
   public acceptableChild(child: IUIElement): boolean {
      return false;
   }

}
```

MyZipCodeField クラスの constraintClass メソッドで、制約クラスを返却します。

```
// 上で実装した制約クラスのクラス名 Constraint を返却します。
public get constraintClass(): UIComponentConstraintClass {
   return Constraint;
}
```

表示形式を実装する

表示形式は INPUT を選択します。
MyZipCodeField クラスの displayType メソッドで、INPUT という文字列を返却します。

```
// エレメントの表示形式を返却します。
public get displayType(): UIElementDisplayType {
   return 'INPUT';
}
```

返却可能な文字列については、「*エレメントの表示形式を実装する*」を参照してください。

固有プロパティを実装する

固有プロパティ value と placeholder を実装します。
PropertyDefinition タイプを宣言し、value と placeholder プロパティの型を UniquePropertyDefinitionType として宣言します。
続いて、それを継承した uniquePropertyDefinition という変数を宣言し、その中で value と placeholder プロパティの定義を記述します。

```
type PropertyDefinition = {
    value: UniquePropertyDefinitionType;
    placeholder: UniquePropertyDefinitionType;
};

// エレメント固有カテゴリのプロパティは uniquePropertyDefinition で設定できます。
const uniquePropertyDefinition: IUniquePropertyDefinition & PropertyDefinition = {
    value: {
        displayName: 'value',
        definition: {
            // value プロパティには何らかの指定が必要である
            required: true,
            // value プロパティの値が変わった時に再レンダリングされる必要がある
            rerender: true
        },
        // 文字列型
        type: 'string',
        // デフォルト値は空文字
        value: ''
    },
    placeholder: {
        displayName: 'placeholder',
        definition: {
            // value プロパティは任意であり、値の指定がなくともよい
            required: false,
            // value プロパティの値が変わった時に再レンダリングされる必要がある
            rerender: true,
        },
        // 真偽値型
        type: 'boolean',
        // デフォルト値は false
        value: false
    }
};
```

MyZipCodeField クラスの uniquePropertyDefinition メソッドで、value プロパティを定義した変数 uniquePropertyDefinition を返却します。

```
public get uniquePropertyDefinition(): IUniquePropertyDefinition {
    return uniquePropertyDefinition;
}
```

createElement メソッドを実装する

createElement メソッドを実装します。

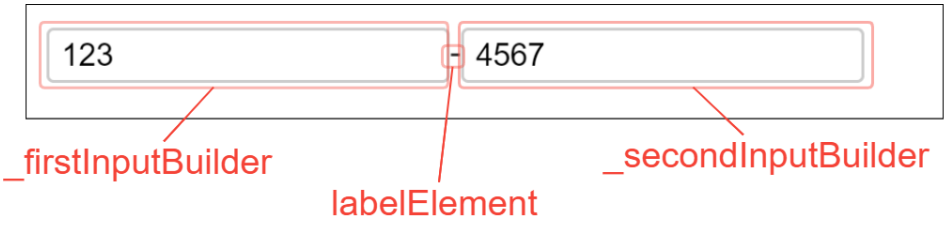
以下の処理を記述します。

- タグの下に <input> タグと <label> タグが配置されるようにする。
- 子要素の <input> タグに入力があった際、親要素の value プロパティを更新する。
- setAttribute メソッドを使用して、属性を付与する。
- setCSS メソッドを使用して、CSS を指定する。

実装は以下の通りです。

```
export class MyZipCodeField implements IUIElementCore {

    // 子要素のビルダは updateElement メソッド内からもアクセスされるため、クラスの先頭で宣言しておきます。
    private _firstInputBuilder: IHTMLElementBuilder;
    private _secondInputBuilder: IHTMLElementBuilder;
```

```
// エレメント作成時に一度だけ呼び出されるメソッドです。
// このメソッド内でエレメントを作成して返却します。
// ここでは、子要素の作成、イベントリスナ、属性、スタイルの付与を行っています。
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    // エレメントを作成します。
    const builder = window.imHichee.HTMLElementBuilder.createElement('div', HTMLDivElement);

    // 子タグのテキストボックスを作成します。
    const inputElement1 = document.createElement('input');
    const inputElement2 = document.createElement('input');

    // テキストボックスの input イベントが呼ばれると、value プロパティを更新するようにリスナを付与します。
    inputElement1.addEventListener('input', (e) => {
        properties.setProperty('value', inputElement1.value + inputElement2.value, false);
    });
    inputElement2.addEventListener('input', (e) => {
        properties.setProperty('value', inputElement1.value + inputElement2.value, false);
    });

    this._firstInputBuilder = window.imHichee.HTMLElementBuilder.fromElement(inputElement1);
    this._secondInputBuilder = window.imHichee.HTMLElementBuilder.fromElement(inputElement2);
    const labelElement = window.imHichee.HTMLElementBuilder.createElement('label', HTMLLabelElement).setText('-');

    // setAttribute() を使用すると、作成したエレメントに属性を追加できます。
    builder.setAttribute('my-attribute', 'zip-code-input');

    // setCSS() を使用すると、作成したエレメントにstyleを追加できます。
    builder.setCSS('padding', '10px');

    // appendChild() を利用して、上で作成したエレメント（ビルダ）を子エレメントとして追加します。
    // ここで追加した子エレメントは、IM-BloomMakerのデザイナ上で1つ1つの要素を動かしたりプロパティを設定したりすることはできません。全体で1つのエレメントとして扱われます。
    builder
        .appendChild(this._firstInputBuilder)
        .appendChild(labelElement)
        .appendChild(this._secondInputBuilder);

    // 返却値には作成したエレメントを指定します。
    return builder;
}
```



updateElement メソッドを実装する

続いて、updateElement メソッドを実装します。
「*固有プロパティを実装する*」で value と placeholder プロパティに `rerender: true` を設定しているため、value または placeholder プロパティの値が変更されるたびにこのメソッドが呼び出されます。

以下の処理を記述します。

- value プロパティの値を取得し、先頭の3桁とそれ以降に分割する。
  それぞれの値を子要素のテキストボックスにそれぞれセットする。
- placeholder プロパティの値を取得し、true であればテキストボックスにプレースホルダを表示する。
  false であればプレースホルダを消去する。

実装は以下の通りです。

```
// レンダリングが必要になった際に呼び出されるメソッドです。
// プロパティで紐づけている変数の値が変更された際に、エレメント側を更新する必要がある場合などにその処理を記述します。
public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // 各プロパティの値を取得します。
    const value = properties.getProperty('value').toString();
    const placeholder = properties.getProperty('placeholder').toBoolean();

    // 取得したプロパティの値を2つの入力エレメントに反映するため、前3桁と残りに分割します。
    const value1 = value.substring(0, 3);
    const value2 = value.substring(3);

    // 子ノードの <input> タグの value プロパティと placeholder 属性に値をセットします。
    // 子ノードの IHTMLElementBuilder は自動で build() が呼ばれない仕様であるため、独自で build() を実行する必要があります。
    this._firstInputBuilder.setValue(value1).setAttribute('placeholder', placeholder ? '000' : '').build();
    this._secondInputBuilder.setValue(value2).setAttribute('placeholder', placeholder ? '0000' : '').build();
}
```

実装したクラスの登録

{VSCODE_HOME}/src/index.ts で以下のように、カテゴリとエレメントを登録します。

ここでは、カテゴリIDは programming-sample としています。

```
import {MyZipCodeField} from './public/elements/MyZipCodeField';

// エレメントのリポジトリは以下のように取得します。
const elementRepository = window.imHichee.UIElementRepository;

// エレメントのカテゴリを登録します。
// 新たにカテゴリを追加したい場合には IUIElementRepository.registerCategory() を使用して、カテゴリを追加してください。
// id にはカテゴリの ID 、sortNumber にはカテゴリを表示させる際の優先順位を指定してください。
// カテゴリは sortNumber を基準に昇順で表示されます。
elementRepository.registerCategory('programming-sample', {sortNumber: 3000});

// 作成したエレメントをカテゴリに登録します。
// categoryId には登録先のカテゴリ名、sortNumber はカテゴリ内でエレメントを表示させる際の優先順位を指定できます。
// sortNumber を基準に昇順で表示されます。
elementRepository.registerClass(MyZipCodeField, {
    categoryId: 'programming-sample',
    sortNumber: 0,
});

// メッセージプロパティを JSON 化したものが以下に置換され、メッセージとして登録されます。
// そのため、この記述は削除しないでください。
window.imHichee.MessageBuilder.register('6549e165-925d-4de2-8c52-ec3cda282ed2');
```

プロパティファイルの実装

{VSCODE_HOME}/src/public/messages に component_ja.properties を作成し、以下のようにメッセージプロパティを記述します。

```
# エレメント - カテゴリ
CAP.Z.IWP.HICHEE.COMPONENT.CATEGORY.NAME.programming-sample=プログラミングガイドサンプル

# エレメント - 名称
CAP.Z.IWP.HICHEE.COMPONENT.NAME.MyZipCodeField=郵便番号入力

# エレメント - ヘルプ
CAP.Z.IWP.HICHEE.COMPONENT.DESCRIPTION.MyZipCodeField=郵便番号を入力するためのサンプルエレメントです。
```

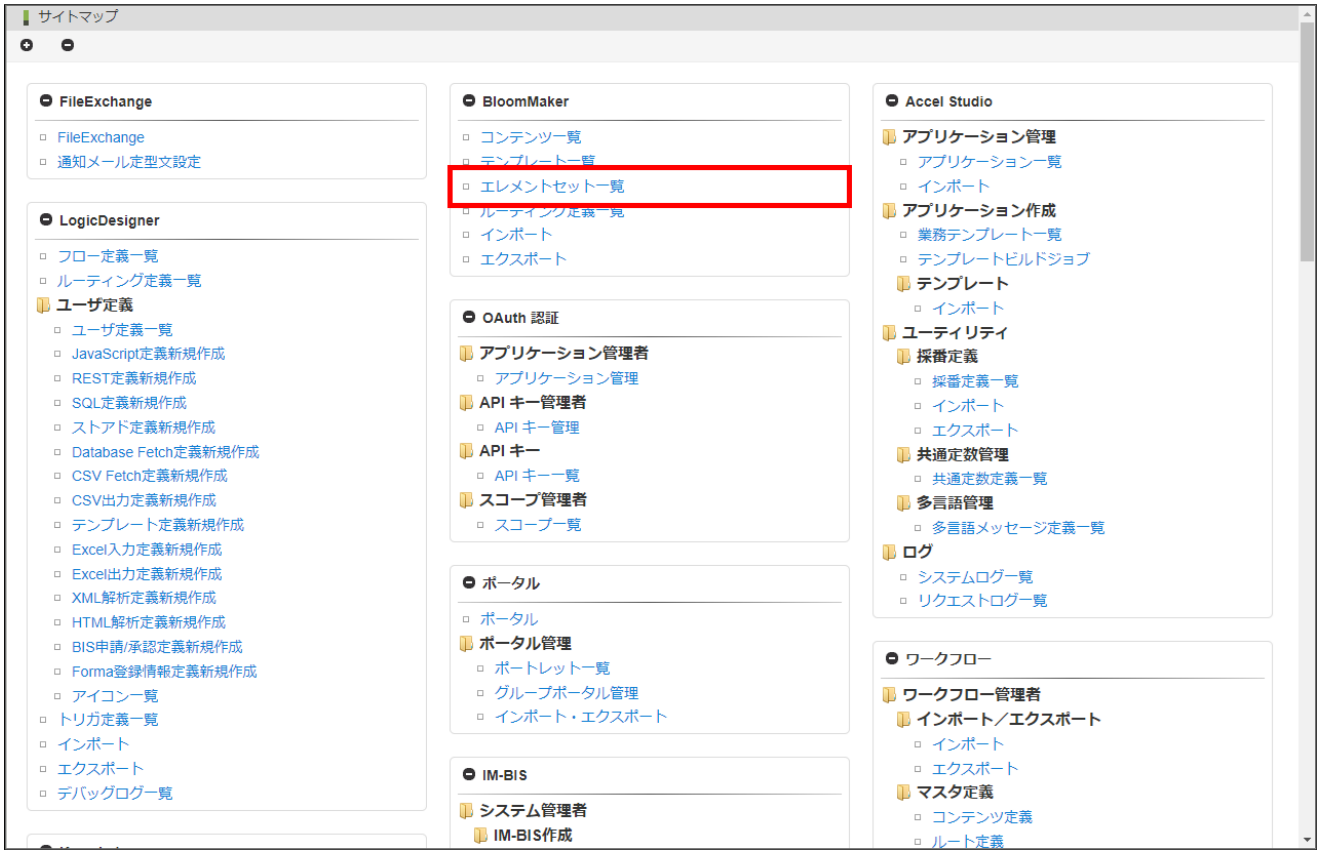
レイアウトモードで使用可能にする

独自で作成したエレメントをレイアウトモードで使用するためには、ユーザ部品としてエレメントセットを登録します。

コラム

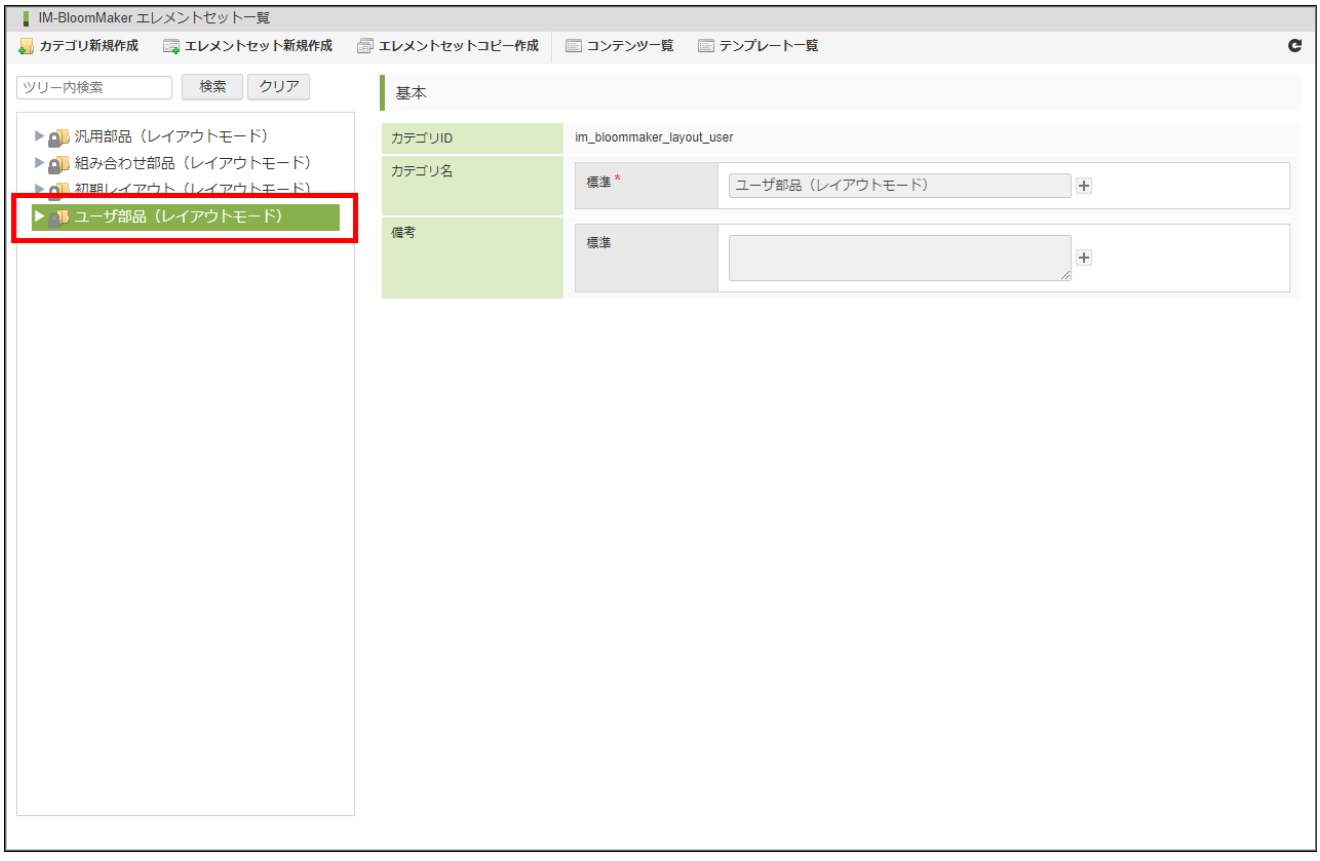
サンプルプロジェクトをビルドした後、アプリケーションサーバにデプロイを行ってください。

1. 「サイトマップ」→「BloomMaker」→「エレメントセット一覧」から、「エレメントセット一覧」画面を開きます。



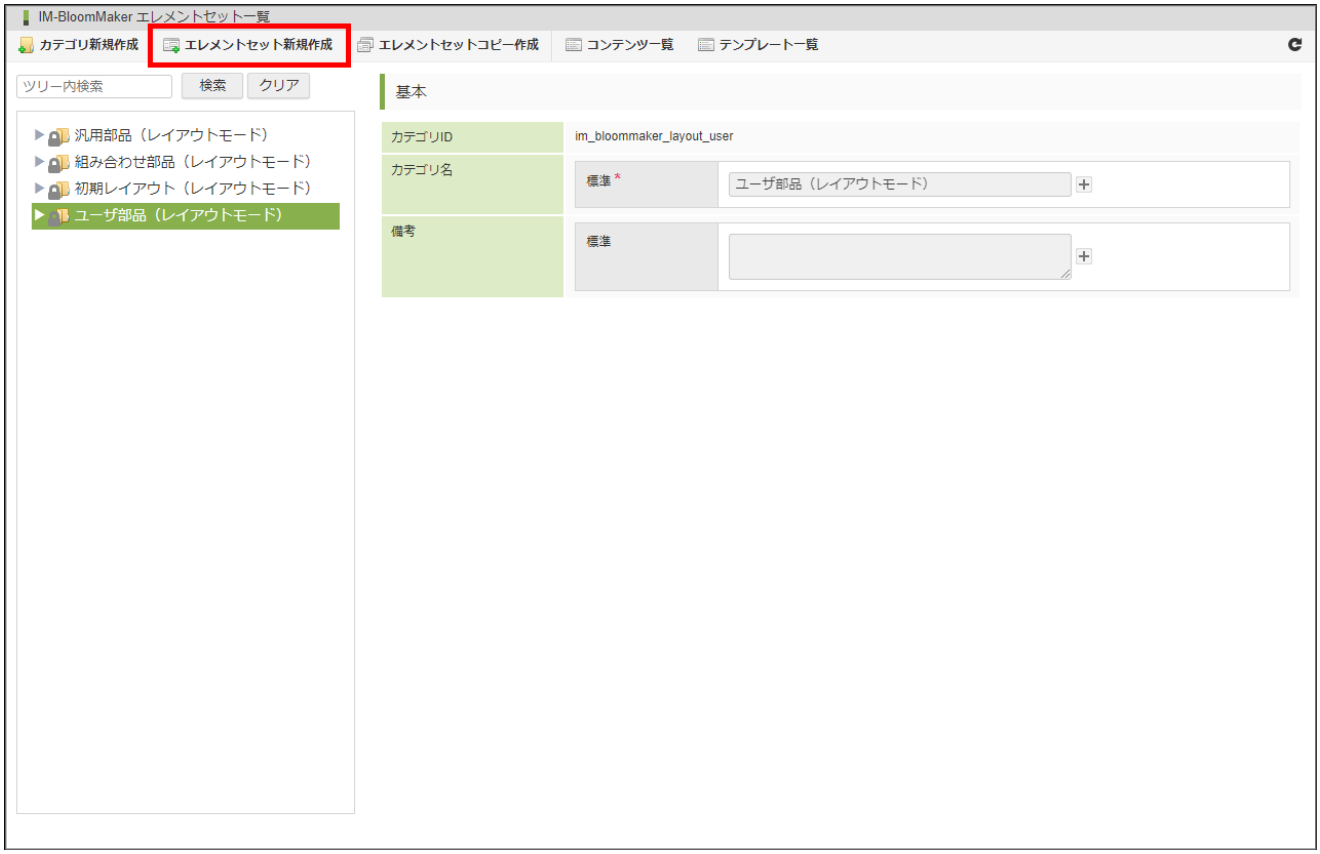
図：「サイトマップ」画面

2. エレメントセットカテゴリから「ユーザ部品（レイアウトモード）」を選択します。



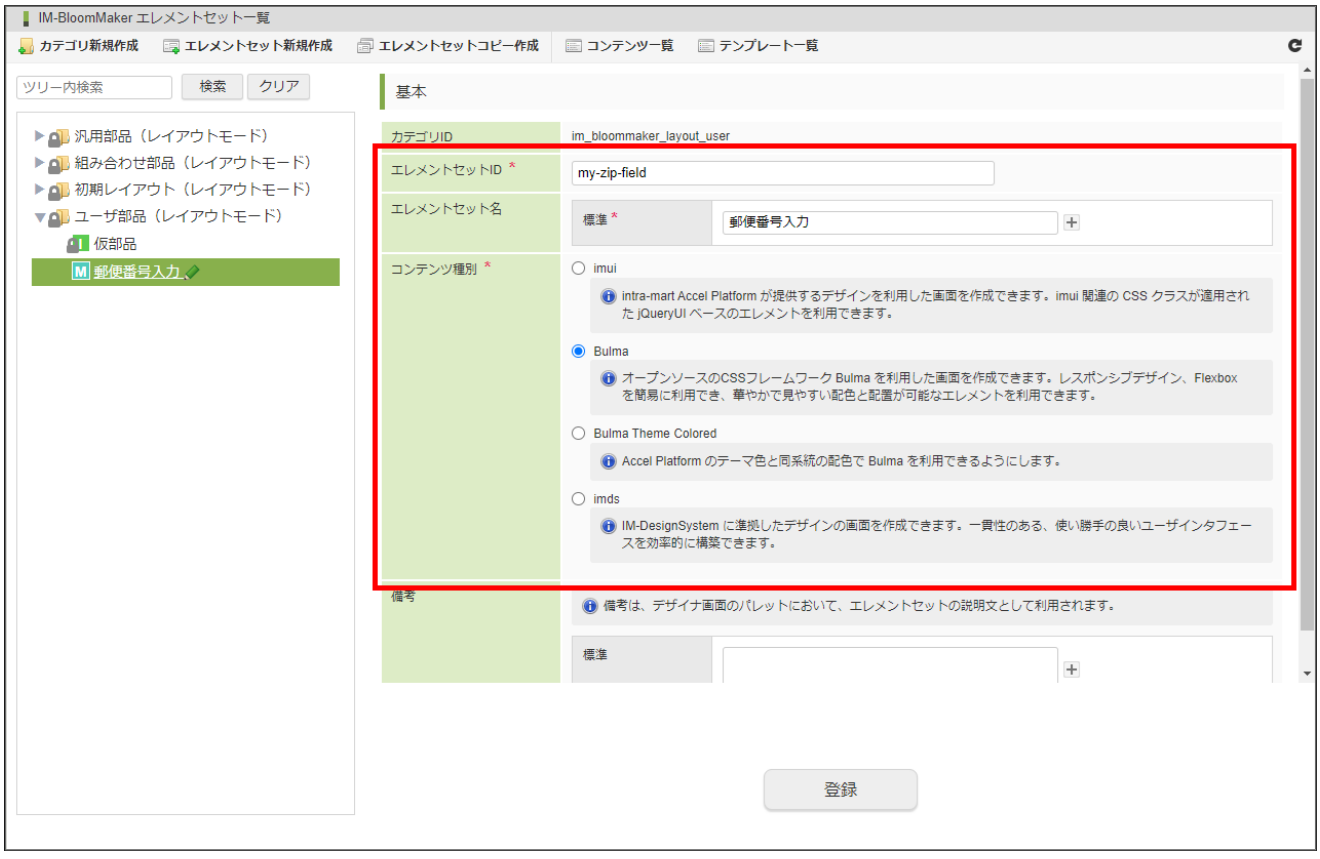
図：「エレメントセット一覧」画面 - エレメントセットカテゴリの選択

3. 「エレメントセット新規作成」をクリックします。



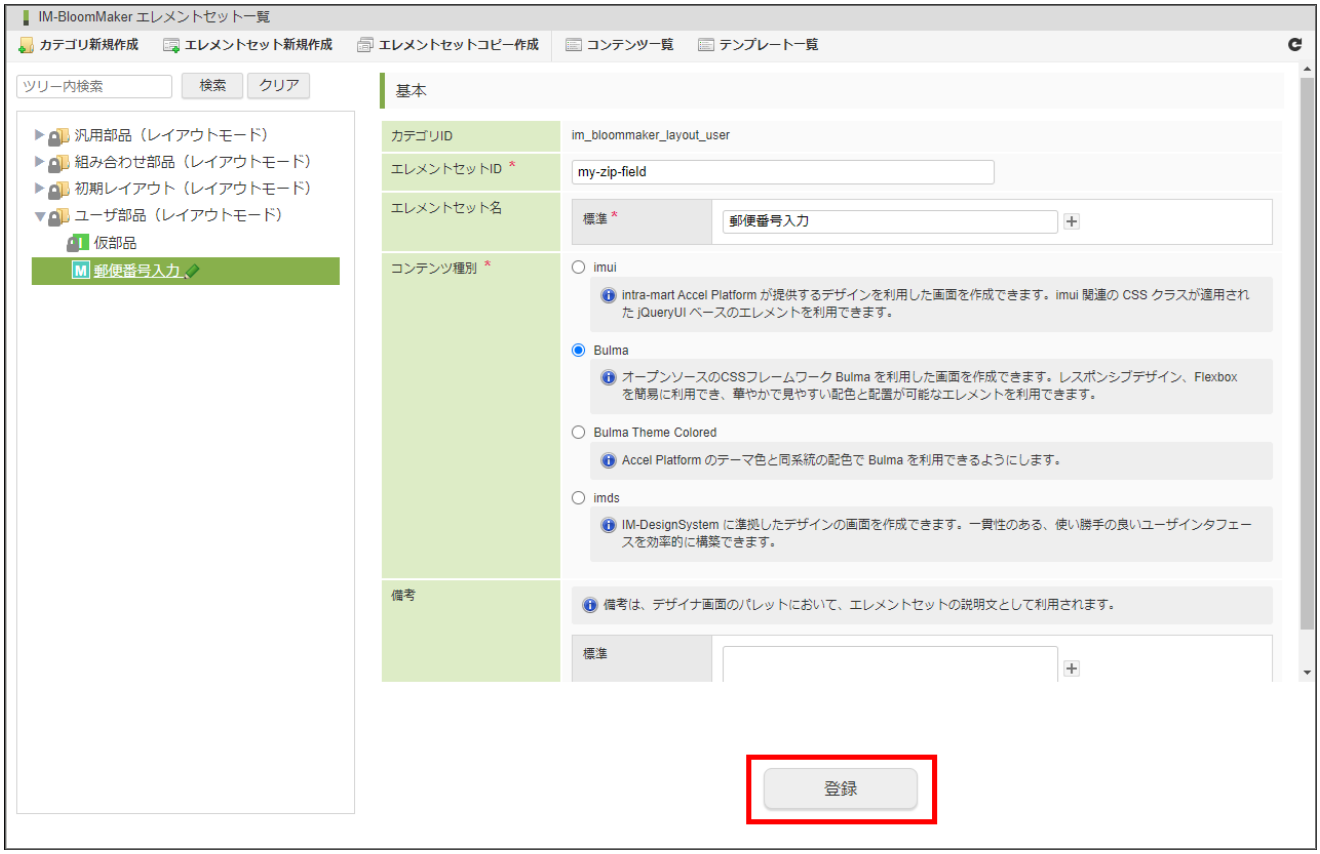
図：「エレメントセット一覧」画面 - 「エレメントセット新規作成」

4. エレメントセット情報の各項目を以下のとおりに設定します。
   - エレメントセットID：my-zip-field
   - エレメントセット名：郵便番号入力
   - コンテンツ種別：「Bulma」



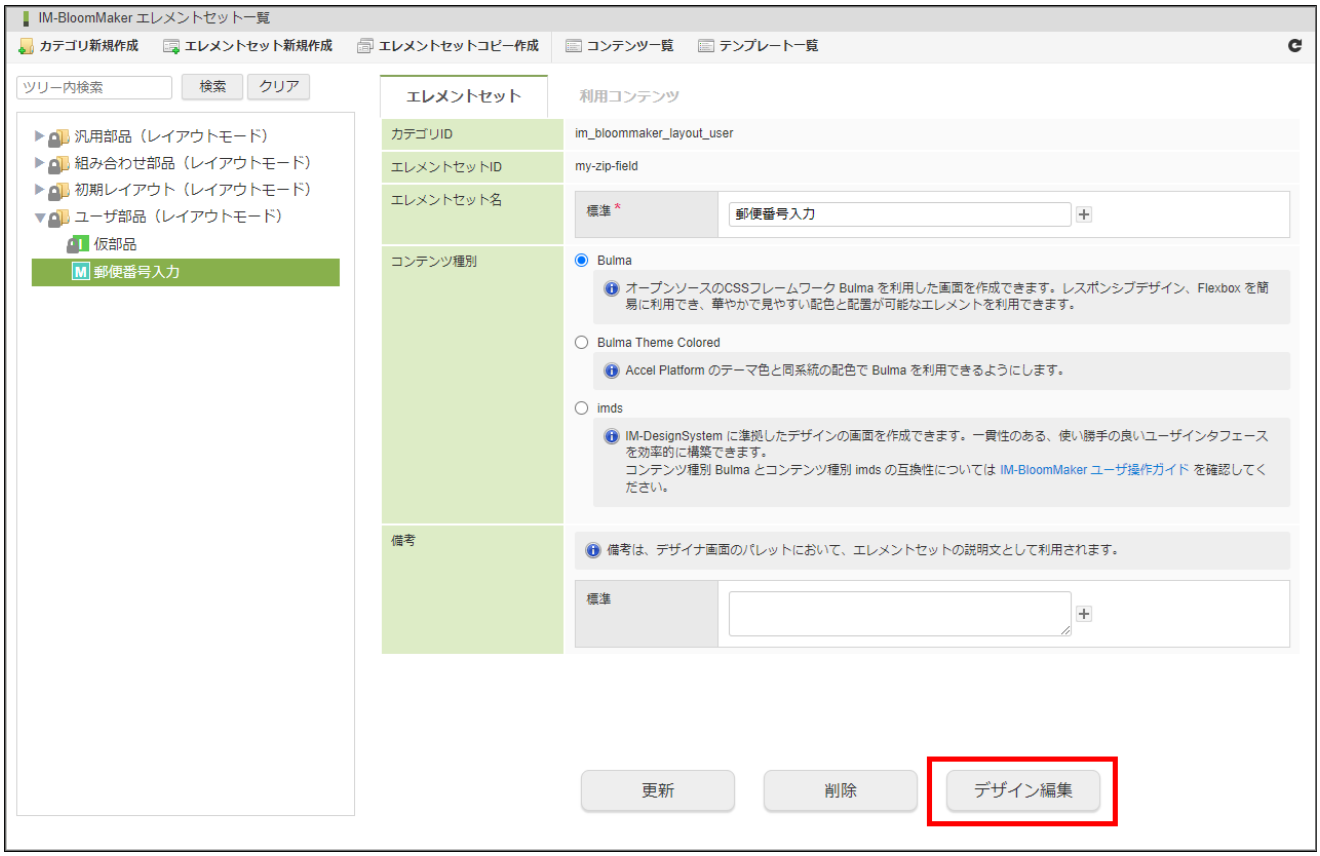
図：エレメントセット情報

5. 「登録」をクリックします。



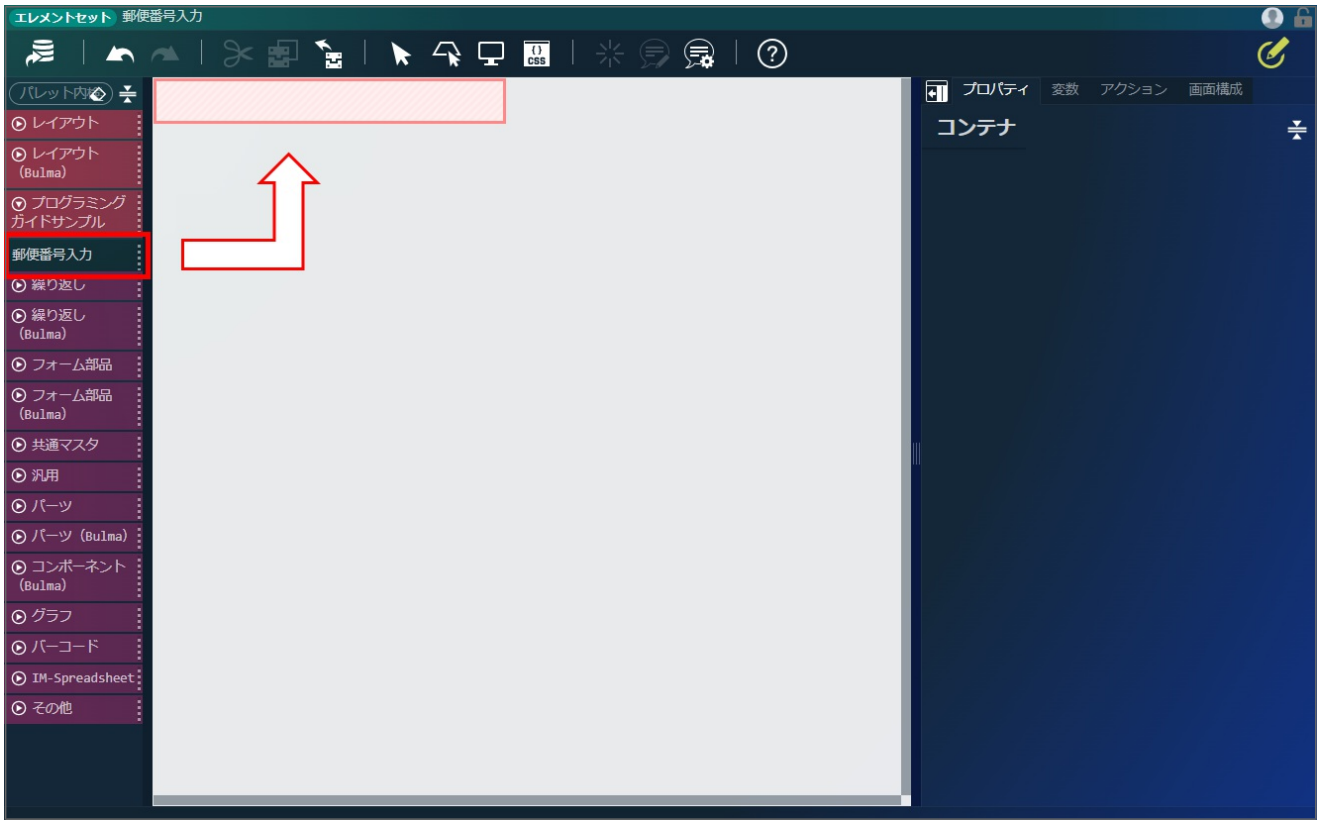
図：エレメントセットの登録

6. 「デザイン編集」をクリックして、デザイナを開きます。



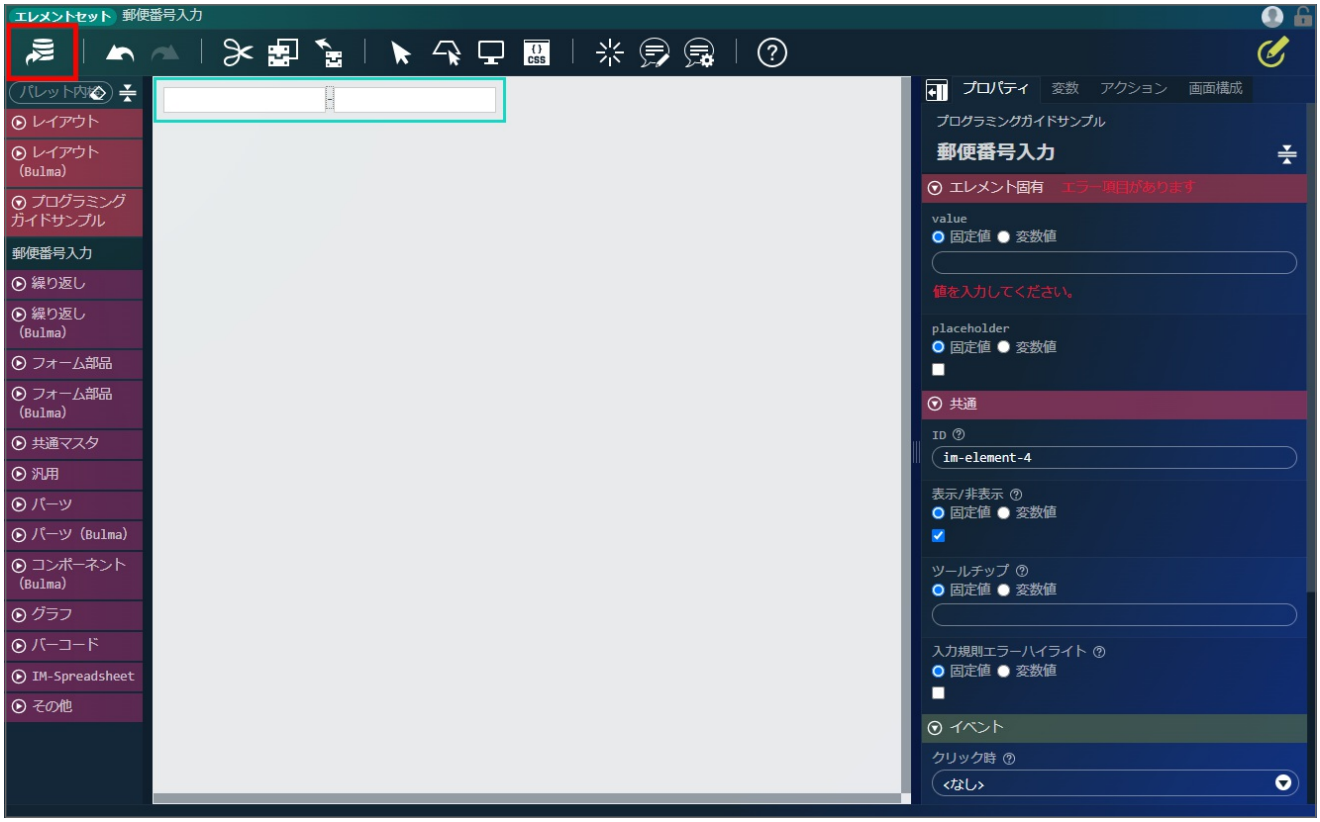
図：エレメントセットのデザイン画面表示

7. エレメントパレットから、「プログラミングガイドサンプル」-「郵便番号入力」をコンテナにドラッグ&ドロップします。



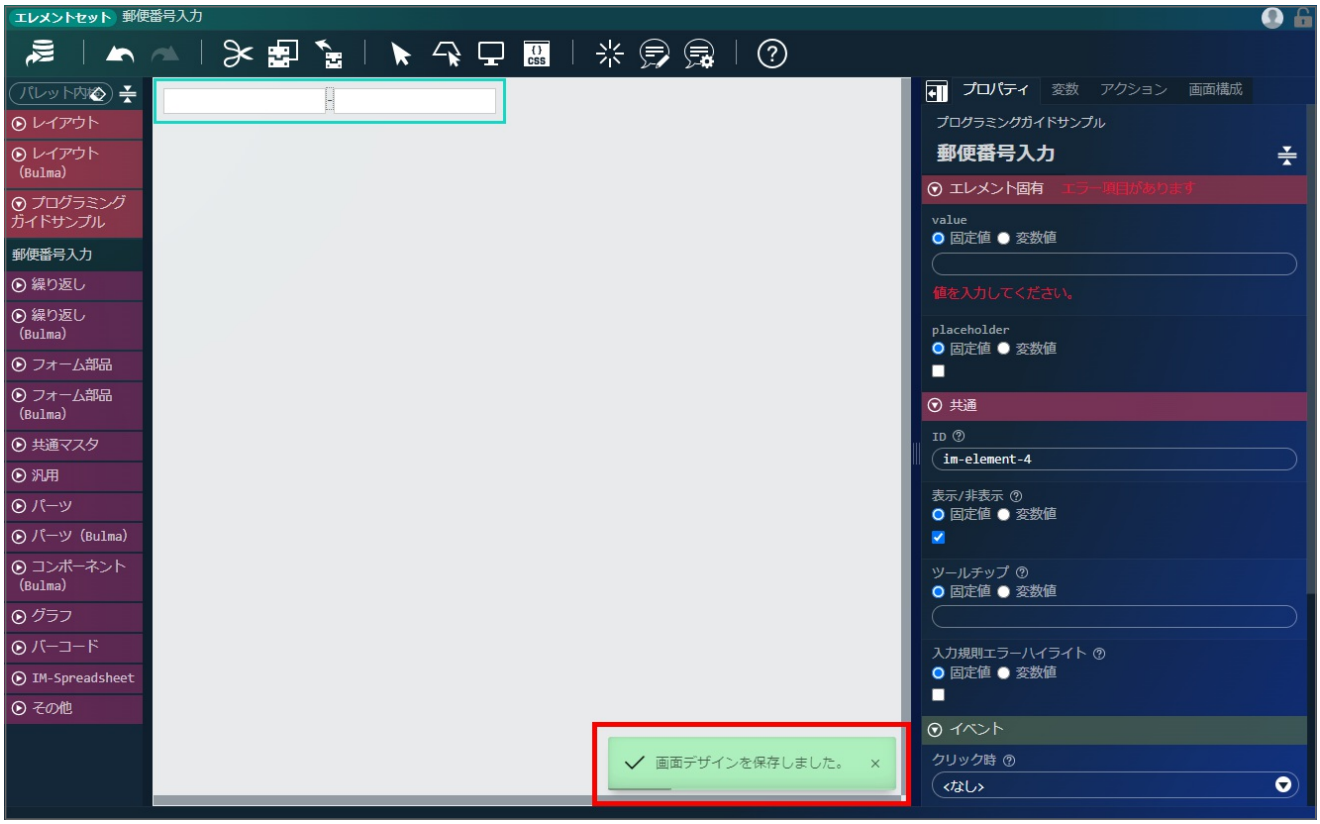
図：エレメントセットへエレメントの配置

8. 「デザイナ」画面上部、ヘッダ内の「上書き保存」アイコンをクリックします。



図：ヘッダ -「上書き保存」アイコン

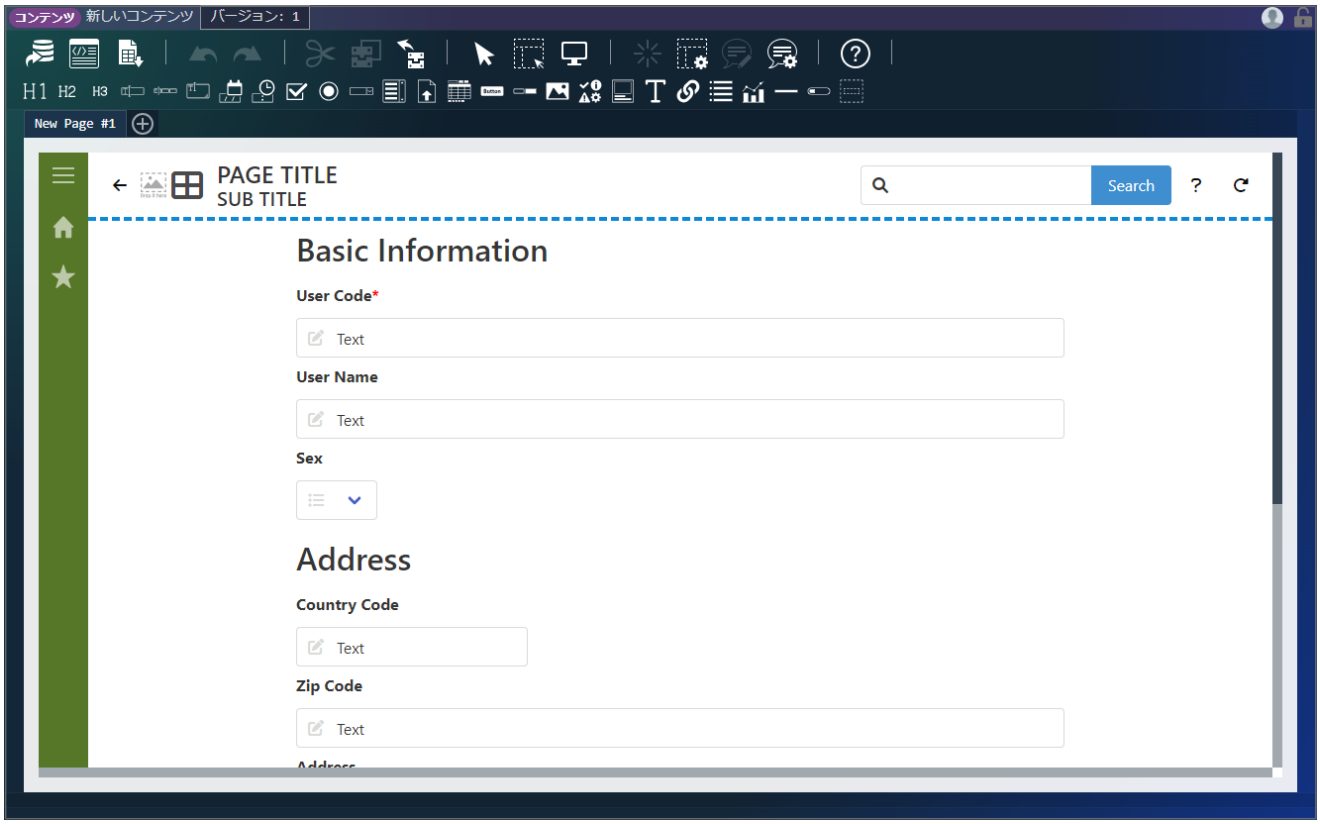
9. コンテナ下部に保存メッセージが表示されます。



図：保存メッセージ

これでレイアウトモードで「郵便番号入力」が使用可能になりました。
以下の手順で、レイアウトモードで「郵便番号入力」を配置します。

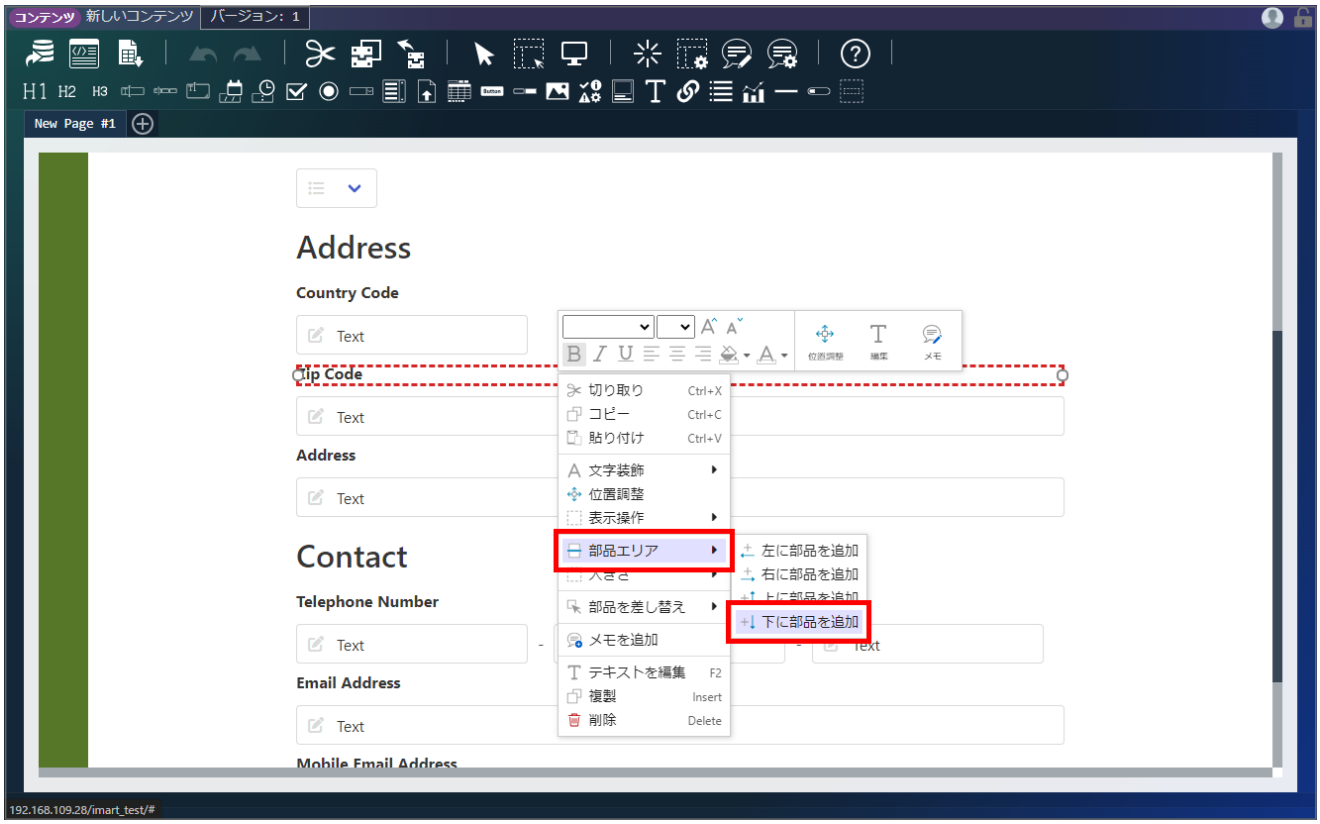
1. レイアウトモードでデザイナを開きます。



図：レイアウトモードのデザイナ

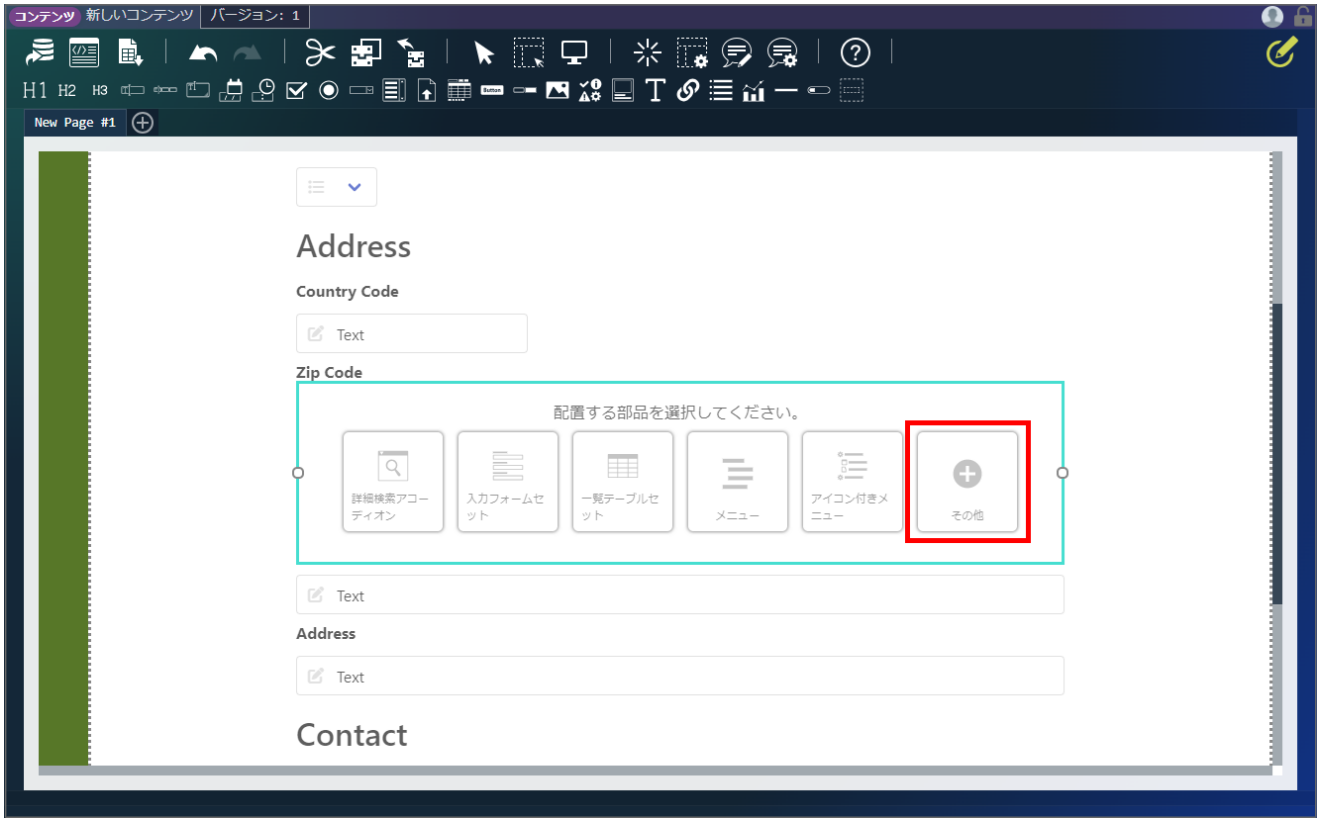
2. 「郵便番号入力」を配置したい場所に、部品エリアを追加します。

   下図の例では、右クリックメニューから「部品エリア」-「下に部品を追加」を選択し、「Zip Code」タイトルの下部に空白エリアを追加します。



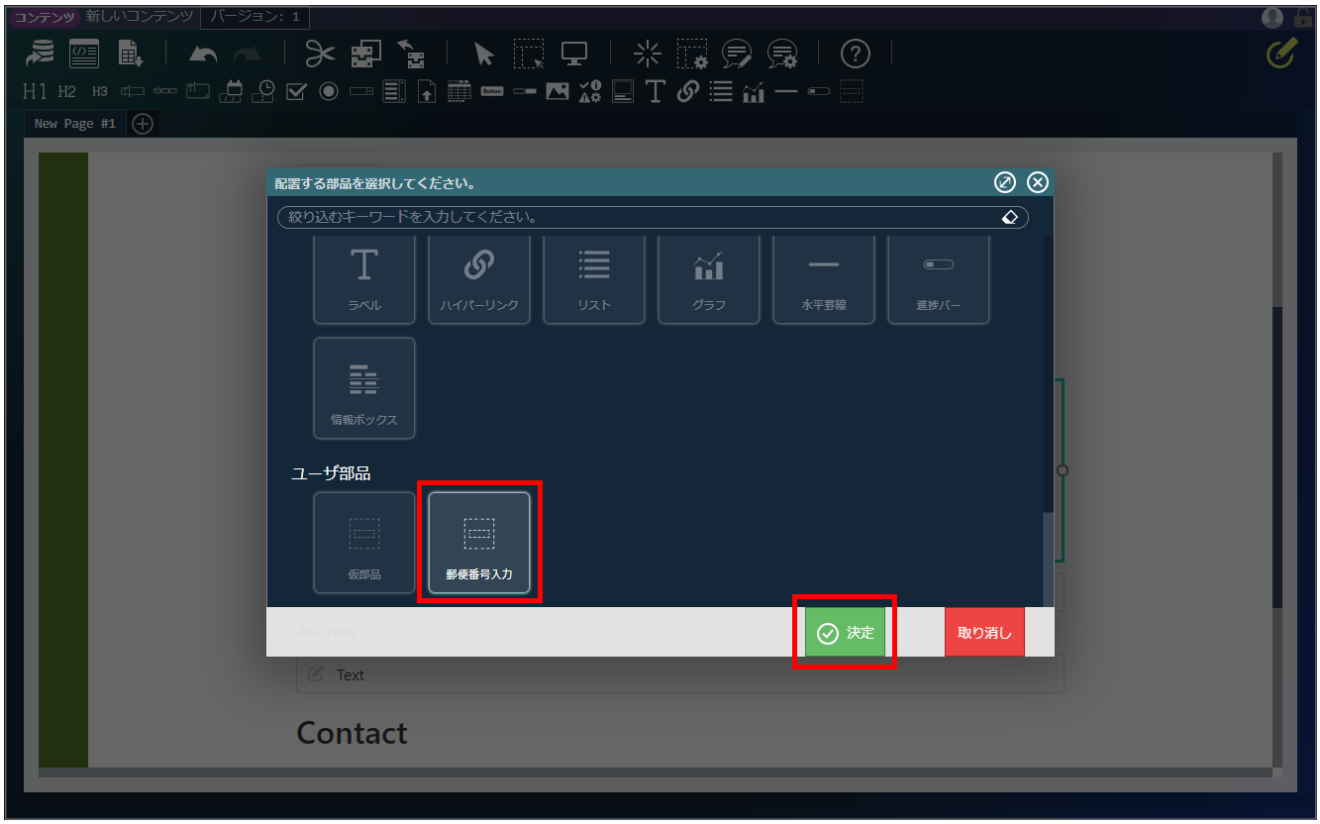
図：空白エリアの追加

3. 「部品選択」から「その他」をクリックします。



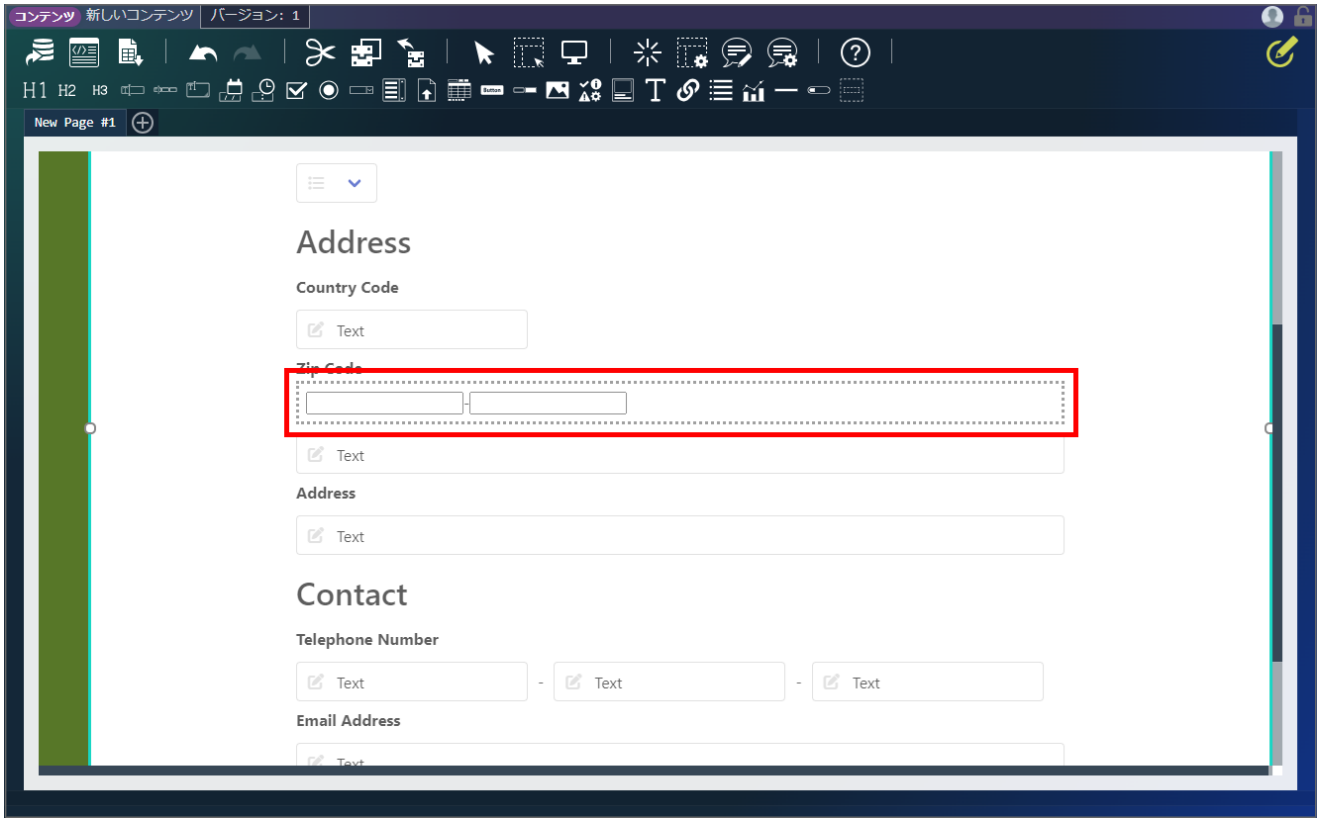
図：部品一覧

4. 部品選択ダイアログの「ユーザ部品」カテゴリから「郵便番号入力」を選択し、「決定」ボタンをクリックします。



図：部品選択ダイアログ

5. 「郵便番号入力」エレメントが配置できました。



図：デザイナ上へのエレメントの配置

アクションアイテムを実装する

この章では、アクションアイテムの実装方法について解説します。

アクションアイテム本体のファイルの実装

アクションアイテム本体のクラスの実装

まずは、アクションアイテム本体のファイルを実装していきます。
{VSCODE_HOME}/src/public/actions に、アクションアイテム本体を実装するファイル（.tsファイル）を作成します。
ここでは、MyShowAlertActionItem.ts というファイル名で作成しています。

このファイルに、IUIContainerActionItemCore インタフェースを実装した、アクションアイテムのクラスを定義します。

```
// パラメータの型を定義します。
type ParameterDefinition = {
    // TODO
};

export class MyShowAlertActionItem implements IUIContainerActionItemCore {
    // アクションアイテム名を返却するメソッドです。
    // { アクションアイテムのクラス名 } .name のように、アクションアイテムのクラス名を返却してください。
    // アクションアイテム名が他のアクションアイテム名と重複すると、正常に動作しない恐れがあります。
    public get actionTypeName(): string {
        return MyShowAlertActionItem.name;
    }

    // アクションアイテムの説明ラベルのメッセージキーを返却するメソッドです。
    public get messageKey(): string {
        return 'CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem';
    }

    // アクションアイテムのパラメータの定義を返却するメソッドです。
    public get parameterDefinition(): ParameterDefinition {
        // TODO
        return {};
    }

    // アクションアイテムが実行されたときの処理を記述します。
    public async run(
        context: IUIContainerActionContext,
        container: IUIContainer,
        component: IUIComponent,
        parameters: IUIContainerActionParameterAccessor<ParameterDefinition>
    ): Promise<void> {
        // TODO
    }
}
```

run メソッドを実装する

アクションアイテム本体の run メソッド内でアクションアイテムが実行されたときの処理を記述します。

例として、 window.alert を実行する場合の実装を以下に示します。

```
// アクションアイテムが実行されたときの処理を記述します。
public async run(
    context: IUIContainerActionContext,
    container: IUIContainer,
    component: IUIComponent,
    parameters: IUIContainerActionParameterAccessor<ParameterDefinition>
): Promise<void> {
    window.alert('any messages');
}
```

パラメータの利用方法

アクションアイテムを作成する際は、任意のパラメータを付与することも可能です。
window.alert の引数にメッセージを指定可能にするため、アクションアイテムのパラメータを追加で実装します。

以下の手順で実装します。

1. ParameterDefinition型を定義し、その中で、必要なプロパティを UIContainerActionParameterDefinitionType 型で宣言します。

   例えば message1、message2 というプロパティが必要な場合、まずは以下のように宣言します。

```
// パラメータの型を定義します。
type ParameterDefinition = {
  // message1 と message2 のパラメータを持ちます。
  message1: UIContainerActionParameterDefinitionType;
  message2: UIContainerActionParameterDefinitionType;
};
```

2. アクションアイテム本体のクラスの parameterDefinition メソッド内で、パラメータの定義を返却します。

   ParameterDefinition に従ってプロパティの定義を記述します。

```
// アクションアイテムのパラメータの定義を返却するメソッドです。
public get parameterDefinition(): ParameterDefinition {
  // ParameterDefinition 型で返却する必要があります。
  return {
    message1: {
      type: 'literal', // 固定文字列で指定したい場合
      messageKey: 'CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem.message1' // パラメータのラベル
    },
    message2: {
      type: 'variable-all', // 変数でしたい場合
      messageKey: 'CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem.message2' // パラメータのラベル
    }
  };
}
```

コラム

parameterDefinition メソッドの返り値について

parameterDefinition メソッドの返り値は、ParameterDefinition 型（UIContainerActionParameterDefinitionType 型のプロパティを持つオブジェクト）にする必要があります。
UIContainerActionParameterDefinitionType の実装は以下の通りです。
parameterDefinition メソッドでは、この形式に従って必要な定義を記述してください。
? がついている定義は省略可能です。

```
type UIContainerActionParameterDefinitionType = {
  /** パラメータ定義タイプ */
  readonly type: UIContainerActionParameterCandidateType;

  /** パラメータ名を示すメッセージキー */
  readonly messageKey: string;

  /** パラメータの選択候補 */
  readonly candidate?: {
    readonly displayName: string;
    readonly value: string;
  }[];
};

type UIContainerActionParameterCandidateType =
  | 'variable-input' /** 定数・入力 */
  | 'variable-output' /** 変数 */
  | 'variable-all' /** 変数・定数・入力 */
  | 'literal' /** 直接入力 */
  | 'label-target' /** ラベル定義 */
  | 'label' /** アクション内ラベル */
  | 'action' /** アクション（自分を除く） */
  | 'page' /** コンテナページ */
  | 'table-select' /** テーブル */
  | 'checkbox' /** チェックボックス */
  | 'javascript' /** JavaScript エディタ */
  | 'hidden'; /** 隠しパラメータ */
```

3. run メソッド内で window.alert の引数にメッセージを指定可能にするため、アクションアイテムのパラメータを追加で実装します。

```
  // アクションアイテムが実行されたときの処理を記述します。
  public async run(
      context: IUIContainerActionContext,
      container: IUIContainer,
      component: IUIComponent,
      parameters: IUIContainerActionParameterAccessor<ParameterDefinition>
  ): Promise<void> {
      // アクションアイテムのパラメータを文字列で取得します。
      const message1 = parameters.getParameter('message1').toArgument(container).toString();
      const message2 = parameters.getParameter('message2').toArgument(container).toString();

      window.alert(message1 + '\n' + message2);
  }
```

実装したクラスの登録

アクションアイテム本体のクラスを登録する処理を {VSCODE_HOME}/src/index.ts に実装します。
UIContainerActionItemRepository クラスのメソッドを利用します。

```
import {MyShowAlertActionItem} from './public/actions/MyShowAlertActionItem';

// アクションアイテムのリポジトリは以下のように取得します。
const actionItemRepository = window.imHichee.UIContainerActionItemRepository;

// アクションアイテムのカテゴリを登録します。
// 新たにカテゴリを追加したい場合には UIContainerActionItemRepository.registerCategory() を使用して、カテゴリを追加してください。
// 一つ目の引数 id にはカテゴリのID（任意）、二つ目の引数にはアクションアイテムの登録オプションを指定可能です。
// sortNumber でカテゴリを表示させる際の優先順位を指定可能です。
// カテゴリは sortNumber を基準に昇順で表示されます。
actionItemRepository.registerCategory('programming-sample', {sortNumber: 120});

// 作成したアクションアイテムをカテゴリに登録します。
// categoryId には登録先のカテゴリ名、sortNumber はカテゴリ内でアクションアイテムを表示させる際の優先順位を指定できます。
// sortNumber を基準に昇順で表示されます。
actionItemRepository.registerClass(MyShowAlertActionItem, {
    categoryId: 'programming-sample',
    sortNumber: 0,
});
```

プロパティファイルの実装

アクションアイテムのラベル、新たに追加したカテゴリの名称、ヘルプを画面上に表示するために、プロパティファイルを実装する必要があります。
必要な言語に応じて、{VSCODE_HOME}/src/public/messages 配下にプロパティファイルを作成します。
日中英の3ロケール（言語）に対応する場合は、以下の4ファイルが必要です。不要なロケールはプロパティファイルを作成する必要はありません。

- caption_ja.properties
  日本語のプロパティファイルです。
- caption_en.properties
  英語のプロパティファイルです。
- caption_zh_CN.properties
  中国語のプロパティファイルです。
- caption.properties
  対応するロケールのプロパティが欠落している場合に参照されるプロパティファイルです。

それぞれのプロパティは以下のように記述します。

- アクションアイテムのラベル：CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.{アクションアイテム本体のクラスに利用したメッセージ}={表示したいラベルの内容}
- アクションアイテムのカテゴリ名：CAP.Z.IWP.HICHEE.ACTION.ITEM.CATEGORY.NAME.{index.ts内で指定したカテゴリID}={表示したいカテゴリ名}
- アクションアイテムのヘルプ：CAP.Z.IWP.HICHEE.ACTION.ITEM.DESCRIPTION.{アクションアイテムのクラス名}={表示したいヘルプの内容}

日本語のプロパティファイルを記述する際の例を以下に示します。（先頭に#をつけて、コメントを記述することも可能です。）

```
# アクションアイテムの説明ラベル
CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem=アラートでメッセージを表示

# アクションアイテムのパラメータラベル
CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem.message1=メッセージ1
CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem.message2=メッセージ2

# アクションアイテム - カテゴリ名
CAP.Z.IWP.HICHEE.ACTION.ITEM.CATEGORY.NAME.programming-sample=プログラミングガイドサンプル

# アクションアイテム - ヘルプ
CAP.Z.IWP.HICHEE.ACTION.ITEM.DESCRIPTION.MySampleElement=サンプルアクションアイテムです。
```

なお、以下のようにアクションアイテムのラベルに {パラメータ名} を埋め込むことにより、「アラートでメッセージ○と○を表示」のように穴埋め形式でラベルを表示させることが可能です。

```
# アクションアイテムの説明ラベル
CAP.Z.IWP.HICHEE.ACTION.ITEM.LABEL.MyShowAlertActionItem=アラートでメッセージ{message1}と{message2}を表示
```

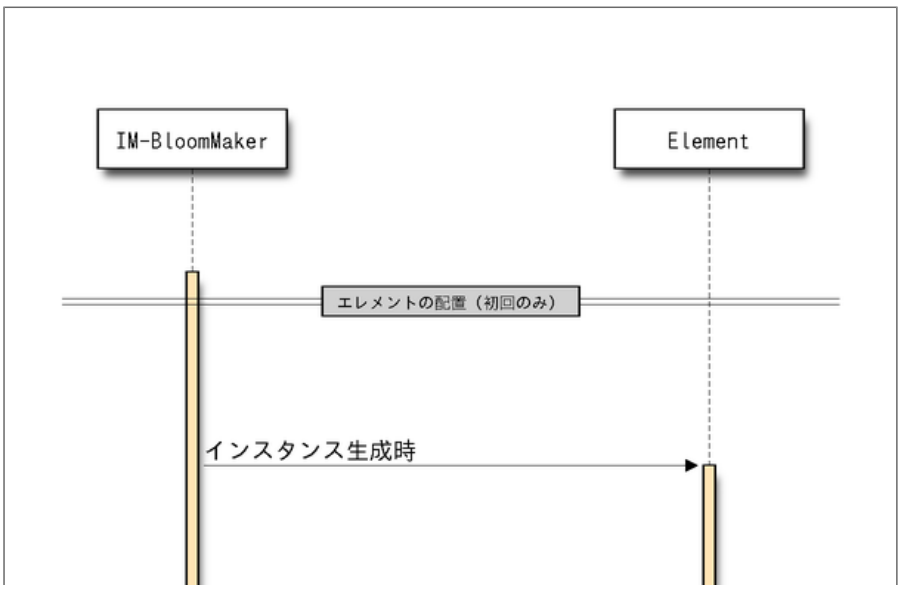
> **コラム**
>
> **パラメータの必須・任意について**
>
> メッセージ内に埋め込んだ {パラメータ名} は、自動的に必須項目として扱われます。アクションエディタでパラメータが未指定の時はアクションの保存時に警告が表示されます。
> メッセージに埋め込まれていないパラメータは、すべて任意項目として扱われます。

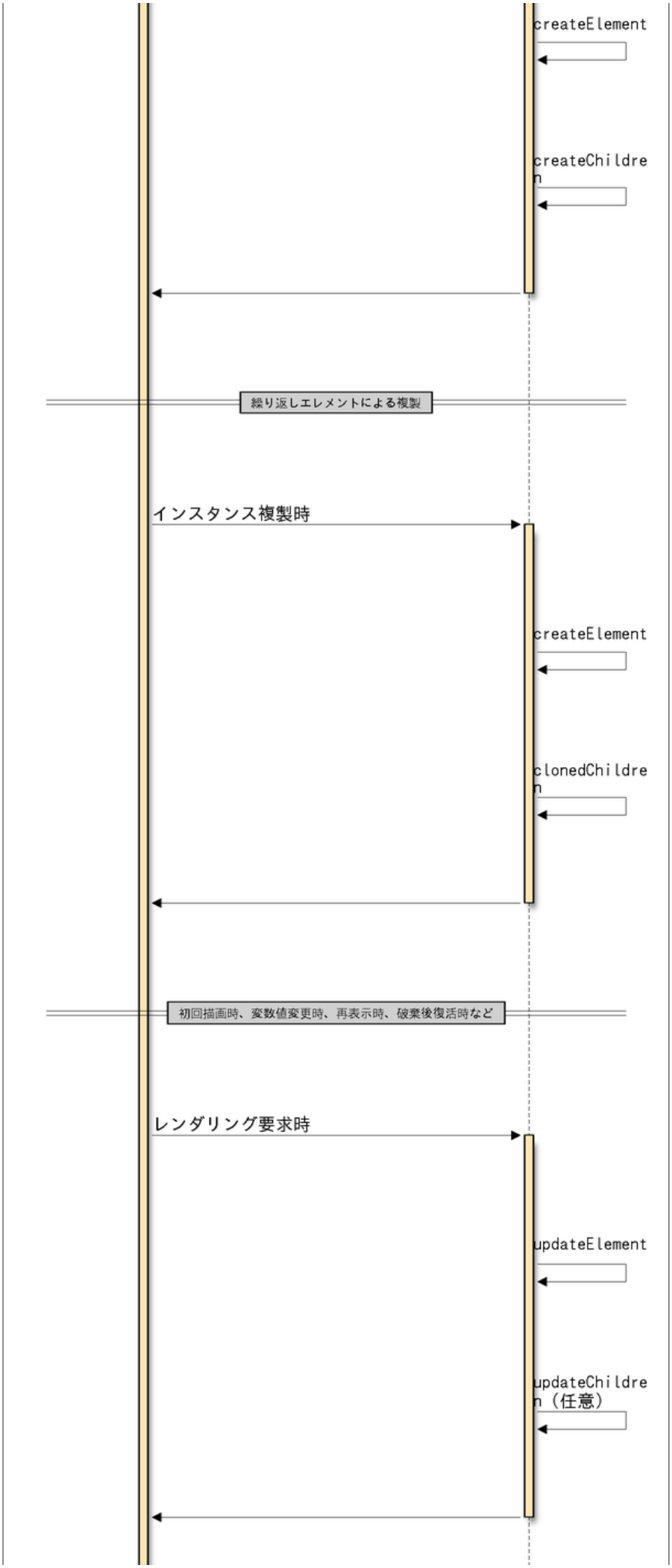
> **注意**
>
> **アクションアイテムの命名について**
>
> アクションアイテムのクラス名が（IM-BloomMaker 標準のアクションアイテムも含めて）複数のアクションアイテムで重複すると、正常に動作しないため、一意なクラス名を付けるようにしてください。
>
> 2020 Spring(Yorkshire) 時点の、既存のアクションアイテムのクラス名の一覧は、「*付録) 2020 Spring(Yorkshire) 時点のアクションアイテムのクラス名一覧*」を参照してください。
> また、2020 Spring(Yorkshire) 以降のリリースで追加されるアクションアイテムに関しては、すべてクラス名の先頭に *Im* が付与されます。
> 独自のアクションアイテムを実装する際は、独自の接頭辞を付与するなどの方法で、クラス名の重複を回避してください。

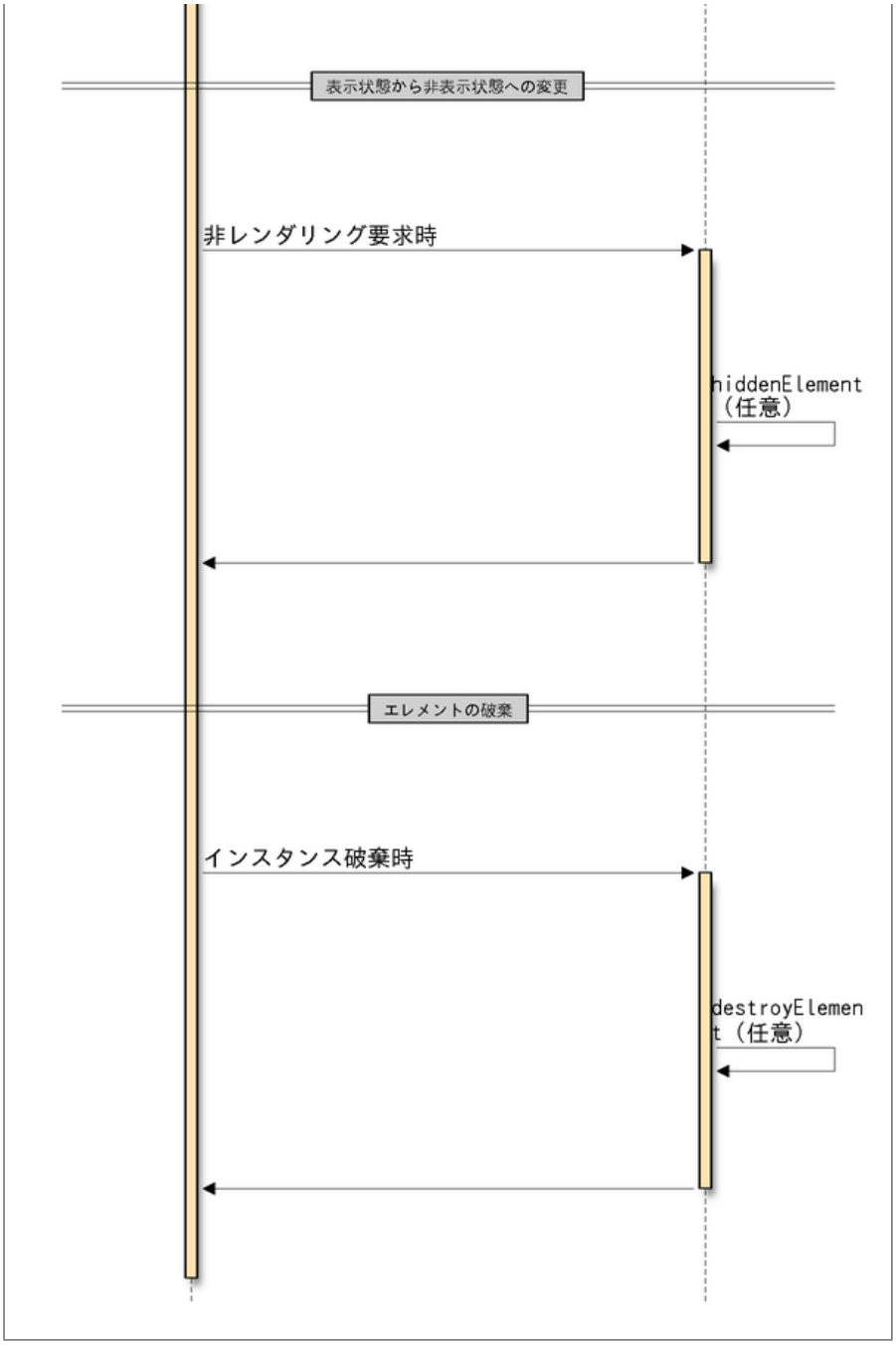
アクションアイテムの作成に必要な作業は以上です。
「*bundleの生成*」に戻って残りの作業を行うことで、実装したアクションアイテムを利用できます。

実装上の補足

エレメントのライフサイクル図




createElement
createChildren
繰り返しエレメントによる複製
インスタンス複製時
createElement
clonedChildren
初回描画時、変数値変更時、再表示時、破棄後復活時など
レンダリング要求時
updateElement
updateChildren（任意）



エレメントでのプロパティ操作

createElement や updateElement メソッドの中で、プロパティを操作したい場合があります。
プロパティから値を取得したい場合、以下のように実装します。

```
const uniquePropertyDefinition: PropertyDefinition & IUniquePropertyDefinition = {
    stringProperty: {
        displayName: 'stringProperty',
        definition: {},
        type: 'string',
        value: 'string',
    },
    integerProperty: {
        displayName: 'integerProperty',
        definition: {},
        type: 'integer',
        value: 1,
    },
    decimalProperty: {
        displayName: 'decimalProperty',
        definition: {},
        type: 'decimal',
        value: 10,
    },
    booleanProperty: {
        displayName: 'booleanProperty',
        definition: {},
        type: 'boolean',
        value: true,
    },
}

...

public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // 文字列として取得する
    const stringValue = properties.getProperty('stringProperty').toString();

    // 数値として取得する
    const integerValue = properties.getProperty('integerProperty').toNumber();
    const decimalValue = properties.getProperty('decimalProperty').toNumber();

    // 真偽値として取得する
    const booleanValue = properties.getProperty('booleanProperty').toBoolean();

    // null かどうかを判定する
    const isNull = properties.getProperty('sampleProperty').isNull();

    // 配列かどうかを判定する
    const isArray = properties.getProperty('arrayProperty').isArray();

    // オブジェクトかどうかを判定する
    const isObject = properties.getProperty('objectProperty').isObject();

    ...
}
```

プロパティに値をセットしたい場合、以下のように実装します。
メソッドの第2引数にはプリミティブな値を指定します。

```
public updateElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): void {
    // 文字列をセットする
    properties.setProperty('stringProperty', 'foo', false);

    // 数値をセットする
    properties.setProperty('integerProperty', 1, false);
    properties.setProperty('decimalProperty', 10, false);

    // 真偽値をセットする
    properties.setProperty('booleanProperty', true, false);

    // null をセットする
    properties.setProperty('sampleProperty', null, false);

    // 配列をセットする
    properties.setProperty('arrayProperty', ['foo', 'bar'], false);

    // オブジェクトをセットする
    properties.setProperty('objectProperty', {foo: 'bar'}, false);

    ...
}

public createElement(
  container: IUIContainer,
  properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);
    const tag = builder.tag as HTMLInputElement;

    // 自身のエレメントも含めて再レンダリングするときだけ第3引数に true を指定する
    // 自身の updateElement が再度呼び出されるため、無限ループに注意する
    tag.addEventListener('xxx', () => {
        properties.setProperty('stringProperty', 'somevalue', true);
    }

    return builder;
}
```

アクションでのパラメータ操作

アクションアイテムを実行する際に、アクションアイテムのパラメータに設定された値を取得する場合は、parameters.getParameter を使用します。
変数の値を書き換えたい場合は、container.parameters.writeAs*** を使用します。
以下のように実装します。

```
public run(
    context: IUIContainerActionContext,
    container: IUIContainer,
    component: IUIComponent,
    parameters: IUIContainerActionParameterAccessor<ParameterDefinition>
) {
    // 取得
    // アクションダイアログの設定どおりに取得したい場合
    const paramRaw = parameters.getParameter('param').toRaw(container);

    // 変数の値を取得したい場合
    const paramArgument = parameters.getParameter('param').toArgument(container);

    ...

    // セット
    // プリミティブな値をセットする場合
    container.parameters.writeAsPrimitive(container, parameters.getParameter('param').toRaw(), paramPrimitive);

    // Argument をセットする場合
    container.parameters.writeAsArgument(container, parameters.getParameter('param').toRaw(), paramArgument);

    // 配列を考慮してセットする場合
    // 変数のパラメータが配列の場合、自動的に要素数1の ArrayArgument に変換されます。
    // 変数のパラメータが配列でなく、かつ value に配列が指定された場合、最初の要素に対して Argument に変換されます。
    container.parameters.writeAsPrimitiveStrictArrayTyped(container, parameters.getParameter('param').toRaw(), param);

    ...
}
```

Argument の生成方法

window.imHichee.ArgumentFactory を使用してください。
作成する Argument の型に合わせ、以下のメソッドを利用してください。

- createAsArray: (container: IUIContainer, raw: IArgument[]) => IArrayArgument;
  - ArrayArgument を作成
- createAsBoolean: (container: IUIContainer, raw: boolean) => IArgument;
  - BooleanArgument を作成
- createAsDate: (container: IUIContainer, raw: string | number | {date: Date; offset?: number}) => IDateArgument;
  - DateArgument を作成
- createAsDouble: (container: IUIContainer, raw: number) => INumberArgument;
  - DoubleArgument を作成
- createAsEmptyObject: (container: IUIContainer) => IObjectArgument;
  - 空の ObjectArgument を作成
- createAsFraction: (container: IUIContainer, raw: IFraction) => INumberArgument;
  - FractionArgument を作成
- createAsInteger: (container: IUIContainer, raw: number) => INumberArgument;
  - IntegerArgument を作成
- createAsNull: (container: IUIContainer) => IArgument;
  - NullArgument を作成
- createAsObject: (container: IUIContainer, raw: IKeyValueMap<string, IArgument>) => IObjectArgument;
  - ObjectArgument を作成
  - 引数の raw には `new window.imHichee.KeyValueMap<string, IArgument>()` で作成したインスタンスを指定してください。
- createAsParameterPath: (container: IUIContainer, target: IUIComponent, raw: string) => IParameterPathArgument;
  - ParameterPathArgument を作成
- createAsString: (container: IUIContainer, raw: string) => IArgument;
  - StringArgument を作成

使用例は以下のとおりです。

```
const ArgumentFactory = window.imHichee.ArgumentFactory;
const KeyValueMap = window.imHichee.KeyValueMap;

// StringArgument を作成する
const stringArgument = ArgumentFactory.createAsString(container, 'foo');

// ObjectArgument の中身を作成する
const obj = new KeyValueMap<string, IArgument>();
obj.put('key1', ArgumentFactory.createAsInteger(container, 1));
obj.put('key2', ArgumentFactory.createAsInteger(container, 2));
// ObjectArgument を作成する
const objectArgument = ArgumentFactory.createAsObject(container, obj);
```

プロパティのイベントに指定されているイベントタイプ

プロパティの「イベント」で指定されるアクションが、どの DOM イベントを利用しているかを列挙します。
onMouseDown 以下はプロパティには表示されませんが、エレメントを作成する際に利用可能なイベントです。

イベントに指定されたアクションは createElement と updateElement の間に DOM にバインドされます。

イベント	イベントタイプ
クリック時	click
ダブルクリック時	dblclick
キー押下時	keydown
フォーカスイン	focus
フォーカスアウト	blur
入力値変更時	change
入力値変更中	input
onMouseDown	mousedown
onMouseMove	mousemove
onMouseEnter	mouseenter
onMouseLeave	mouseleave
onMouseOut	mouseout
onMouseOver	mouseover
onMouseUp	mouseup
onKeyPress	keypress
onKeyUp	keyup

```
// 例：フォーカスアウト時に、プロパティで指定されたアクションの前に処理を行う
public createElement(
  container: IUIContainer,
  properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
   const builder = window.imHichee.HTMLElementBuilder.createElement('input', HTMLInputElement);
   const tag = builder.tag as HTMLInputElement;

   // 上記のフォーカスアウトに相当するイベントタイプ blur を指定する
   tag.addEventListener('blur', () => {
      // 処理
   }

   return builder;
}
```

エレメントのインスタンスを作成

エレメントのインスタンスを作成する場合は、コンテナからエレメントコントローラを取得し、コントローラにインスタンスの作成を依頼しま

す。
以下のように実装します。

```
// 例：自身エレメントの子として、いくつかのエレメントを作成する
public createChildren(
    self: IUIElement,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IUIElement[] {
    return [
        // クラスを直接指定する場合
        container.controller.createInstanceByClass(MyZipCodeField, self),
        // クラス名（文字列）を指定する場合
        container.controller.createInstanceByTypeName('MyZipCodeField', self)
    ];
}
```

親エレメントの取得

親エレメントを取得する場合は、element.parent プロパティから取得します。
以下のように実装します。

```
// 例：フォーカスアウト時に、self から親エレメントを取得する
public createChildren(
    self: IUIElement,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IUIElement[] {
    const tag = self.builder.tag;

    // 上記のフォーカスアウトに相当するイベントタイプ blur を指定する
    tag.addEventListener('blur', () => {
        // 最新の親を取得する場合は self.parent から取得する
        const parent = self.parent;

        // parent はエレメントまたはコンテナ。 parent.clazz で判別可能
        if (parent.clazz === 'UIContainer') {
            // コンテナの場合
        } else if (parent.clazz === 'UIElement') {
            // エレメントの場合
        }

        ...
    }

    ...
}
```

子エレメントの取得

子エレメントを取得する場合は、element.children プロパティから取得します。
ただし、children プロパティから取得した子エレメントは、順序が保証されていません。
画面上で配置されている順序通りに取得したい場合は、コンテナからエレメントコントローラを取得し、コントローラから並び替え済みの子エレメント一覧を取得します。

以下のように実装します。

```
// 例：フォーカスアウト時に、self から子エレメントを取得する
public createChildren(
    self: IUIElement,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IUIElement[] {
    const tag = self.builder.tag;

    // 上記のフォーカスアウトに相当するイベントタイプ blur を指定する
    tag.addEventListener('blur', () => {
        // 最新の子エレメントを取得する場合は、self.children から取得する
        // ただし、これは順不同であり、必ずしも配置順ではないことに注意する
        const children = self.children;

        // 配置順で子エレメントを取得する場合は、container 経由で取得する
        // ただし、並び替えを行うため、パフォーマンスが悪化するので、順序が重要な場合にのみ使用する
        const sortedChildren = container.controller.getSortedChildren(self);

        ...
    }

    ...
}
```

エレメント・アクションアイテムを無効化する

この章では、任意のエレメントやアクションアイテムを無効化し、利用できなくする方法を説明します。

> **注意**
>
> エレメント・アクションアイテムの無効化は、2021 Summer(Cattleya) 以降のバージョンで可能です。
> 無効化したエレメント、アクションアイテムは、デザイナ画面のパレット上に表示されなくなります。
> また、2021 Winter(Dandelion) 以降のバージョンでは、無効化したエレメントやアクションアイテムを利用しているコンテンツのデザイナ画面・プレビュー画面を開いた際に、エラーメッセージが表示されます。

作業の手順

作業の手順は「*エレメント・アクションアイテムを作成する*」と同様です。「*事前準備*」にしたがって事前準備を行い、作業の流れを確認してください。
「*実装作業*」を「*エレメントを無効化する処理の実装*」、「*アクションアイテムを無効化する処理の実装*」に読み替えてください。

エレメントを無効化する処理の実装

エレメントを無効化する処理を `{VSCODE_HOME}/src/index.ts` に実装します。

```
// エレメントのリポジトリを取得します。
const elementRepository = window.imHichee.UIElementRepository;

// 無効化したいエレメントのクラス名を配列に列挙してください。
const excludesElementClassNames = ['MySampleElement'];

// IUIElementRepository.classes で登録済みのエレメントの一覧を取得し、filter を使って無効化するエレメントを絞り込みます。
const disableElements = elementRepository.classes.filter((clazz) => {
  return excludesElementClassNames.find((excludeClazz) => excludeClazz === clazz.name);
});

// エレメントを無効化する場合は、IUIElementRepository.setClassOptions() の第二引数の hiddenInPalette に true を指定します。
disableElements.forEach((item) => elementRepository.setClassOptions(item, {hiddenInPalette: true}));
```

IM-BloomMaker が提供する標準のエレメントを無効化したい場合は、「*IM-BloomMakerが提供する標準のエレメントのクラス名と提供開始バージョン一覧*」を参照して、無効化したいエレメントのクラス名を `excludesElementClassNames` の配列に列挙してください。

アクションアイテムを無効化する処理の実装

アクションアイテムを無効化する処理を `{VSCODE_HOME}/src/index.ts` に実装します。

```
// アクションアイテムのリポジトリを取得します。
const actionItemRepository = window.imHichee.UIContainerActionItemRepository;

// 無効化したいアクションアイテムのクラス名を配列に列挙してください。
const excludesActionItemClassNames = ['MySampleActionItem'];

// IUIContainerActionItemRepository.classes で登録済みのアクションアイテムの一覧を取得し、filter を使って無効化するアクションアイテムを絞り込みます。
const disableActionItems = actionItemRepository.classes.filter((clazz) => {
  return excludesActionItemClassNames.find((excludeClazz) => excludeClazz === clazz.name);
});

// アクションアイテムを無効化する場合は、IUIContainerActionItemRepository.setClassOptions() の第二引数の hiddenInPalette に true を指定します。
disableActionItems.forEach((item) => actionItemRepository.setClassOptions(item, {hiddenInPalette: true}));
```

IM-BloomMaker が提供する標準のアクションアイテムを無効化したい場合は、「*IM-BloomMakerが提供する標準のアクションアイテムのクラス名と提供開始バージョン一覧*」を参照して、無効化したいアクションアイテムのクラス名を `excludesActionItemClassNames` の配列に列挙してください。

カスタムツールバーの実装

ツールバーの拡張について

IM-BloomMaker のレイアウトモードでは、エレメントの編集機能を簡易化しているため、エレメントのプロパティエディタは表示されません。
プロパティエディタの代わりにツールバーが用意されています。
エレメントを選択すると、エレメントのプロパティ設定に応じて IM-BloomMaker が判断し、*基本的な編集機能* を自動的に用意した状態で、ツールバーが表示されます。
基本的な編集機能よりも、エレメントに対して高度な編集が必要な場合は、ツールバーの「操作」機能に操作メニューを実装する必要があります。

この章では、既存・自作のエレメントに対して、ツールバーに機能メニューを実装し、利用できるようにするまでの流れを説明します。

> **注意**
>
> ツールバーと操作メニューは、2024 Spring(Iris) 以降のバージョンで利用可能です。
> レイアウトモードでのみ使用可能です。 デベロップモードでは動作しません。

基本的な編集機能

ツールバーには、以下の基本的な編集機能が用意されています。

- フォントの変更
- テキストサイズの変更
- テキストの装飾（太字・斜体・下線・取り消し線）
- 横位置の設定（左寄せ・中央寄せ・右寄せ）
- 背景色、文字色の変更
- *アイコンの変更*
  - 固有プロパティの追加設定が必要です。
- 位置調整
- *テキストの直接編集*
  - 固有プロパティの追加設定が必要です。
- メモの追加・更新・削除
- 切り取り・コピー・貼り付け
- エレメントの削除

上記以外でエレメント固有のプロパティ値や、大きく見た目を変更する機能を付加したい場合は、操作メニューの実装が必要です。
「*事前準備*」から操作メニューを実装してください。

アイコンの変更

エレメントの固有プロパティの設定で *iconClass* を “font-awesome” に設定すると、エレメント選択時に表示されるツールバーに「アイコン」機能が追加されます。
エレメント固有プロパティの詳細については、「*固有プロパティを追加する*」を参照してください。

テキストの直接編集

エレメントの固有プロパティの設定で *directInput* を true に設定すると、エレメント選択時に表示されるツールバーに「テキスト編集」機能が追加されます。
エレメント固有プロパティの詳細については、「*固有プロパティを追加する*」を参照してください。

事前準備

自作のエレメントに対して操作メニューを実装する場合は、「*エレメント・アクションアイテムを作成する*」にしたがってエレメントを作成してください。
既存のエレメントに対して操作メニューを実装する場合は、「*事前準備*」にしたがって事前準備を行ってください。

Layouter を実装する

エレメント独自の操作メニューは、IUIElementLayouterExpandCaller インタフェースを持つクラスに実装していきます。

{VSCODE_HOME}/src/public/layouter に、操作メニューを実装するファイル（.tsファイル）を作成します。
ここでは、MyZipCodeFieldLayouter.ts というファイル名で作成しています。

このファイルに、IUIElementLayouterExpandCaller インタフェースを実装した、操作メニューのクラスを定義していきます。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

    public isSupported(element: IUIElement): boolean {
        // 引数が操作メニュー表示対象のエレメントであれば、true を返却する
        // ここでは、エレメントが MyZipCodeField であれば true を返却する
        return element.instance.elementTypeName === MyZipCodeField.name;
    }

    public initialize(element: IUIElement, container: IUIContainer): void {
        // ここに操作メニューのイベントハンドラを実装する
    }

}
```

Layouter をレポジトリに登録する

作成した Layouter クラスを UIElementLayouterExpandCallerRepository に登録します。
登録処理は {VSCODE_HOME}/src/index.ts に実装します。

```
const caller = window.imHichee.UIElementLayouterExpandCallerRepository;
if (caller) {
    caller.registerInstance(new MyZipCodeFieldLayouter());
}
```

以上で準備は完了です。

ツールバー拡張の実装

固有プロパティを変更する「部品操作」メニューを実装する

イベントリスナを実装する

エレメントを選択した際に表示されるツールバーの「操作」メニューをクリックすると、ツールバー上に表示されている各操作以外で、エレメントに関する全ての操作が可能なメニューが追加表示されます。
この操作メニューには、エレメントのプロパティ値を変更するために用意された「部品操作」グループと、エレメント内部を別のものに差し替えるために用意された「部品選択」グループがあります。

操作メニューの「部品操作」グループに、エレメント固有プロパティの変更機能を付与するためには、「*事前準備*」にて用意した Layouter クラスに、特定のイベントリスナを実装します。
実装するイベントリスナは、以下の通りです。

- designer-toolbar-show
  - 操作メニュー、または、ツールバーの表示が必要となったときに呼び出されるイベントです。
  - 操作メニューに表示する内容を返却します。
- designer-toolbar-hide
  - 操作メニュー、または、ツールバーが非表示になったときに呼び出されるイベントです。
  - 実装は任意です。
- designer-toolbar-push-button
  - designer-toolbar-show イベントで返却した操作メニュー、または、ツールバー上の項目が選択されたときに呼び出されるイベントです。
  - リクエストに従って、表示したメニューに応じた操作を、エレメントに対して行います。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // 操作メニュー・ツールバーが表示されたとき
    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      return {}; // メニューに表示する項目を返却する
    });

    // 操作メニュー・ツールバーが非表示になったとき（任意）
    eventListener.addEventListener(element, 'designer-toolbar-hide', () => {
      // 必要であれば、ツールバー非表示時の処理を実装する
    });

    // 操作メニュー・ツールバーから項目が選択されたとき
    eventListener.addEventListener(element, 'designer-toolbar-push-button', (eventValue) => {
      // エレメントの操作を実装する
    });
  }

}
```

自身のエレメントを操作するメニューを実装する

操作メニューに項目を追加する

操作メニューに項目を追加する場合は、イベントリスナで `designer-toolbar-show` のリスナを追加し、各項目の情報を返却します。
ここでは、郵便番号入力エレメントのプレースホルダ表示（placeholder プロパティの値）を切り替えるために、メニュー項目を追加します。
メニュー項目を特定するために、任意の ID を指定します。 この例では `placeholder` とします。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // 操作メニュー・ツールバーが表示されたとき
    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      // placeholder プロパティ値を取得する
      const placeholder = element.propertyAccessor.getProperty('placeholder').toBoolean();

      // メニューに表示する内容を返却する
      return {
        items: {
          // 「部品操作」グループ
          operations: [{
            // プレースホルダを示す任意の ID を指定する（後続のメニュー選択時の判別で使用）
            id: 'placeholder',
            // メニューの左側に表示するアイコンを指定する
            icon: placeholder
              ? [{type: 'font-awesome', value: 'fas fa-check'}]
              : [{type: 'font-awesome', value: 'fas im-empty'}],
            // メニューのラベルに表示する文字列を指定する
            label: 'プレースホルダを表示'
          }]
        }
      };
    });
  }

}
```

コラム

icon の指定について

- type に 'font-awesome' を指定した場合、value には Font Awesome のクラス名を指定します。
  value に im-empty を指定した場合、アイコン 1 つに相当する空白を挿入します。
- type に 'svg' を指定した場合、value には svg ファイルへのパスを指定します。
  パスは、Webサーバ上に配置したファイルにアクセスするための URL のうち、コンテキストパスより先を指定します。
  例えば URL が http://localhost:8080/imart/im_hichee/images/svg/foo.svg の場合、'im_hichee/images/svg/foo.svg' を指定します。
- icon は配列です。 アイコンの指定を複数指定できます。
  例えば独自クラスを css または less ファイルとして用意し、スタイルシートを指定することで、アイコンを重ねて表示するような自由な表現ができます。

項目が選択されたときの処理を実装する

メニュー項目を追加したら、メニュー項目が実際に選択されたときの処理を実装します。
メニュー選択時の処理は、イベントリスナで designer-toolbar-push-button のリスナを追加し、引数 eventValue の値を取得します。

最初に、eventValue.button が 'operation' であることを確認します。 これが、操作メニューの「部品操作」グループ内にある項目が選択されたことを示します。
次に eventValue.id を確認します。 ここには、選択されたメニューの ID が格納されています。 追加したメニュー項目の ID と一致する場合は、プレースホルダ表示（placeholder プロパティの値）を切り替える処理を実装します。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      // （省略）
    });

    // 操作メニューまたはツールバーから項目が選択されたとき
    eventListener.addEventListener(element, 'designer-toolbar-push-button', (eventValue) => {
      // 「部品操作」グループから項目が選択されたとき
      if (eventValue.button === 'operation') {
        // 「プレースホルダを表示」が選択されたとき
        if (eventValue.id === 'placeholder') {
          // placeholder プロパティ値を取得する
          const placeholder = element.propertyAccessor.getProperty('placeholder').toBoolean();
          // placeholder プロパティ値の真偽値を反転して、プロパティ値を再設定する
          element.propertyAccessor.setProperty('placeholder', !placeholder, true);
        }
      }
    });
  }

}
```

動作を確認する

デベロップモードで郵便番号入力エレメントを配置したコンテンツ定義を作成し、レイアウトモードに切り替えます。
郵便番号入力エレメントをクリックし、ツールバーから「操作」 -「部品操作」グループ -「郵便番号入力」 -「プレースホルダを表示」が表示されることを確認します。

「プレースホルダを表示」を選択すると、プレースホルダの表示が切り替わることを確認します。
また、再度「操作」メニューを表示した際に、「プレースホルダを表示」にチェックマークが表示、または、消去されることを確認します。

ツールバーの追加項目を実装する

イベントリスナを実装する

ツールバーに項目を追加する場合は、同じ機能を操作メニューの項目に追加する必要があります。
「*自身のエレメントを操作するメニューを実装する*」を参考に、イベントリスナを実装してください。

操作メニューをツールバーに表示する

ツールバーに項目を追加する

ツールバーに項目を追加する場合は、操作メニューに対して、ツールバーに表示するための情報を追加します。
ツールバーでは、アイコンの他に選択中を示すフラグ（active）を指定します。 このフラグを切り替えることで、ツールバーのアイコンをハイライト表示させることができます。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // 操作メニュー・ツールバーが表示されたとき
    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      // メニューに表示する内容を返却する
      return {
        items: {
          // 「部品操作」グループ
          operations: [{
            id: 'placeholder',
            icon: placeholder
              ? [{type: 'font-awesome', value: 'fas fa-check'}]
              : [{type: 'font-awesome', value: 'fas im-empty'}],
            label: 'プレースホルダを表示',
            // ツールバーに表示する内容を追加で返却する
            toolbar: {
              // プレースホルダが有効な場合、背景色をアクティブ色（灰色）に指定する
              active: placeholder,
              // ツールバー上のアイコンを指定する
              icon: [{type: 'font-awesome', value: 'fas fa-comment-dots'}],
              // ツールバー上のラベルに表示する文字列を指定する
              label: 'プレースホルダ'
            }
          }]
        }
      };
    });
  }

}
```

> **注意**
>
> 操作メニューを表示せずに、ツールバーのみに項目を表示することはできません。

項目が選択されたときの処理を実装する

「*固有プロパティを変更する「部品操作」メニューを実装する*」で実装した「項目が選択されたときの処理を実装する」と同じです。
すでに実装済みの場合は、そのまま流用できます。

動作を確認する

デベロップモードで郵便番号入力エレメントを配置したコンテンツ定義を作成し、レイアウトモードに切り替えます。
郵便番号入力エレメントをクリックし、ツールバーに「プレースホルダ」が表示されることを確認します。（表示エリア幅の関係上、ラベルは「プレース...」と表示されます）

「プレースホルダ」をクリックすると、プレースホルダの表示が切り替わることを確認します。
また、再度「プレースホルダ」をクリックした際に、「プレースホルダ」の背景色が変わることを確認します。

エレメント内部を差し替える「部品選択」メニューを実装する

イベントリスナを実装する

エレメントを選択した際に表示されるツールバーの「操作」メニューをクリックすると、ツールバー上に表示されている各操作以外で、エレメントに関する全ての操作が可能なメニューが追加表示されます。
この操作メニューには、エレメントのプロパティ値を変更するために用意された「部品操作」グループと、エレメント内部を別のものに差し替えるために用意された「部品選択」グループがあります。

操作メニューの「部品選択」グループに、エレメント内部を別のものに差し替える機能を付与するためには、「*事前準備*」にて用意した Layouter クラスに、特定のイベントリスナを実装します。

実装するイベントリスナは、「*固有プロパティを変更する「部品操作」メニューを実装する*」と同一です。

- designer-toolbar-show
  - 操作メニュー、または、ツールバーの表示が必要となったときに呼び出されるイベントです。
  - 操作メニューに表示する内容を返却します。
- designer-toolbar-push-button
  - designer-toolbar-show イベントで返却した操作メニュー、または、ツールバー上の項目が選択されたときに呼び出されるイベントです。
  - リクエストに従って、表示したメニューに応じた操作を、エレメントに対して行います。

ただし、designer-toolbar-show で返却する内容と、designer-toolbar-push-button で受け取る引数の値が異なります。

操作メニューをツールバーに表示する

ツールバーに項目を追加する

操作メニューに項目を追加する場合は、イベントリスナで designer-toolbar-show のリスナを追加し、各項目の情報を返却します。
サンプルの郵便番号入力エレメントには、表示するボックス形式を変更する機能はありませんが、仮に「テキストボックス形式」「選択ボックス形式」を選択できるよう設定してみます。
メニュー項目を特定するために、任意の ID を指定します。 この例では textbox および selectbox とします。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // 操作メニュー・ツールバーが表示されたとき
    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      // メニューに表示する内容を返却する
      return {
        items: {
          // 「部品選択」グループ
          addChildElements: [{
            id: 'textbox',
            label: 'テキストボックス形式'
          }, {
            id: 'selectbox',
            label: '選択ボックス形式'
          }]
        }
      };
    });
  }

}
```

項目が選択されたときの処理を実装する

メニュー項目を追加したら、メニュー項目が実際に選択されたときの処理を実装します。
メニュー選択時の処理は、イベントリスナで designer-toolbar-push-button のリスナを追加し、引数 eventValue の値を取得します。

最初に、eventValue.button が 'addChildElement' であることを確認します。 これが、操作メニューの「部品選択」グループ内にある項目が選択されたことを示します。
次に eventValue.id を確認します。 ここには、選択されたメニューの ID が格納されています。 追加したメニュー項目の ID と一致する場合は、エレメント内を差し替える操作を実装します。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    eventListener.addEventListener(element, 'designer-toolbar-show', () => {
      // （省略）
    });

    // 操作メニューまたはツールバーから項目が選択されたとき
    eventListener.addEventListener(element, 'designer-toolbar-push-button', (eventValue) => {
      // 「部品選択」グループから項目が選択されたとき
      if (eventValue.button === 'addChildElement') {
        // 「テキストボックス形式」が選択されたとき
        if (eventValue.id === 'textbox') {
          // TODO: ここにテキストボックスへ差し替える処理を実装
        }
        // 「選択ボックス形式」が選択されたとき
        if (eventValue.id === 'selectbox') {
          // TODO: ここに選択ボックスへ差し替える処理を実装
        }
      }
    });
  }

}
```

動作を確認する

デベロップモードで郵便番号入力エレメントを配置したコンテンツ定義を作成し、レイアウトモードに切り替えます。
郵便番号入力エレメントをクリックし、ツールバーから「操作」 - 「部品選択」グループ - 「郵便番号入力」を選択すると、「テキストボックス形式」と「選択ボックス形式」が表示されることを確認します。

部品を差し替える処理を実装した場合は、メニューを選択したときに実際に差し変わることを確認します。

テーブルエディタを実装する

イベントリスナを実装する

ツールバー以外に、テーブルを編集するための専用バー（テーブルエディタ）が用意されています。
このテーブルエディタでは、以下の機能を提供します。

- 各行・各列の全体選択バー
- 各行・各列の間に挿入するアイコン
  - 空行・空列の追加
  - 前行・前列と同じ内容を追加
- 各行・各列を削除するアイコン
- 各行・各列の大きさを変更

郵便番号入力エレメントはテーブルではありませんが、説明のためにテーブルエディタを実装してみます。
特定のエレメントに対してテーブルエディタを使用可能とするためには、「*事前準備*」にて用意した Layouter クラスに、特定のイベントリスナを実装します。
実装するイベントリスナは、以下の通りです。

- `designer-get-table-size`
  - インラインメニュー、または、ツールバーの表示が必要となったときに呼び出されるイベントです。
  - このイベントを実装し、行・列の各サイズ情報を返却すると、エディタ上にテーブル用の行列エディタが表示されます。
  - テーブルの行列を操作する機能を付与したい場合は、このイベントは必須です。 機能を使用しない場合は、このイベントは不要です。
- `designer-set-table-size`
  - テーブルの行列を操作する機能において、行・列のサイズが変更されたときに呼び出されるイベントです。

    - リクエストに従って、行・列のサイズを更新する操作を、エレメントに対して行います。
- designer-toolbar-push-table
    - designer-get-table-size イベントで許可した操作において、ツールバー上の項目が選択されたときに呼び出されるイベントです。
    - リクエストに従って、表示した項目に応じた操作を、エレメントに対して行います。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // テーブル用の行列エディタが表示されたとき
    eventListener.addEventListener(element, 'designer-get-table-size', async () => {
      return {columns: [], rows: []}; // テーブルの各行列サイズを返却する
    });

    // テーブル用の行列エディタで、サイズが変更されたとき
    eventListener.addEventListener(element, 'designer-set-table-size', (eventValue) => {
      // テーブルの各行列サイズを設定する
    });

    // テーブル用のツールバーから項目が選択されたとき
    eventListener.addEventListener(element, 'designer-toolbar-push-table', (eventValue) => {
      // テーブル行列の操作を実装する
    });
  }

}
```

テーブルの各行列サイズを返却する

テーブルエディタに、テーブルの基本的な情報を与えるため、イベントリスナで designer-get-table-size のリスナを追加し、各行・各列の情報を返却します。
今回はサンプルとして、列の情報のみを返却してみます。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    // テーブルエディタが表示されたとき
    eventListener.addEventListener(element, 'designer-get-table-size', async () => {
      // エレメントの表示位置を取得する
      const rect = element.getClientRect();

      // テーブルの各行列サイズを返却する
      return {
        // 列
        columns: [
          // position には、列の堺となる場所（単位：px）を指定します。
          // supported には、テーブルエディタに表示する機能を指定します。 表示されるだけで、実際に動作しません。
          // サイズ変更を動作させるためには、designer-set-table-size イベントを実装します。
          // 行・列の追加・削除を動作させるためには、designer-toolbar-push-table イベントを実装します。
          {position: rect.left}, // 左上の位置
          {position: rect.left + 100, supported: {changeWidth: true, cloneColumn: true, insertColumn: true, removeColumn: true}},
          {position: rect.left + 200, supported: {changeWidth: true, cloneColumn: true, insertColumn: true, removeColumn: true}}
        ],
        // 行
        rows: [
          {position: rect.top} // 左上の位置
        ]
      };
    });
  }

}
```

テーブルの各行列サイズを更新する

テーブルエディタで表示されている行・列の堺にある三角マークをドラッグすると、サイズの変更が開始されます。
ドラッグ操作が終了したときに、最終的に決定したサイズを取得するため、イベントリスナで `designer-set-table-size` のリスナを追加し、新しいサイズの情報を取得します。
サイズを取得したら、実際にエレメント上で行・列のサイズを変更する実装を行います。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    eventListener.addEventListener(element, 'designer-get-table-size', async () => {
      // （省略）
    });

    // テーブルエディタで行列のサイズが変更されたとき
    eventListener.addEventListener(element, 'designer-set-table-size', (eventValue) => {
      // テーブルの列サイズの変更要求があったとき
      if (eventValue.direction === 'column') {
        const index = eventValue.index; // 列番号
        const size = eventValue.size;   // 変更後のサイズ（単位：px）
        // TODO: ここに、列サイズを変更する処理を実装する
      }
      // テーブルの行サイズの変更要求があったとき
      if (eventValue.direction === 'row') {
        // （省略）
      }
    });
  }

}
```

テーブルの行列を追加・削除する

「*テーブルの各行列サイズを返却する*」で、行・列の複製や追加・削除を許可した場合、行・列選択バーをクリックするとツールバーが表示されます。
このツールバー上のアイコンをクリックしたとき、テーブルの行・列編集操作を行うため、イベントリスナで `designer-toolbar-push-table` のリスナを追加し、選択された操作種別を取得します。
操作種別を取得したら、実際にテーブルの操作をする実装を行います。

以下のように実装します。

```
export class MyZipCodeFieldLayouter implements IUIElementLayouterExpandCaller {

  public isSupported(element: IUIElement): boolean {
    return element.instance.elementTypeName === MyZipCodeField.name;
  }

  public initialize(element: IUIElement, container: IUIContainer): void {
    const eventListener = container.eventListener;

    eventListener.addEventListener(element, 'designer-get-table-size', async () => {
      // （省略）
    });

    eventListener.addEventListener(element, 'designer-set-table-size', (eventValue) => {
      // （省略）
    });

    // テーブルエディタのツールバーから項目が選択されたとき
    eventListener.addEventListener(element, 'designer-toolbar-push-table', (eventValue) => {
      const index = eventValue.index; // 行・列番号

      if (eventValue.button === 'addColumn') {
        // 「列を追加」アイコンが選択されたとき
        // TODO: ここに列を末尾に追加する処理を実装する
      } else if (eventValue.button === 'cloneColumn') {
        // 「列を複製」アイコンが選択されたとき
        // TODO: ここに index の位置にある列の右隣りに、同じ内容を複製する処理を実装する
      } else if (eventValue.button === 'insertColumn') {
        // 「列を挿入」アイコンが選択されたとき
        // TODO: ここに index の位置にある列の右隣りに、空列を追加する処理を実装する
      } else if (eventValue.button === 'removeColumn') {
        // 「列を削除」アイコンが選択されたとき
        // TODO: ここに index の位置にある列を削除する処理を実装する
      } else if (eventValue.button === 'moveColumnLeft') {
        // 「列を左に移動」アイコンが選択されたとき
        // TODO: ここに index の位置にある列の左隣りに、列を移動する処理を実装する
      } else if (eventValue.button === 'moveColumnRight') {
        // 「列を右に移動」アイコンが選択されたとき
        // TODO: ここに index の位置にある列の右隣りに、列を移動する処理を実装する
      } else if (eventValue.button === 'addRow') {
        // 「行を追加」アイコンが選択されたとき
        // TODO: ここに行を末尾に追加する処理を実装する
      } else if (eventValue.button === 'cloneRow') {
        // 「行を複製」アイコンが選択されたとき
        // TODO: ここに index の位置にある行の下隣りに、同じ内容を複製する処理を実装する
      } else if (eventValue.button === 'insertRow') {
        // 「行を挿入」アイコンが選択されたとき
        // TODO: ここに index の位置にある行の下隣りに、空列を追加する処理を実装する
      } else if (eventValue.button === 'removeRow') {
        // 「行を削除」アイコンが選択されたとき
        // TODO: ここに index の位置にある行を削除する処理を実装する
      } else if (eventValue.button === 'moveRowUp') {
        // 「行を上に移動」アイコンが選択されたとき
        // TODO: ここに index の位置にある行の上隣りに、行を移動する処理を実装する
      } else if (eventValue.button === 'moveRowDown') {
        // 「行を下に移動」アイコンが選択されたとき
        // TODO: ここに index の位置にある行の下隣りに、行を移動する処理を実装する
      }
    });
  }

}
```

動作を確認する

デベロップモードで郵便番号入力エレメントを配置したコンテンツ定義を作成し、レイアウトモードに切り替えます。
郵便番号入力エレメントをクリックすると、テーブルエディタが表示されます。 ツールバーに隠れて見えない場合は、ツールバーをドラッグ＆ドロップして、移動させてください。

行・列の堺にある三角アイコンをドラッグすると、三角アイコンが上下または左右に移動することを確認します。
実際に行・列のサイズを変更する実装を行った場合、三角アイコンを移動させて、実際に行・列のサイズが変更されることを確認します。

行・列選択バーをクリックすると、追加のツールバーが表示されることを確認します。
実際に行・列の編集操作をする実装を行った場合、ツールバー上のアイコンをクリックして、実際に行・列の編集操作が行われることを確認します。

IM-BloomMakerが提供する標準のエレメントのクラス名と提供開始バージョン一覧

「レイアウト」カテゴリ

エレメント名	クラス名	提供開始バージョン
ボックス	Box	2019 Summer(Waltz)
フレックスコンテナ	FlexBox	2019 Summer(Waltz)
テーブル（レイアウト）	SimpleLayoutTable	2019 Summer(Waltz)
リスト（レイアウト）	LayoutUnorderedList	2019 Summer(Waltz)
連番付きリスト（レイアウト）	LayoutOrderedList	2019 Summer(Waltz)
見出しレベル1	Heading1	2019 Summer(Waltz)
見出しレベル2	Heading2	2019 Summer(Waltz)
見出しレベル3	Heading3	2019 Summer(Waltz)
見出しレベル4	Heading4	2019 Summer(Waltz)
見出しレベル5	Heading5	2019 Summer(Waltz)
見出しレベル6	Heading6	2019 Summer(Waltz)
段落	Paragraph	2019 Summer(Waltz)
サイドメニューコンテナ	ImSideMenuContainer	2020 Summer(Zephirine)

「繰り返し」カテゴリ

エレメント名	クラス名	提供開始バージョン
ボックス（繰り返し）	RepeatBox	2019 Summer(Waltz)
インラインフレックス（繰り返し）	RepeatInlineFlex	2019 Winter(Xanadu)
テーブル（繰り返し）	SimpleDataTable	2019 Summer(Waltz)
リスト（繰り返し）	DataUnorderedList	2019 Summer(Waltz)
連番付きリスト（繰り返し）	DataOrderedList	2019 Summer(Waltz)

「フォーム部品」カテゴリ

エレメント名	クラス名	提供開始バージョン
フォーム	Form	2019 Summer(Waltz)
テキスト入力	InputText	2019 Summer(Waltz)
テキストエリア	TextArea	2019 Summer(Waltz)
パスワード入力	InputPassword	2019 Summer(Waltz)
数値入力	InputNumber	2019 Summer(Waltz)
数値入力（フォーマット）	ImInputFormattedNumber	2022 Winter(Freesia)
ラジオボタン	InputRadio	2019 Summer(Waltz)
チェックボックス	InputCheckBox	2019 Summer(Waltz)
トグルスイッチ	InputToggleSwitch	2021 Summer(Cattleya)
ファイル選択	InputFile	2019 Summer(Waltz)
日付入力	ImInputDate	2019 Summer(Waltz)

エレメント名	クラス名	提供開始バージョン
時刻入力	ImInputTime	2022 Winter(Freesia)
プルダウン	SelectBox	2019 Summer(Waltz)
複数選択	ListBox	2019 Summer(Waltz)
コンボボックス	ImComboBox	2019 Summer(Waltz)
リッチテキストボックス	RichTextBox	2019 Summer(Waltz)
色選択	InputColor	2019 Summer(Waltz)
隠しパラメータ	InputHidden	2019 Summer(Waltz)
ボタン	Button	2019 Summer(Waltz)
送信ボタン	InputSubmit	2019 Summer(Waltz)
ファイルアップロード	ImFileUpload	2020 Spring(Yorkshire)
マルチファイルアップロード	ImMultipleFileUpload	2021 Spring(Bergamot)
リッチテーブル	ImRichTable	2020 Spring(Yorkshire)

「共通マスタ」カテゴリ

エレメント名	クラス名	提供開始バージョン
単一選択ユーザ検索	ImMasterAutocompletesSingleUserSearch	2021 Spring(Bergamot)
複数選択ユーザ検索	ImMasterAutocompletesMultipleUserSearch	2021 Spring(Bergamot)
所属組織による単一選択ユーザ検索	ImMasterAutocompletesSingleUserDeptSearch	2021 Spring(Bergamot)
所属組織による複数選択ユーザ検索	ImMasterAutocompletesMultipleUserDeptSearch	2021 Spring(Bergamot)
単一選択会社検索	ImMasterAutocompletesSingleCompanySearch	2021 Spring(Bergamot)
複数選択会社検索	ImMasterAutocompletesMultipleCompanySearch	2021 Spring(Bergamot)
単一選択組織検索	ImMasterAutocompletesSingleDepartmentSearch	2021 Spring(Bergamot)
複数選択組織検索	ImMasterAutocompletesMultipleDepartmentSearch	2021 Spring(Bergamot)

「汎用」カテゴリ

エレメント名	クラス名	提供開始バージョン
ハイパーリンク	Anchor	2019 Summer(Waltz)
ラベル	Label	2019 Summer(Waltz)
強調ラベル	Strong	2019 Summer(Waltz)
入力規則エラーメッセージ	ImValidationErrorMessage	2021 Spring(Bergamot)
マークダウン	ImMarkdown	2024 Autumn(Jasmine)

「パーツ」カテゴリ

エレメント名	クラス名	提供開始バージョン
画像埋め込み	Image	2019 Summer(Waltz)
コンテナページ埋め込み	ImContainerPageFrame	2020 Winter(Azalea)
インラインフレーム	InlineFrame	2019 Summer(Waltz)
水平罫線	HorizontalRule	2019 Summer(Waltz)
進捗バー	ProgressBar	2019 Summer(Waltz)
動画埋め込み	Video	2019 Summer(Waltz)

エレメント名	クラス名	提供開始バージョン
音声埋め込み	Audio	2019 Summer(Waltz)
外部リソース埋め込みコンテナ	ImObjectContainer	2021 Spring(Bergamot)

「その他」カテゴリ

エレメント名	クラス名	提供開始バージョン
タイマー	Timer	2019 Summer(Waltz)
排他制御	ImSharedResource	2021 Winter(Dandelion)

「レイアウト（imui）」カテゴリ

エレメント名	クラス名	提供開始バージョン
見出しレベル1	ImTitleH1	2019 Summer(Waltz)
見出しレベル1（小窓用）	ImTitleSmallWindowH1	2019 Summer(Waltz)
見出しレベル2	ImChapterTitleH2	2019 Summer(Waltz)
見出しレベル3	ImSectionTitleH3	2019 Summer(Waltz)
見出しレベル4	ImSubsectionTitleH4	2019 Summer(Waltz)
見出しレベル5	ImParagraphTitleH5	2019 Summer(Waltz)
見出しレベル6	ImSubparagraphTitleH6	2019 Summer(Waltz)
フォームコンテナ	ImFormContainer	2019 Summer(Waltz)
アコーディオン	ImAccordion	2019 Winter(Xanadu)
オペレーションボックス	ImOperationBox	2019 Summer(Waltz)
ツールボックス	ImToolBox	2019 Summer(Waltz)
補足ボックス	ImSupplementation	2019 Summer(Waltz)
アイコン付きリスト	ImOperationList	2019 Summer(Waltz)
レイアウト調整ボックス	ImBoxLayout	2019 Summer(Waltz)
バレットリスト	ImBulletedList	2019 Summer(Waltz)
アイコン付きリンクリスト	ImLinkItemMenu	2019 Summer(Waltz)
横方向のテーブル	ImTableHorizontal	2019 Summer(Waltz)
縦方向のテーブル	ImTableVertical	2019 Summer(Waltz)
見出し付きテーブル	ImTableWithHeadline	2019 Summer(Waltz)
入れ子テーブル	ImTableInner	2019 Summer(Waltz)
横方向汎用テーブル	ImTableMixed	2019 Summer(Waltz)
入力フォーム用テーブル	ImTableForm	2019 Summer(Waltz)
検索条件用テーブル	ImTableSearchCondition	2019 Summer(Waltz)
パンくずリスト	ImBreadcrumbsWrap	2019 Summer(Waltz)

「繰り返し（imui）」カテゴリ

エレメント名	クラス名	提供開始バージョン
横方向のテーブル（繰り返し）	ImHorizontalDataTable	2019 Winter(Xanadu)
縦方向のテーブル（繰り返し）	ImVerticalDataTable	2019 Winter(Xanadu)
入れ子テーブル（繰り返し）	ImInnerDataTable	2019 Winter(Xanadu)

エレメント名	クラス名	提供開始バージョン
横方向汎用テーブル（繰り返し）	ImDataTableMixed	2019 Winter(Xanadu)
バレットリスト（繰り返し）	ImBulletedDataList	2019 Winter(Xanadu)
アイコン付きリンクリスト（繰り返し）	ImDataLinkItemMenu	2019 Winter(Xanadu)

「フォーム部品（imui）」カテゴリ

エレメント名	クラス名	提供開始バージョン
ボタン	ImButton	2019 Summer(Waltz)
オペレーションパーツ	ImOperationParts	2019 Summer(Waltz)
アイコン付きボタン	ImIconButton	2020 Summer(Zephirine)
日付入力	ImCalendar	2019 Summer(Waltz)

「パーツ（imui）」カテゴリ

エレメント名	クラス名	提供開始バージョン
アイコン	ImInlineIcon	2019 Summer(Waltz)

「レイアウト（Bulma）」カテゴリ

エレメント名	クラス名	提供開始バージョン
ボックス	ImBulmaBox	2020 Summer(Zephirine)
カラム	ImBulmaColumns	2020 Summer(Zephirine)
コンテナ	ImBulmaContainer	2020 Summer(Zephirine)
水平コンテナ	ImBulmaLevel	2020 Summer(Zephirine)
中央揃え水平コンテナ	ImBulmaCenteredLevel	2020 Summer(Zephirine)
テーブルコンテナ	ImBulmaTableContainer	2020 Summer(Zephirine)
メディアオブジェクト	ImBulmaMediaObject	2020 Summer(Zephirine)
ヒーロー	ImBulmaHero	2020 Summer(Zephirine)
セクション	ImBulmaSection	2020 Summer(Zephirine)
フッタ	ImBulmaFooter	2020 Summer(Zephirine)
タイル	ImBulmaTiles	2020 Summer(Zephirine)
コンテンツ	ImBulmaContent	2020 Summer(Zephirine)
見出しレベル1	ImBulmaTitle1	2020 Summer(Zephirine)
見出しレベル2	ImBulmaTitle2	2020 Summer(Zephirine)
見出しレベル3	ImBulmaTitle3	2020 Summer(Zephirine)
見出しレベル4	ImBulmaTitle4	2020 Summer(Zephirine)
見出しレベル5	ImBulmaTitle5	2020 Summer(Zephirine)
見出しレベル6	ImBulmaTitle6	2020 Summer(Zephirine)

「繰り返し（Bulma）」カテゴリ

エレメント名	クラス名	提供開始バージョン
カラム（繰り返し）	ImBulmaRepeatColumns	2020 Summer(Zephirine)
テーブルコンテナ（繰り返し）	ImBulmaRepeatableTableContainer	2020 Summer(Zephirine)

エレメント名	クラス名	提供開始バージョン
フィールドエレメント配置アイテム（繰り返し）	ImBulmaRepeatableFieldDivision	2021 Spring(Bergamot)

「フォーム部品（Bulma）」カテゴリ

エレメント名	クラス名	提供開始バージョン
フィールド	ImBulmaField	2020 Summer(Zephirine)
水平フィールド	ImBulmaHorizontalField	2020 Summer(Zephirine)
テキスト入力	ImBulmaInputTextControl	2020 Summer(Zephirine)
数値入力	ImBulmaInputNumberControl	2020 Summer(Zephirine)
数値入力（フォーマット）	ImBulmaInputFormattedNumberControl	2022 Winter(Freesia)
パスワード入力	ImBulmaInputPasswordControl	2020 Summer(Zephirine)
メールアドレス入力	ImBulmaInputEmailControl	2020 Summer(Zephirine)
電話番号入力	ImBulmaInputTelControl	2020 Summer(Zephirine)
日付入力	ImBulmaInputDateControl	2020 Winter(Azalea)
日付入力（カレンダー）	ImBulmaCalendarControl	2025 Autumn(Lilac)
時刻入力	ImBulmaInputTimeControl	2022 Winter(Freesia)
テキストエリア	ImBulmaTextareaControl	2020 Summer(Zephirine)
プルダウン	ImBulmaSelectControl	2020 Summer(Zephirine)
複数選択	ImBulmaListBoxControl	2020 Summer(Zephirine)
チェックボックス	ImBulmaCheckboxControl	2020 Summer(Zephirine)
ラジオボタン	ImBulmaRadioControl	2020 Summer(Zephirine)
テキスト表示	ImBulmaDisplayTextControl	2021 Spring(Bergamot)
ボタン	ImBulmaButtonControl	2020 Summer(Zephirine)
削除ボタン	ImBulmaDelete	2020 Summer(Zephirine)
エレメント配置アイテム	ImBulmaFieldDivision	2021 Spring(Bergamot)

「パーツ（Bulma）」カテゴリ

エレメント名	クラス名	提供開始バージョン
画像埋め込み	ImBulmaImage	2020 Summer(Zephirine)
進捗バー	ImBulmaProgressBar	2020 Summer(Zephirine)
アイコン	ImBulmaFontAwesomeIcon	2020 Summer(Zephirine)
タグ	ImBulmaTag	2020 Summer(Zephirine)
通知ボックス	ImBulmaNotification	2020 Summer(Zephirine)

「コンポーネント（Bulma）」カテゴリ

エレメント名	クラス名	提供開始バージョン
パンくずリストコンテナ	ImBulmaBreadcrumb	2020 Summer(Zephirine)
ページネーション	ImBulmaPagination	2020 Summer(Zephirine)
ナビゲーションバー	ImBulmaNavbar	2020 Summer(Zephirine)
ナビゲーションバーリンクアイテム	ImBulmaNavbarItemAnchor	2020 Summer(Zephirine)
ナビゲーションバーエレメント配置アイテム	ImBulmaNavbarItemDivision	2020 Summer(Zephirine)

エレメント名	クラス名	提供開始バージョン
ナビゲーションバードロップダウンアイテム	ImBulmaNavbarItemHasDropdown	2020 Summer(Zephirine)
ナビゲーションバー水平罫線	ImBulmaNavbarDivider	2020 Summer(Zephirine)
メッセージ	ImBulmaMessage	2020 Summer(Zephirine)
タブ	ImBulmaTabs	2020 Summer(Zephirine)
タブセット	ImBulmaTabsSet	2026 Spring(Mimosa)
カード	ImBulmaCard	2020 Summer(Zephirine)
ドロップダウン	ImBulmaDropdown	2020 Summer(Zephirine)
ドロップダウンリンクアイテム	ImBulmaDropdownItemAnchor	2020 Summer(Zephirine)
ドロップダウンエレメント配置アイテム	ImBulmaDropdownItemDivision	2020 Summer(Zephirine)
ドロップダウン水平罫線アイテム	ImBulmaDropdownItemDivider	2020 Summer(Zephirine)
メニュー	ImBulmaMenu	2020 Summer(Zephirine)

「グラフ」カテゴリ

エレメント名	クラス名	提供開始バージョン
折れ線グラフ	LineChart	2019 Winter(Xanadu)
棒グラフ	BarChart	2019 Winter(Xanadu)
円グラフ	PieChart	2019 Winter(Xanadu)
レーダーチャート	RadarChart	2019 Winter(Xanadu)
散布図	ScatterPlot	2019 Winter(Xanadu)

「バーコード」カテゴリ

エレメント名	クラス名	提供開始バージョン
バーコードスキャナ	ImBarcodeScanner	2021 Summer(Cattleya)
QRコードスキャナ	ImQRCodeScanner	2021 Summer(Cattleya)

「レイアウト（imds）」カテゴリ

エレメント名	クラス名	提供開始バージョン
ボックス	ImdsBox	2024 Spring(Iris)
カラム	ImdsColumns	2024 Spring(Iris)
コンテナ	ImdsContainer	2024 Spring(Iris)
水平コンテナ	ImdsLevel	2024 Spring(Iris)
中央揃え水平コンテナ	ImdsCenteredLevel	2024 Spring(Iris)
テーブルコンテナ	ImdsTableContainer	2024 Spring(Iris)
メディアオブジェクト	ImdsMediaObject	2024 Spring(Iris)
ヒーロー	ImdsHero	2024 Spring(Iris)
セクション	ImdsSection	2024 Spring(Iris)
フッタ	ImdsFooter	2024 Spring(Iris)
タイル	ImdsTiles	2024 Spring(Iris)
コンテンツ	ImdsContent	2024 Spring(Iris)
ヘッダ	ImdsHeader	2024 Spring(Iris)

エレメント名	クラス名	提供開始バージョン
見出しレベル1	ImdsTitle1	2024 Spring(Iris)
見出しレベル2	ImdsTitle2	2024 Spring(Iris)
見出しレベル3	ImdsTitle3	2024 Spring(Iris)
見出しレベル4	ImdsTitle4	2024 Spring(Iris)
見出しレベル5	ImdsTitle5	2024 Spring(Iris)
見出しレベル6	ImdsTitle6	2024 Spring(Iris)

「繰り返し（imds）」カテゴリ

エレメント名	クラス名	提供開始バージョン
カラム（繰り返し）	ImdsRepeatColumns	2024 Spring(Iris)
テーブルコンテナ（繰り返し）	ImdsRepeatableTableContainer	2024 Spring(Iris)
フィールドエレメント配置アイテム（繰り返し）	ImdsRepeatableFieldDivision	2024 Spring(Iris)

「フォーム部品（imds）」カテゴリ

エレメント名	クラス名	提供開始バージョン
フォームコンテナ	ImdsFormContainer	2024 Spring(Iris)
リッチフォームコンテナ	ImdsRichFormContainer	2024 Spring(Iris)
フィールドグループ	ImdsFieldGroup	2024 Spring(Iris)
インプットグループ	ImdsInputGroup	2024 Spring(Iris)
テキスト入力	ImdsInputTextControl	2024 Spring(Iris)
数値入力	ImdsInputNumberControl	2024 Spring(Iris)
数値入力（フォーマット）	ImdsInputFormattedNumberControl	2024 Spring(Iris)
パスワード入力	ImdsInputPasswordControl	2024 Spring(Iris)
メールアドレス入力	ImdsInputEmailControl	2024 Spring(Iris)
電話番号入力	ImdsInputTelControl	2024 Spring(Iris)
日付入力	ImdsInputDateControl	2024 Spring(Iris)
日付入力（カレンダー）	ImdsCalendarControl	2025 Autumn(Lilac)
時刻入力	ImdsInputTimeControl	2024 Spring(Iris)
テキストエリア	ImdsTextareaControl	2024 Spring(Iris)
プルダウン	ImdsSelectControl	2024 Spring(Iris)
複数選択	ImdsListBoxControl	2024 Spring(Iris)
チェックボックス	ImdsCheckboxControl	2024 Spring(Iris)
ラジオボタン	ImdsRadioControl	2024 Spring(Iris)
トグルスイッチ	ImdsToggleSwitch	2024 Autumn(Jasmine)
テキスト表示	ImdsDisplayTextControl	2024 Spring(Iris)
ボタン	ImdsButtonControl	2024 Spring(Iris)
削除ボタン	ImdsDelete	2024 Spring(Iris)
エレメント配置アイテム	ImdsFieldDivision	2024 Spring(Iris)

「パーツ（imds）」カテゴリ

エレメント名	クラス名	提供開始バージョン
画像埋め込み	ImdsImage	2024 Spring(Iris)
進捗バー	ImdsProgressBar	2024 Spring(Iris)
進捗サークル	ImdsProgressCircular	2024 Autumn(Jasmine)
アイコン	ImdsFontAwesomeIcon	2024 Spring(Iris)
タグ	ImdsTag	2024 Spring(Iris)
通知ボックス	ImdsNotification	2024 Spring(Iris)
水平罫線	ImdsHorizontalDivider	2024 Autumn(Jasmine)

「コンポーネント（imds）」カテゴリ

エレメント名	クラス名	提供開始バージョン
パンくずリストコンテナ	ImdsBreadcrumb	2024 Spring(Iris)
ページネーション	ImdsPagination	2024 Spring(Iris)
ナビゲーションバー	ImdsNavbar	2024 Spring(Iris)
ナビゲーションバーリンクアイテム	ImdsNavbarItemAnchor	2024 Spring(Iris)
ナビゲーションバーエレメント配置アイテム	ImdsNavbarItemDivision	2024 Spring(Iris)
ナビゲーションバードロップダウンアイテム	ImdsNavbarItemHasDropdown	2024 Spring(Iris)
ナビゲーションバー水平罫線	ImdsNavbarDivider	2024 Spring(Iris)
メッセージ	ImdsMessage	2024 Spring(Iris)
タブ	ImdsTabs	2024 Spring(Iris)
タブセット	ImdsTabsSet	2026 Spring(Mimosa)
カード	ImdsCard	2024 Spring(Iris)
ドロップダウン	ImdsDropdown	2024 Spring(Iris)
ドロップダウンリンクアイテム	ImdsDropdownItemAnchor	2024 Spring(Iris)
ドロップダウンエレメント配置アイテム	ImdsDropdownItemDivision	2024 Spring(Iris)
ドロップダウン水平罫線アイテム	ImdsDropdownItemDivider	2024 Spring(Iris)
メニュー	ImdsMenu	2024 Spring(Iris)
ステッパー	ImdsStepper	2024 Autumn(Jasmine)
アコーディオングループ	ImdsAccordionGroup	2024 Autumn(Jasmine)
空状態コンテナ	ImdsEmptyStateContainer	2024 Autumn(Jasmine)

「IM-Copilot」カテゴリ

エレメント名	クラス名	提供開始バージョン
アシスタント実行	ImCopilotAssistant	2024 Autumn(Jasmine)

IM-BloomMakerが提供する標準のアクションアイテムのクラス名と提供開始バージョン一覧

「標準」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
フォーム○を送信する	SubmitFormActionItem	2019 Summer(Waltz)

アクションアイテム名	クラス名	提供開始バージョン
変数○に○を代入する	UIContainerVariableSetActionItem	2019 Summer(Waltz)
変数○の値をクリップボードにコピーする	CopyVariableActionItem	2020 Winter(Azalea)
ラベル○へジャンプする	JumpNextLabelActionItem	2019 Summer(Waltz)
ラベル○	JumpTargetLabelActionItem	2019 Summer(Waltz)
アクション○を実行する	RunSubroutineActionItem	2019 Summer(Waltz)
URL○にリクエストを送信する	SendSimpleAjaxActionItem	2019 Summer(Waltz)
URL○に遷移する	PageTransitionActionItem	2019 Winter(Xanadu)
カスタムスクリプトを実行する	UIContainerCustomScriptActionItem	2019 Summer(Waltz)
ページ○をダイアログで開く	UIDialogShowActionItem	2019 Summer(Waltz)
最前面に表示されているダイアログを閉じる	UIDialogCloseActionItem	2019 Summer(Waltz)
ページ○を開く	ShowContainerPageActionItem	2019 Summer(Waltz)
変数○に一覧データ○から選択したものを代入する	TableSelectActionItem	2019 Summer(Waltz)
変数○に一覧データ○から選択したものを複数代入する	MultipleTableSelectActionItem	2021 Summer(Cattleya)
メッセージ○をアラートダイアログで表示する	ImShowAlertDialogActionItem	2020 Summer(Zephirine)
メッセージ○を確認ダイアログで表示する	ImShowConfirmDialogActionItem	2020 Summer(Zephirine)
メッセージ○をエラーダイアログで表示する	ImShowErrorDialogActionItem	2020 Summer(Zephirine)
入力規則エラーを非表示にする	ImHideValidationErrorActionItem	2020 Summer(Zephirine)
入力規則エラーを表示する	ImShowValidationErrorActionItem	2020 Summer(Zephirine)
変数○に○の各キー名を配列にして代入する	ImUIContainerVariableKeysActionItem	2022 Spring(Eustoma)
変数○に○の各要素の値を配列にして代入する	ImUIContainerVariableValuesActionItem	2022 Spring(Eustoma)
音声または動画○を再生する	ImPlayMediaElementActionItem	2022 Spring(Eustoma)
音声または動画○を一時停止する	ImPauseMediaElementActionItem	2022 Spring(Eustoma)
音声または動画○を再生・一時停止する	ImPlayAndPauseMediaElementActionItem	2022 Spring(Eustoma)
表示中の画面を閉じる	CloseWindowActionItem	2020 Spring(Yorkshire)
URL○をポップアップウィンドウで表示する	ImOpenPopupWindowActionItem	2022 Spring(Eustoma)
ウィンドウ名○のポップアップウィンドウを閉じる	ImClosePopupWindowActionItem	2022 Spring(Eustoma)
親画面の変数○に○を代入する	ImOpenerUIContainerVariableSetActionItem	2022 Spring(Eustoma)
変数○のタイムゾーンを○のタイムゾーンに変更する	ImTimeZoneConversionActionItem	2023 Autumn(Hollyhock)
エレメント○の位置へ遷移する	ImInPageTransitionActionItem	2024 Autumn(Jasmine)
エレメント○にフォーカスを設定する	ImSetFocusActionItem	2025 Spring(Kamille)
空処理	ImEmptyActionItem	2023 Autumn(Hollyhock)

「共通マスタ」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
単一選択ユーザ検索ダイアログを表示する	ImMasterSingleUserSearch	2019 Summer(Waltz)
複数選択ユーザ検索ダイアログを表示する	ImMasterMultipleUserSearch	2019 Summer(Waltz)
単一選択組織検索ダイアログを表示する	ImMasterSingleDepartmentSearch	2019 Summer(Waltz)
複数選択組織検索ダイアログを表示する	ImMasterMultipleDepartmentSearch	2019 Summer(Waltz)
単一選択役職検索ダイアログを表示する	ImMasterSinglePostSearch	2019 Summer(Waltz)
複数選択役職検索ダイアログを表示する	ImMasterMultiplePostSearch	2019 Summer(Waltz)
単一選択パブリックグループ検索ダイアログを表示する	ImMasterSinglePublicGroupSearch	2020 Spring(Yorkshire)
複数選択パブリックグループ検索ダイアログを表示する	ImMasterMultiplePublicGroupSearch	2020 Spring(Yorkshire)
単一選択ロール検索ダイアログを表示する	ImMasterSingleRoleSearch	2020 Winter(Azalea)
複数選択ロール検索ダイアログを表示する	ImMasterMultipleRoleSearch	2020 Winter(Azalea)

「IM-LogicDesigner」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
IM-LogicDesigner フロールーティング○にリクエストを送信する	SendRequestImLogicFlowRouteActionItem	2021 Spring(Bergamot)

「ViewCreator」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
ViewCreator ルーティング○にリクエストを送信する	SendRequestViewCreatorRouteActionItem	2021 Summer(Cattleya)

「imui」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
メッセージ○を表示する	ImuiShowSuccessMessageActionItem	2019 Summer(Waltz)
エラーメッセージ○を表示する	ImuiShowErrorMessageActionItem	2019 Summer(Waltz)
警告メッセージ○を表示する	ImuiShowWarningMessageActionItem	2019 Summer(Waltz)

注意

2020 Summer(Zephirine) より前のバージョンでは、「メッセージ○を表示する」「エラーメッセージ○を表示する」「警告メッセージ○を表示する」は「標準」カテゴリ内に表示されます。

「Bulma」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
ページ○をモーダルで開く	ImBulmaShowModalActionItem	2020 Summer(Zephirine)
ページ○をモーダルカードで開く	ImBulmaShowModalCardActionItem	2020 Summer(Zephirine)
モーダルを閉じる	ImBulmaCloseModalActionItem	2020 Summer(Zephirine)
メッセージ◯を表示する	ImBulmaShowToastActionItem	2021 Winter(Dandelion)

「imds」カテゴリ

アクションアイテム名	クラス名	提供開始バージョン
メッセージ○を確認ダイアログで表示する	ImdsShowConfirmActionItem	2024 Autumn(Jasmine)
ページ○をダイアログで開く	ImdsShowDialogActionItem	2024 Autumn(Jasmine)
ダイアログを閉じる	ImdsCloseDialogActionItem	2024 Autumn(Jasmine)
ページ○をモーダルで開く	ImdsShowModalActionItem	2024 Spring(Iris)
ページ○をモーダルカードで開く	ImdsShowModalCardActionItem	2024 Spring(Iris)
モーダルを閉じる	ImdsCloseModalActionItem	2024 Spring(Iris)
メッセージ◯を表示する	ImdsShowToastActionItem	2024 Spring(Iris)