Copyright © 2019 NTT DATA INTRAMART CORPORATION

↑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 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。

前処理プログラム

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");
```

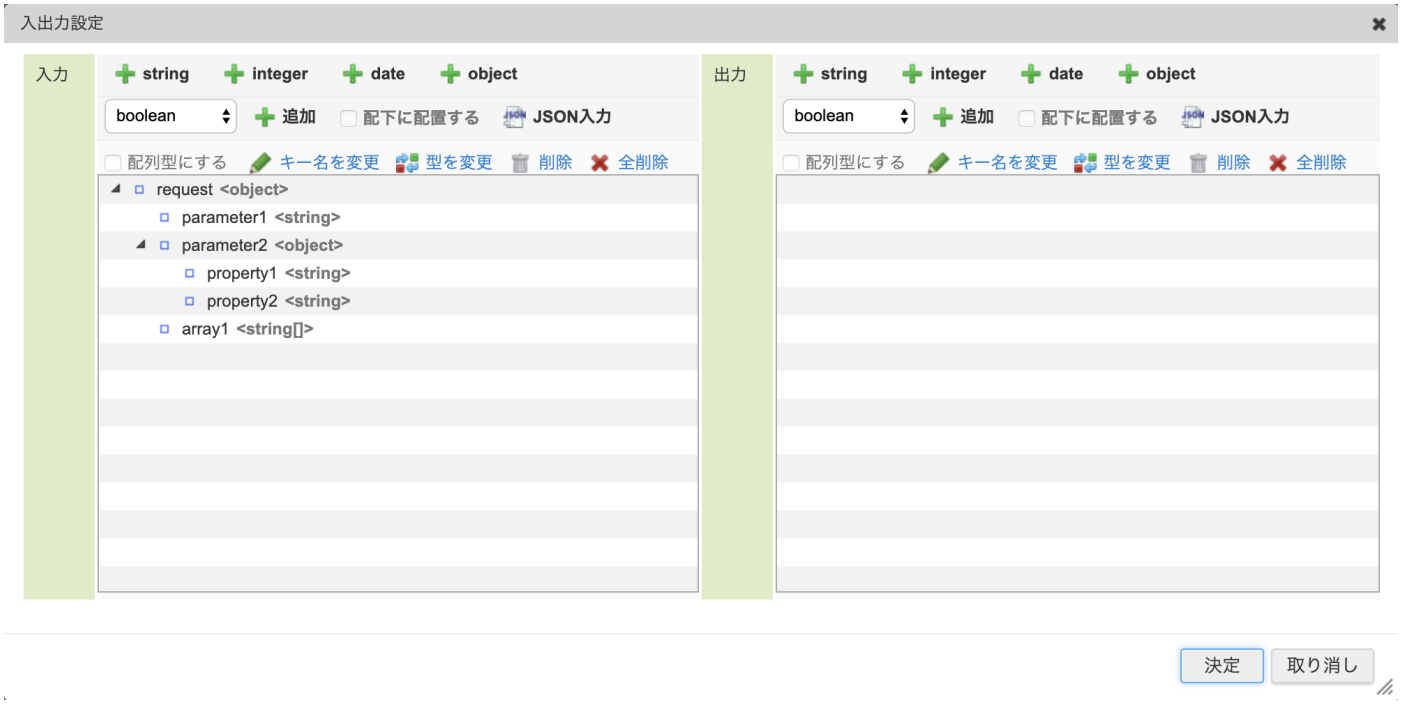
のように取得できます。

JavaScript の前処理プログラムの場合は

```
function execute(context) {
   let parameter1 = context.parsedRequestParameters.parameter1;
   let parameter2 = context.parsedRequestParameters.parameter2;
   let 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。

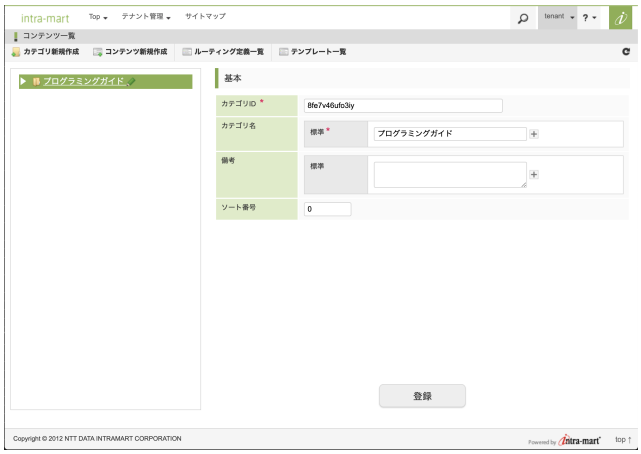
前処理のサンプル実装

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。

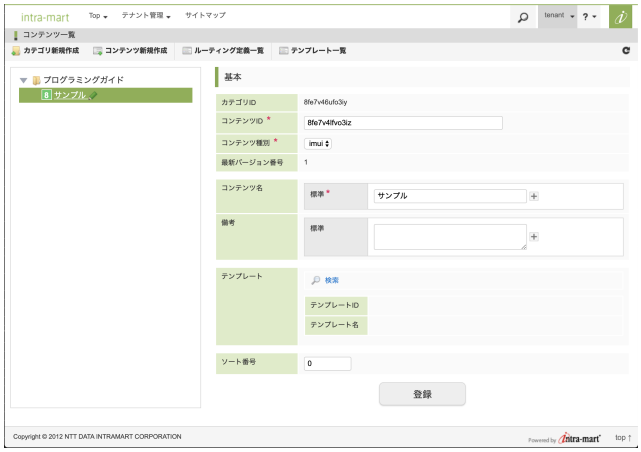
コンテンツの作成

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。

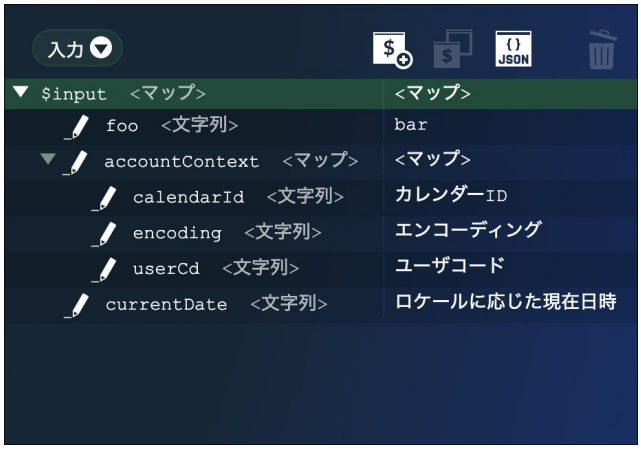


次に作成した「プログラミングガイド」カテゴリを選択した状態で「コンテンツ新規作成」リンクをクリックします。 画面右側の「コンテンツ名」に「サンプル」と入力し、「登録」ボタンをクリック、「登録確認」ダイアログで「決定」ボタンをクリックします。

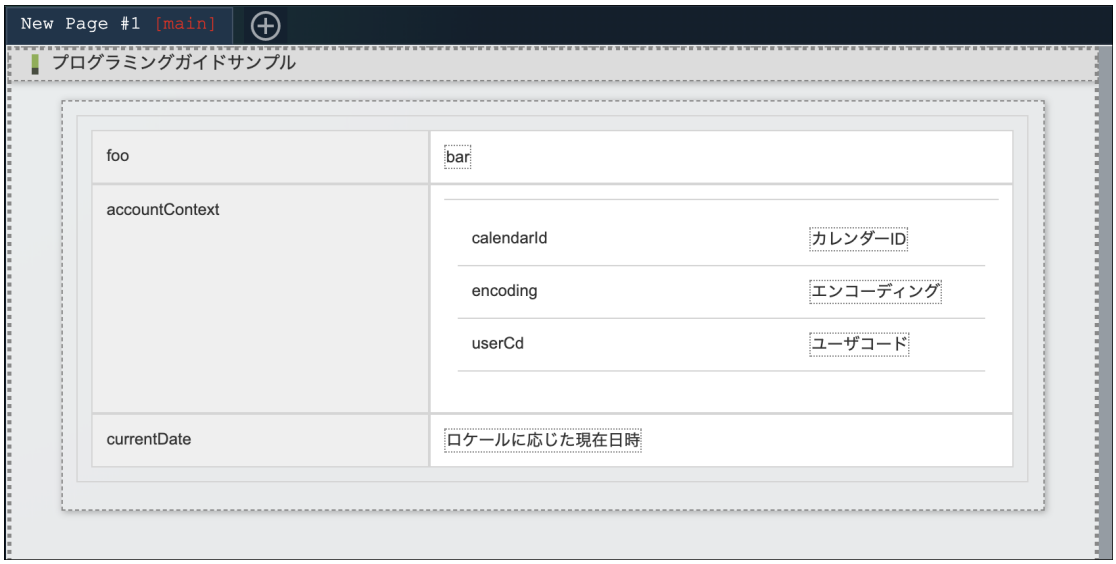


「デザイン編集」ボタンをクリックし、デザイナ画面を開きます。

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。

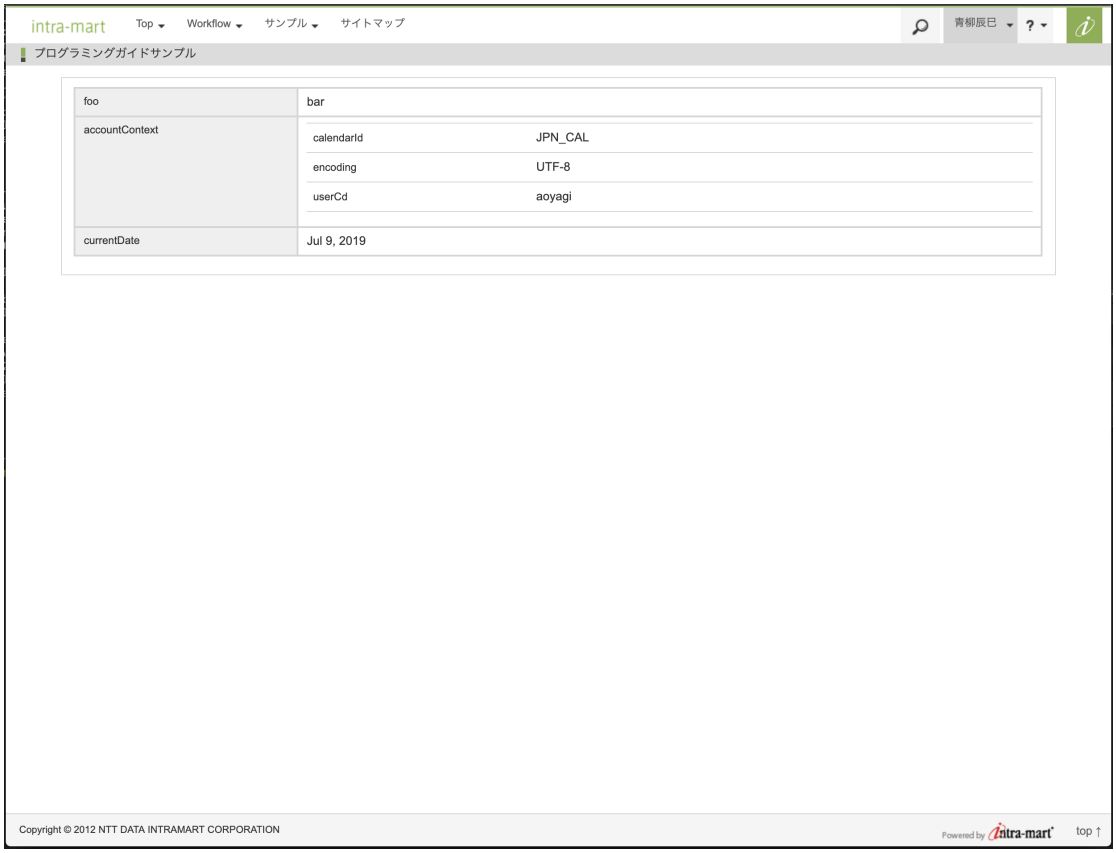
Java による前処理の実装

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。

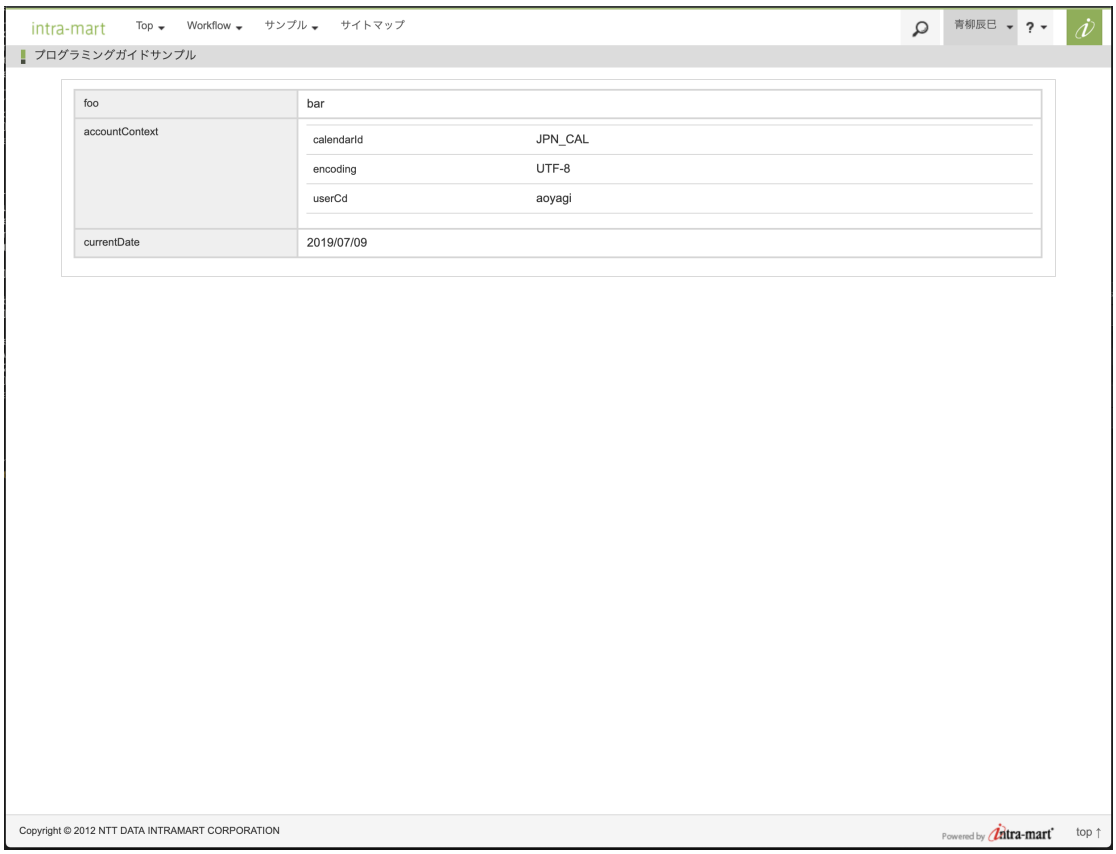
```
public class PreProcessor implements BMContentPreprocessor {

    @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

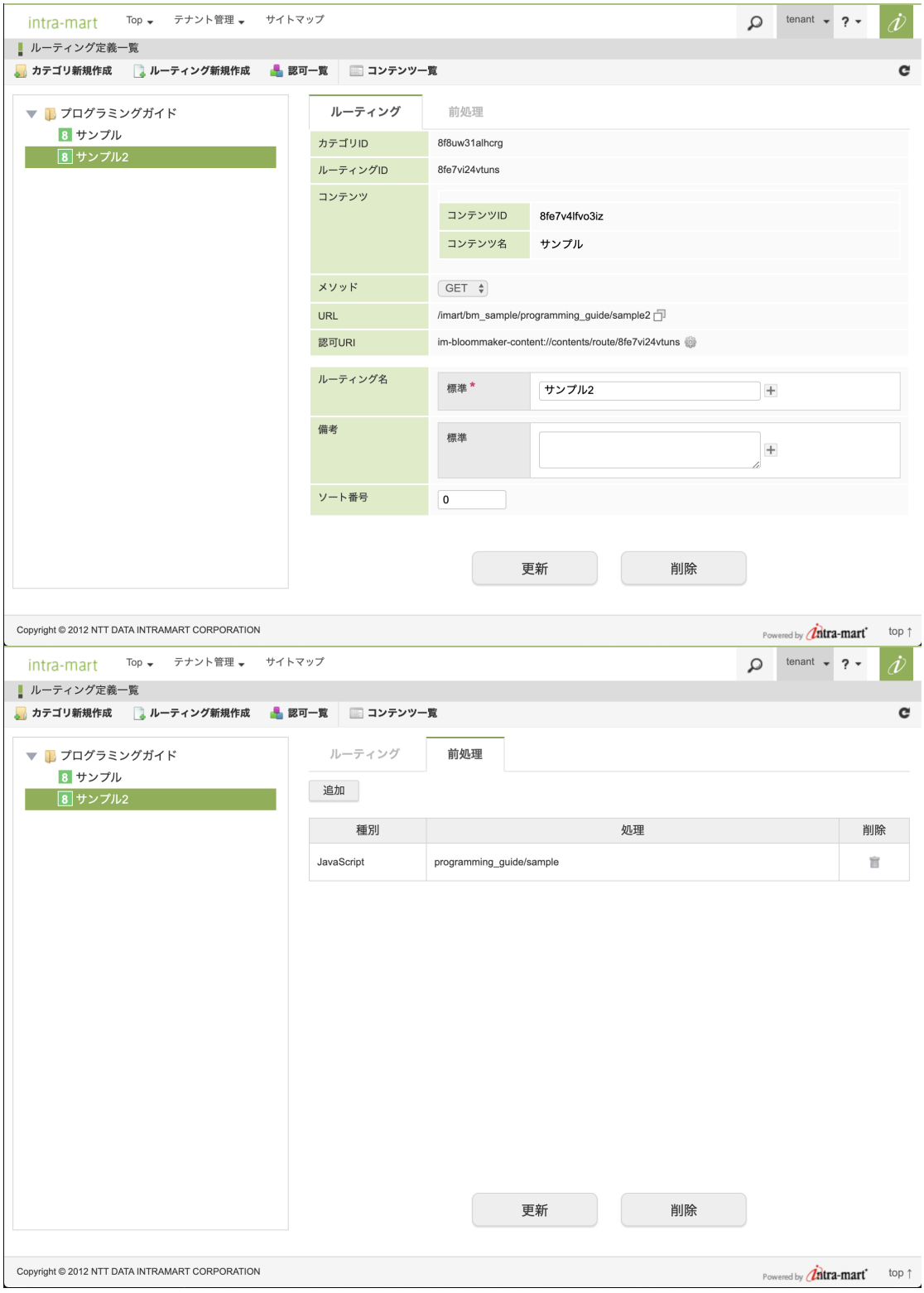
上記の Java による前処理プログラムを JavaScript（スクリプト開発モデル）でも実装してみます。

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。

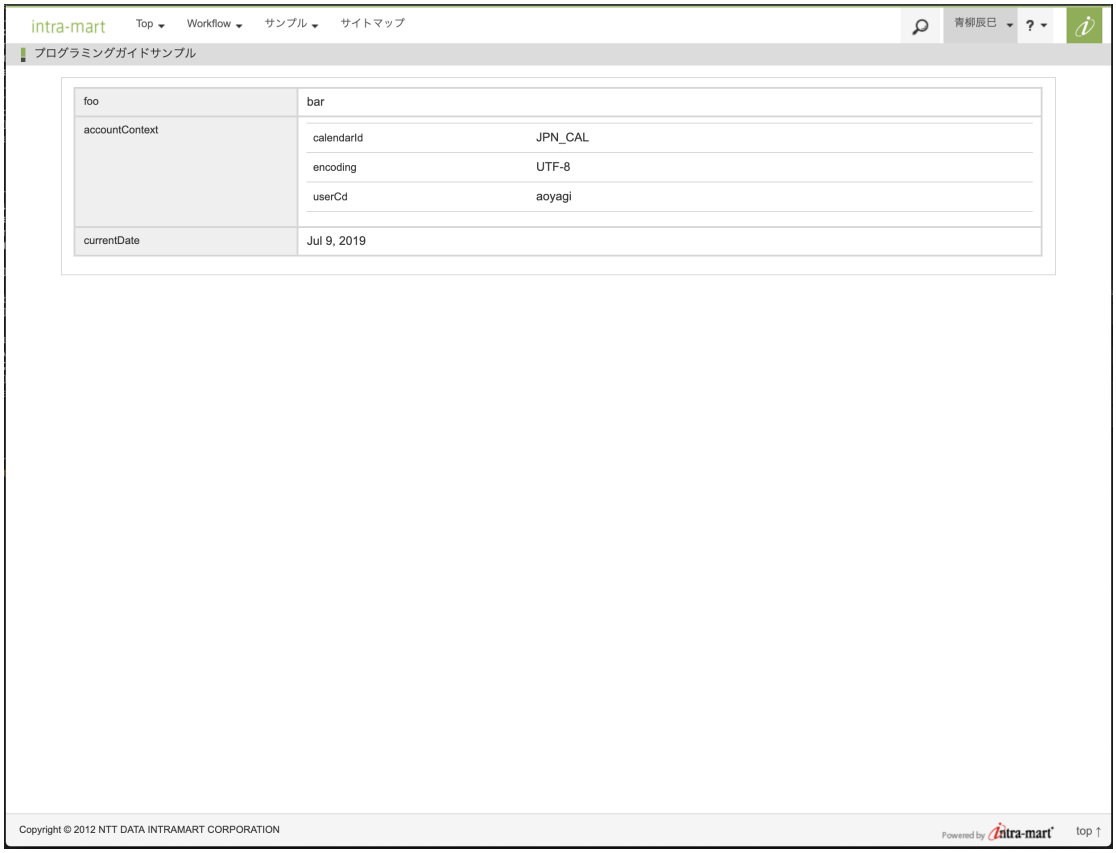
```
function execute(context) {

  // 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。

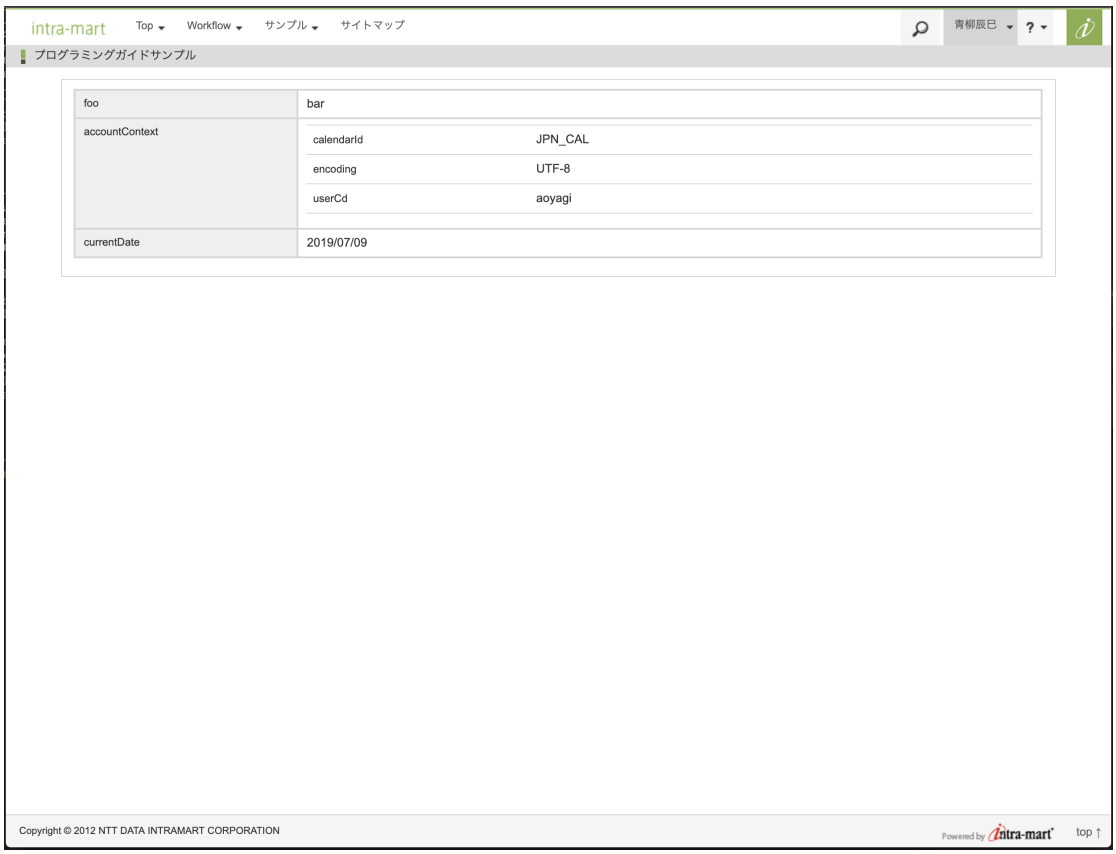
ルーティングは以下のように定義します。



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": ""
 }
}
```

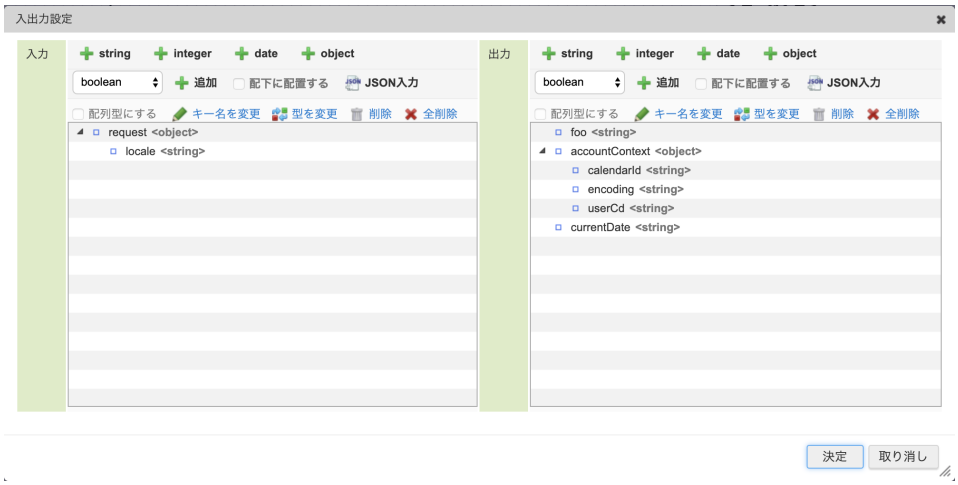
出力

```
{
 "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。

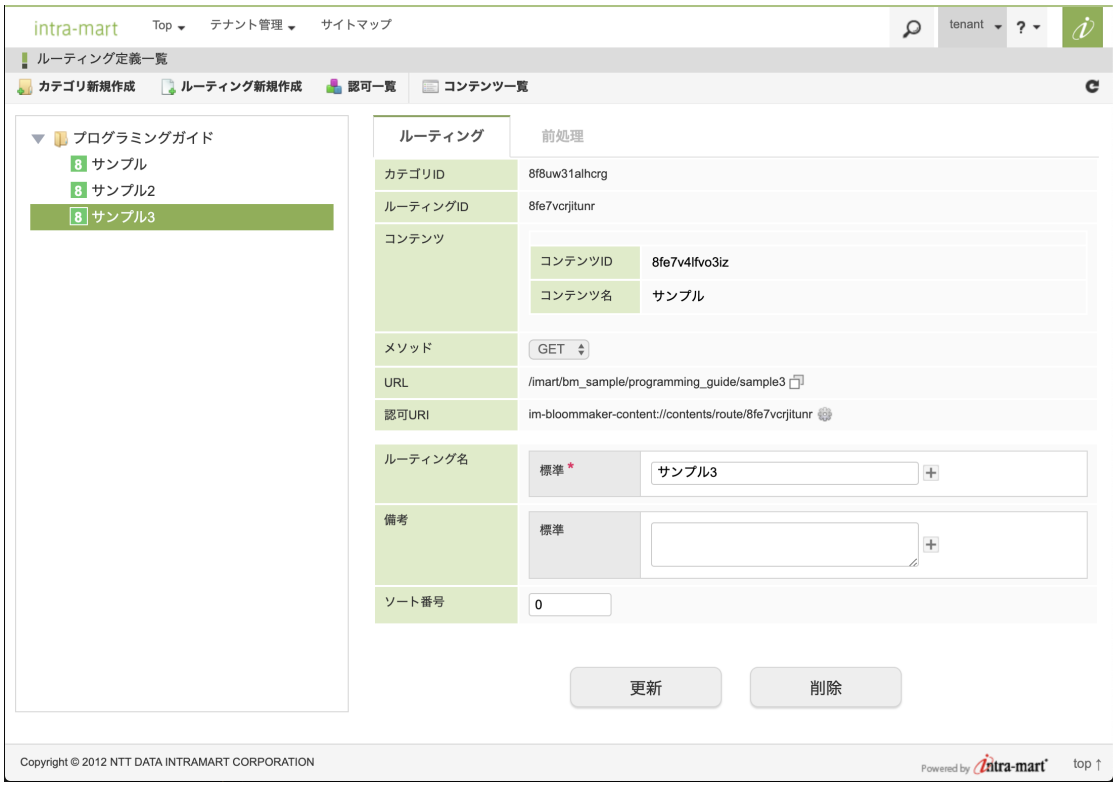
最後にマッピング設定を行います。

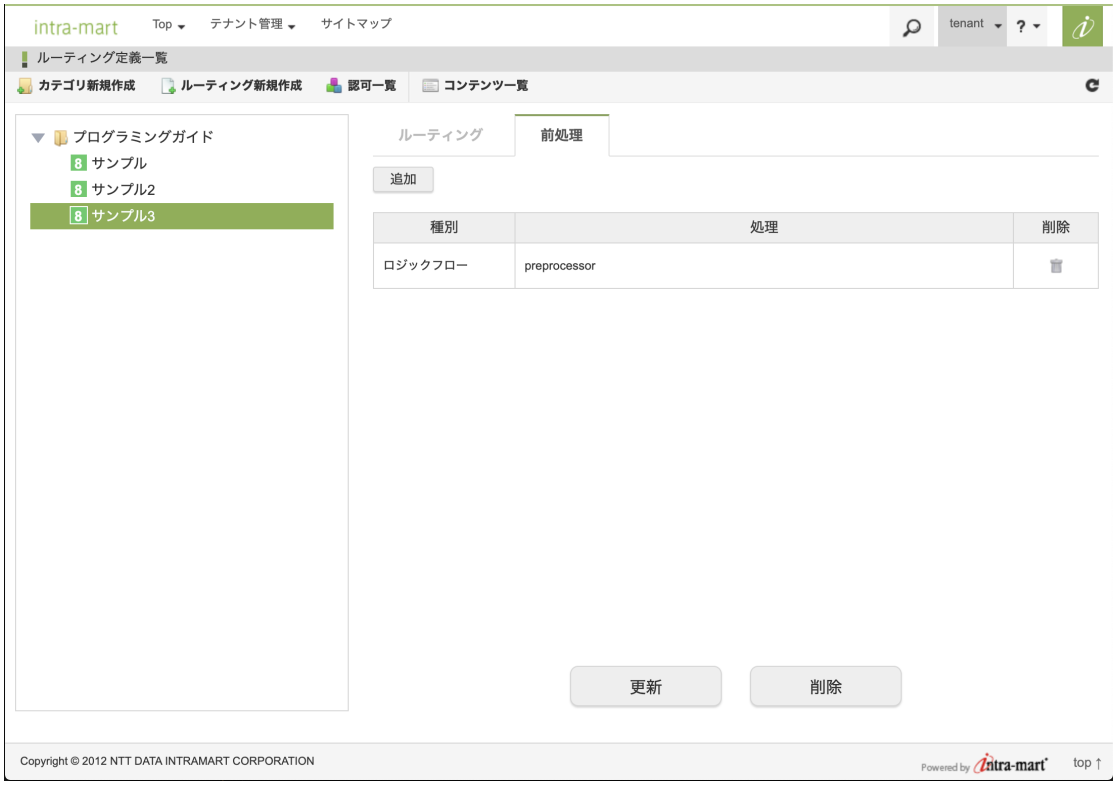
preprocessor のマッピング定義は以下のように定義します。

終了のマッピング定義は以下のように定義します。

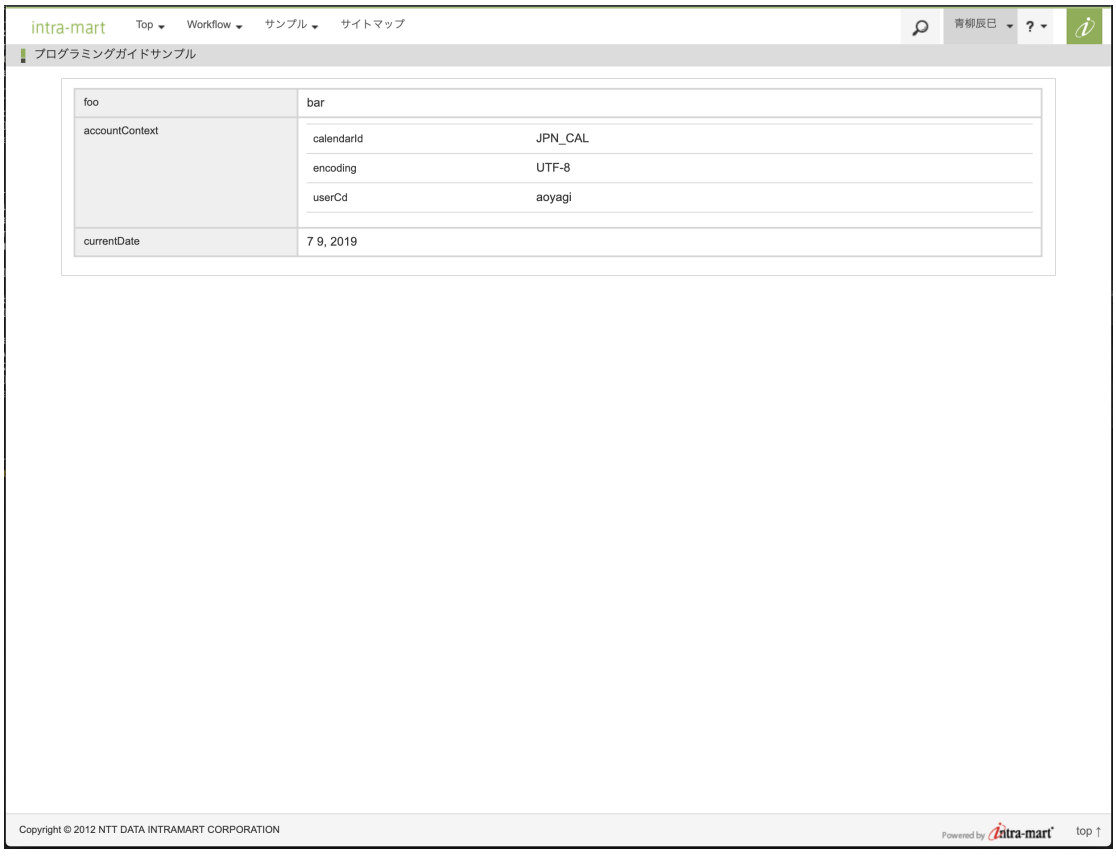
新規保存します。フロー定義IDなど適当な値を入力してください。今回はフローIDなどを preprocessor とします。

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。

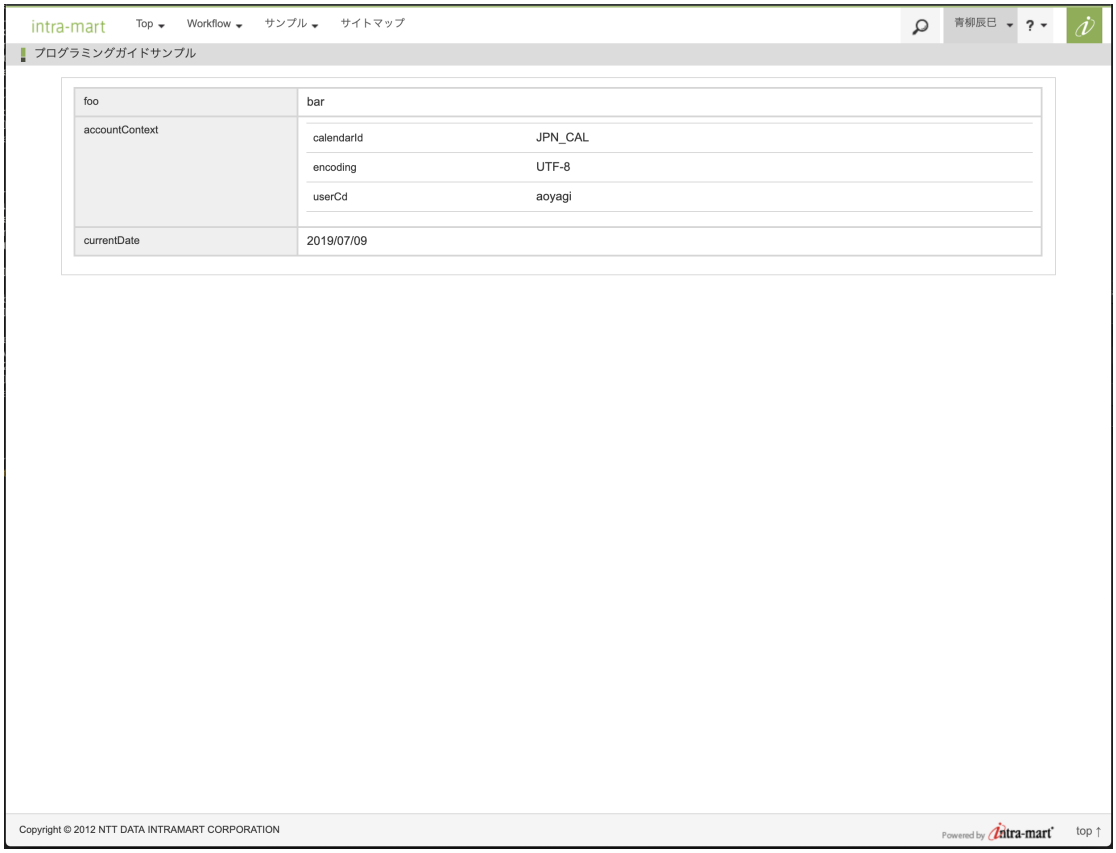




ルーティングに指定した URL にアクセスすると、以下のように表示されます。



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の作成

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 v12 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/ja/

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
>> v12.16.0
npm -v
>> 6.13.4
```

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。

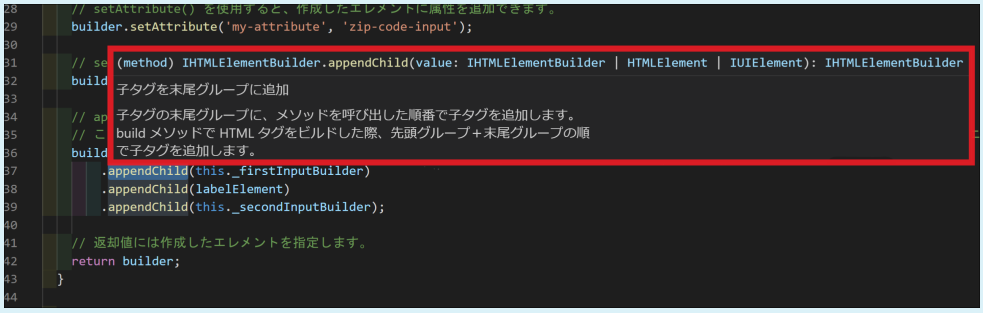
インストーラを以下からダウンロードし、インストールしてください。

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 *4.2.1.1 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.zip
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、*4.3 エレメントのサンプル実装* と *4.4 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。】
            │       └── 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": "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 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 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

4.2 エレメントを実装する / *4.4 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するメソッドです。
  // 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するメソッドです。
  // { 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するメソッドです。
  public get displayType(): UIElementDisplayType {
    // TODO
  }

  // 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
  }

  // 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
  }

  // 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-Hichee 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。
  // build() メソッドは自動で呼び出されるため、明示的に呼び出す必要はありません。
  // updateElement() で更新する必要のない処理は、ここに実装してください。
  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、その処理を記述します。
    public updateElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
    ): void {
        // TODO
    }

    // デザイナ上でエレメントが削除された際などに呼び出されるメソッドです。
    // 基本的には特に何の処理も記述しません。
    public destroyElement(
        builder: IHTMLElementBuilder,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): void {
        return;
    }

    // createChildren() でエレメントに子要素を追加できます。
    // 子要素がない場合は空の配列を返却します。
    // 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。
    // 例えば、rerender: trueのプロパティの値が変更されたときなど、エレメント側を更新する必要がある場合にその処理を記述します。
    // 必要な処理がない場合は、引数の children をそのまま返却します。
    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タイミングで特殊な採番処理等を行いたい場合などにその処理を記述します。
    // 基本的には何の処理も記述しません。
    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() の返却値には false を、
    // false を指定した場合は acceptableChild() の返却値には true を指定してください。
    public get extremity(): boolean {
        return true;
    }

    // エレメントにおける親要素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に親要素を配置できます。
    public acceptableParent(parent: ParentUIElement): boolean {
        return true;
    }

    // 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エレメントに対し子エレメントを配置できます。
    // このメソッドの返却値に true を指定した場合には extremity() の返却値には false を、
    // false を指定した場合には extremity() の返却値には true を指定してください。
    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。

```
// エレメントの表示形式を返却するメソッドです。
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。

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ー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。

```
// 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のビルダーを作成して返却します。
public createElement(
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<PropertyDefinition & ICommonPropertyDefinition>
): IHTMLElementBuilder {
    // エレメントのビルダーを作成するには、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ーを指定します。
    return builder;
}
```

> **コラム**
>
> **エレメントのビルダーを作成する方法について**
>
> createElement メソッドの返却値は、純粋なHTMLのエレメント（HTMLElement 型）ではなく、エレメントのビルダー（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* ）
> ビルダーを作成するには、上の例のように IHTMLElementBuilder.createElement() でタグ名からビルダーを作成するか、IHTMLElementBuilder.fromElement() を利用して純粋なHTMLのエレメントをビルダー化してください。
> 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);

    builder.setAttribute('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ーを生成します。
        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;

    /** 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。

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 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 {

    // 子要素のビルダーを、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);
    });

    // ビルダーを生成し、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ーを生成
    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。

```
// 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 {
    // 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ー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ー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: 120});

// 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');
```

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。
4.1.2.4 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(), updateChildren(), 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 updateChildren(
```

```
    children: IUIElement[],
    self: IUIElement,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
  ): IUIElement[] {
    return children; // 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。

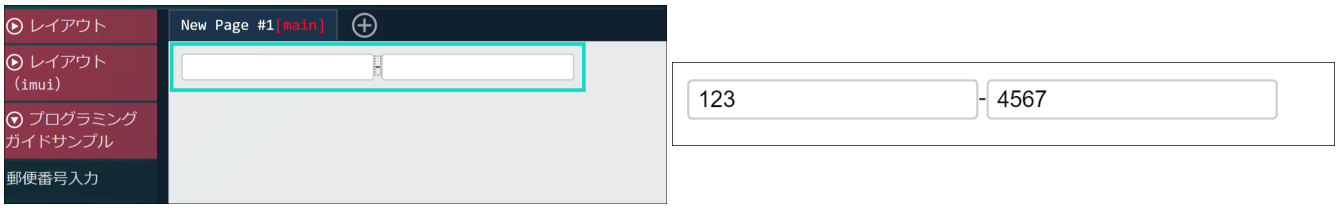
```
export class SampleElementClass extends SampleBaseElementCore { ... }
```

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するメソッドです。
  // { 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するメソッドです。
  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 {};
  }

  // 依存させたいエレメントのクラスを返却します。
  // ここで指定されたエレメントは、当該エレメントとの間に依存関係が定義され、自動的に定義が読み込まれます。
  // 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。
  // 返却値は子エレメントのクラス名です。
  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。
  // このメソッド内でエレメントのビルダーを作成して返却します。
  // 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。
  // 例えば、rerender: trueのプロパティの値が変更されたときなど、エレメント側を更新する必要がある場合にその処理を記述します。
  // 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
  }

  // デザイナ上でエレメントが削除された際などに呼び出されるメソッドです。
  // 基本的には特に何の処理も記述しません。
  public destroyElement(
    builder: IHTMLElementBuilder,
    container: IUIContainer,
    properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
  ): void {
    return;
  }

  // このメソッドでエレメントに子要素を追加できます。
  // このエレメントが作成された際に自動的に用意する子エレメントのインスタンスを返却してください。
  // 子エレメントは配列の格納順に配置されます。
  // 子要素がない場合は空の配列を返却します。
  // 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、都度呼び出されるメソッドです。
```

```
    // 引数で渡された子エレメントの配列を変更する必要がある場合、変更後の配列を返却してください。
    // 変更の必要がない場合は、引数の children をそのまま返却してください。
    public updateChildren(
        children: IUIElement[],
        self: IUIElement,
        container: IUIContainer,
        properties: IUIElementPropertyAccessor<IUniquePropertyDefinition & ICommonPropertyDefinition>
    ): IUIElement[] {
        return children;
    }

    // 仮想エレメントがクローンによって作成された際に呼び出されるメソッドです。
    // エレメントが複製されたタイミングで特殊な採番処理等を行いたい場合などに、その処理を記述します。
    // 引数で渡された子エレメントの配列を変更する必要がある場合、変更後の配列を返却してください。
    // 変更の必要がない場合は、引数の children をそのまま返却してください。
    // 基本的には変更の必要はありません。
    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。
    // true か false を返却値に指定します。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 か false を返却値に指定します。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 か false を返却値に指定します。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（配下への子エレメントの配置可否）を指定するメソッドです。
    // true か false を指定します。
    // 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を true に指定した場合は acceptableChild() の返却値を false 、逆に false を指定した場合は
acceptableChild() の返却値を true に指定するようにしてください。
    public get extremity(): boolean {
        return true;
    }

    // エレメントにおける親エレメントの配置可否を指定するメソッドです。
    // true か false を指定します。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に親エレメントを配置できます。
    public acceptableParent(parent: ParentUIElement): boolean {
        return true;
    }

    // 子エレメントの配置可否を指定するメソッドです。
    // true か false を指定します。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に対し子エレメントを配置できます。
    // このメソッドの返却値を true に指定した場合には extremity() の返却値を false 、逆に false を指定した場合には extremity() の返却
値を true に指定するようにしてください。
    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 を実装します。
PropertyDefinition タイプを宣言し、value 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 プロパティの定義を記述します。

```
type PropertyDefinition = {
    value: 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: {
        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,
        },
        type: 'string',
    },
};
```

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。

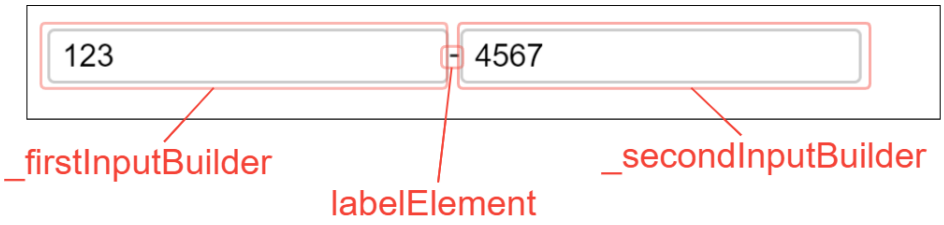
以下の処理を記述します。

- 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, setCSS メソッドを利用して、属性、CSSを付与する。

実装は以下の通りです。

```
export class MyZipCodeField implements IUIElementCore {
    // 子要素のビルダーは 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（ビルダー）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ビルダーを指定します。
    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 プロパティに `rerender: true` を設定しているため、value 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。

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 {
    // プロパティの値を文字列で取得します。
    const value = properties.getProperty('value').toString();

    // 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.substr(0, 3);
    const value2 = value.substr(3);

    // 子ノードの<input>タグのvalue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).build();
    this._secondInputBuilder.setValue(value2).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: 120});

// 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。
```

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
```

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。
4.1.2.4 バンドルの生成 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。