

[Dok](#) [WhatsApp Flows](#) [Reference](#) [Flow JSON](#)[Di Halaman Ini](#)[!\[\]\(cf531ed27e91483460120fcc057b3901\_img.jpg\) Platform WhatsApp Business](#)

# Flow JSON

Leverage Flow JSON to build your user experience.

To visualize the complete user experience, use the Builder. The Builder emulates the entire Flow experience and can be updated on the fly. To navigate to the Builder:

1. Choose your business
2. Click on **All tools** in the left side navigation
3. Open [WhatsApp Manager](#) and select your WABA
4. On the left side navigation, go to **Account tools > Flows**

See this [list of templates](#) you can build with Flow JSON.

## Introduction

Flow JSON enables businesses to create workflows in WhatsApp by accessing the features of WhatsApp Flows using a custom JSON object developed by Meta.

These workflows are initiated, run, and managed entirely inside WhatsApp. It can include multiple screens, data flows, and response messages.

Flow JSON consists of the following sections:

Flow JSON Section	Description
Screen Data Model	Commands to define static types that power the screen.
Screens	Used to compose layouts using standard UI library components.



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

Components	Individual building blocks that make up a screen (text fields, buttons, and so on).
Routing Model	Defines the rules for the screen by limiting the possible state transition. For example, developers can define that from Screen 1 you can only transition to Screen 2 and 3, but not Screens 4 and 5. These rules are used during server / client side payload validations.
Actions	Special type of syntax to invoke pre-defined logic on the client. Allowed actions are: <code>navigate</code> , <code>data_exchange</code> , or <code>complete</code> . From Flow JSON version 6.0 and later, <code>open_url</code> and <code>update_data</code> , is also allowed.

## Top-level Flow JSON Properties

Flow JSON has several required and optional properties that are used in the process of compilation and validation of the Flow.

### Required properties

- `version` - represents the version of Flow JSON to use during the compilation. Please refer to the [list of versions](#) for more details.
- `screens` - represents an array or screen as part of the user experience. This is like a set of different pages on your website.

### Optional properties

- `routing_model` - represents a routing ruling system. The routing model is generated automatically if your Flow doesn't use a Data Endpoint. If it does, the validation system will ask you to provide a routing model.
- `data_api_version` - represents the version to use during communication with the WhatsApp Flows Data Endpoint. Currently, it is **3.0**. If flow uses the data-channel capability, the validation system will ask to provide this property.
- `data_channel_uri` - represents the URL of the WhatsApp Flows Data Endpoint. If a Flow uses the data-channel capability, the validation system will ask to provide this property.

[Aplikasi Saya](#)[Tindakan yang diperlukan](#)[Dokumen](#)

by the [Flows API](#).

```
{
  "version": "2.1",
  "data_api_version": "3.0",
  "routing_model": {"MY_FIRST_SCREEN": ["MY_SECOND_SCREEN"] },
  "screens": [...],
  "data_channel_uri": "https://example.com"
}
```

```
{
  "version": "3.1",
  "data_api_version": "3.0",
  "routing_model": {"MY_FIRST_SCREEN": ["MY_SECOND_SCREEN"] },
  "screens": [...]
}
```

## Screens

Screens are the main unit of a Flow. Each screen represents a single node in the state machine you define. These properties then make up the Flows screen property model:

```
"screen" : {
  "id": string,
  "terminal": ?boolean,
  "success": ?boolean,
  "title": ?string,
  "refresh_on_back": ?boolean,
  "data": ?object,
  "layout": object
}
```

## Required properties

- **id** - unique identifier of the screen which works as a page url. SUCCESS is a reserved keyword and should not be used as a screen id.
- **layout** - associated screen UI Layout that is shown to the user. Layout can be predefined or it can represent a container with fully customizable content built using WhatsApp Flows Library.

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

- **terminal** (optional) - the business flow is the end state machine. It means that each Flow should have a terminal state where we terminate the experience and have the Flow completed. Multiple screens can be marked as terminal. It's mandatory to have a Footer component on the terminal screen.
- **data** (optional) - declaration of dynamic data that fills the components field in the Flow JSON. It uses JSON Schema to define the structure and type of the properties. Below you can find the simple example.

```
{
  "data": {
    "first_name": {
      "type": "string",
      "__example__": "John"
    }
  }
}
```

- **title** (optional) - screen level attribute that is rendered in the top navigation bar.
- **success** - (optional, only applicable on terminal screens) - Defaults to **true**. A Flow can have multiple terminal screens with different business outcomes. This property marks whether terminating on a terminal screen should be considered a successful business outcome.
- **refresh\_on\_back** (optional, and only applicable for Flows with a Data Endpoint) - Defaults to **false**. This property defines whether to trigger a data exchange request with the WhatsApp Flows Data Endpoint when using the back button while on this screen. The property is useful when you need to reevaluate the screen data when returning to the previous screen (example below).
- **sensitive** - (optional, only applicable for Flows version 5.1 and above) - Defaults to an empty array. When a Flow is completed, users will see a new message UI, which can be clicked and users will be able to view the responses submitted by them. This array contains the names of the fields in the screen that contain sensitive data, and should be hidden in the response summary for the consumers.

## Additional information on **refresh\_on\_back**

Given a simple Flow example with the following screens:

1. User chooses an **Appointment Type**.



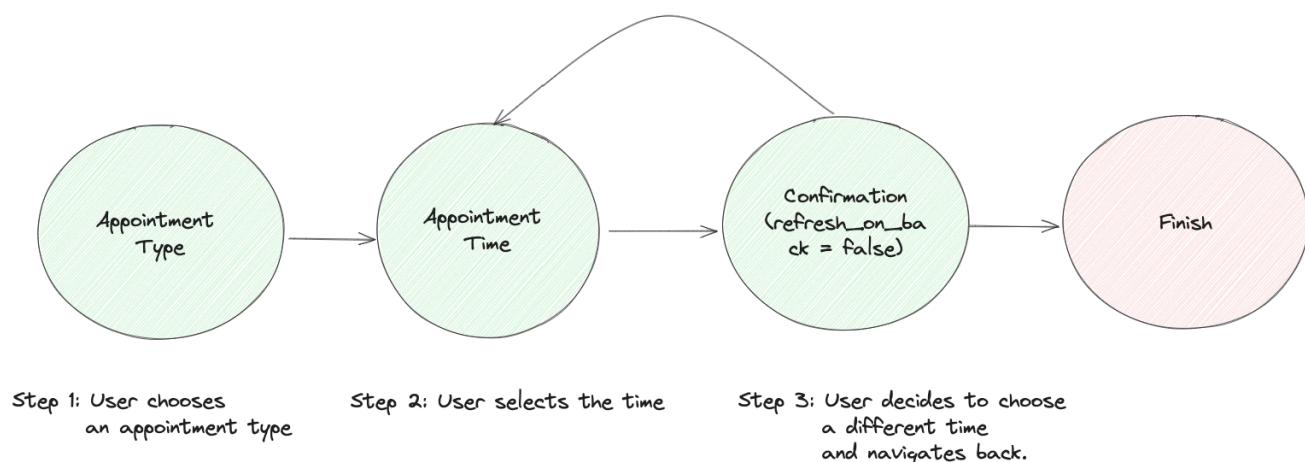
The user may navigate back from the confirmation page to re-select an appointment time. By using the `refresh_on_back` property in the `Confirmation` screen's definition, you can control whether to refresh the list of available times, or to reload the previously shown list.

### **refresh\_on\_back=false (default)**

If `refresh_on_back=false`, when the user goes back to the `Appointment Time` screen the Flow will not request the Flow Data Endpoint and the screen will be loaded with the previously provided data, and the user's prior input. This is the preferred behavior in most cases, given it avoids a roundtrip to the Flow Data Endpoint and provides a snappier experience for the user.

#### Scenario: `refresh_on_back = false`

Step 4: since `refresh_on_back = false`, no Flow Data Endpoint request will be sent; the `Appointment Time` screen will load with the prior data



### **refresh\_on\_back=true**

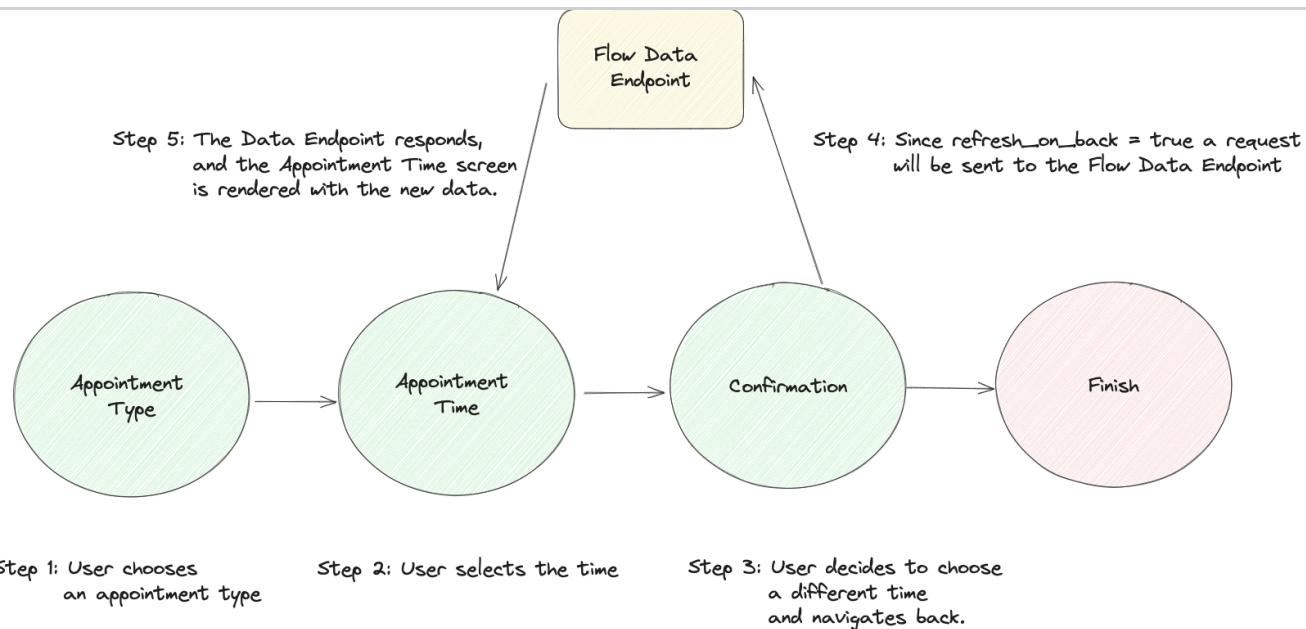
If, however, you need to revalidate or update the data for the screen, set `refresh_on_back=true` on the screen from which the back action is triggered (the `Confirmation` screen in this example). Once the user navigates back to the `Appointment Time` screen, the Flow will send a request to the Flow Data Endpoint and display the screen with the data from the response.



Aplikasi Saya

Tindakan yang diperlukan

Dokumen



### Flow Data Endpoint Request payload in case of `refresh_on_back=true`

The complete payload structure is defined here - in this case the `action` field will be `BACK`, and the `screen` will be set to the name of the `Confirmation` screen.

### Additional information on `sensitive` fields

Given a Flow in which we mark certain fields as `sensitive`, we use the following masking configuration to display the summary upon the Flow's completion:

Component	Masking	Consumer experience
Text Input	✓	Masked input value (••••••••••)
Password / OTP	✗	Hidden completely
Text Area	✓	Masked input value (••••••••••)
Date Picker	✓	Masked input value (••••••••••)
Dropdown	✓	Masked input value (••••••••••)
Checkbox Group	✓	Masked input value (••••••••••)



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

Radio Buttons Group	<input checked="" type="checkbox"/>	Masked input value (••••••••••••)
Opt In	<input type="checkbox"/>	Display as-is (no masking needed)
Document Picker	<input checked="" type="checkbox"/>	Hidden uploaded documents completely
Photo Picker	<input checked="" type="checkbox"/>	Hidden uploaded media completely

## Layout

Layout represents screen UI Content. It can be predefined by the WhatsApp Flows team, or the business can use empty containers and build custom experience using the WhatsApp Flows Library.

Layout has the following properties:

1. **type** - the layout identifier that's used in the template. In the current version of Flow JSON, there is only one layout available - "**SingleColumnLayout**" which represents a vertical flexbox container.
2. **children** - represents an array of components from the WhatsApp Flows Library.

## Routing Model

Routing model configuration is needed only when you use an Endpoint to power your flow.

You can define the routing model, which is a directed graph, as each screen can go to multiple other screens. There can be up to a maximum of 10 "branches", or connections, within the routing model.

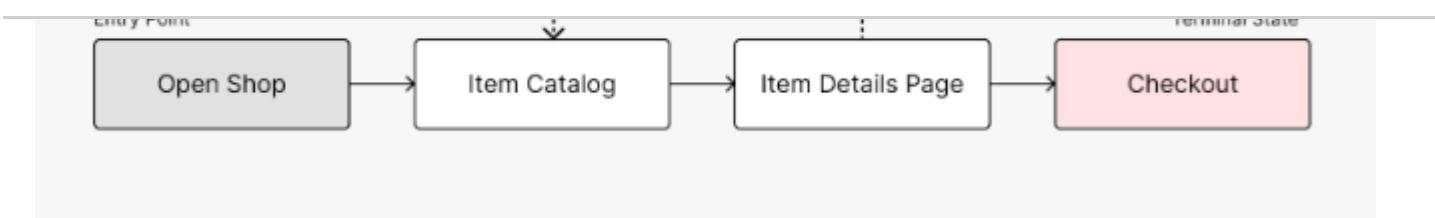
Consider the following flow:



Aplikasi Saya

Tindakan yang diperlukan

Dokumen



The following routing model can be built:

1. Item Catalog => [Item Details Page]
2. Item Details Page => [Item Catalog, Checkout]
3. Checkout => []

## When to define routes

If you don't use an Endpoint, you don't need to define a routing model, **it will be generated automatically**. However, if you want to use a server to power your Flow, you'll have to provide a `routing_model` in your Flow JSON.

## How to define routes

Routes are defined per screen via the `routing_model` property. It is a map of screen ids to an array of other screen ids it can transition to. The terminal screen is defined with `terminal=true`.

## Routing rules

1. Route cannot be the current screen, but the route can be "refreshed" for validation purposes.
2. If there is an edge between two screens, then the user can go back and forth between them using the BACK button.
3. Only forward routes should be specified in the routing model. For example, if you have specified an edge from Screen\_A to Screen\_B then you shouldn't specify another edge from Screen\_B to Screen\_A.
4. Routes can be empty for a screen if there is no forward route from it.
5. There should be an **entry** screen in the routing model. A screen is eligible to be an entry screen if it has no inbound edge.

[Aplikasi Saya](#)[Tindakan yang diperlukan](#)[Dokumen](#)

If your Flow is not using a Data Endpoint then an entry screen will be the one which is not set as the "next" screen in any of the "navigate" actions defined in the Flow.

6. All routes must end at the terminal screen.

### Routing Model Flow JSON Example (Endpoint)

In the example below, there is a simple 3-screen Flow that uses an Endpoint. It is expected that the server will return the next screen with a response to `data_exchange` action. The server has to comply with defined `routing_model` in the Flow JSON:

Flow JSON	Preview
<pre>     "id": "MY_SECOND_SCREEN",     "title": "Second Screen",     "data": {},     "layout": {       "type": "SingleColumnLayout",       "children": [         {           "type": "Footer",           "label": "Continue",           "on-click-action": {             "name": "data_exchange",             "payload": {}           }         }       ]     },     {       "id": "MY_THIRD_SCREEN",       "title": "Third Screen",       "terminal": true,       "success": true,       "data": {},       "layout": {         "type": "SingleColumnLayout",         "children": [           {             "type": "Footer",             "label": "Continue",             "on-click-action": {               "name": "data_exchange",               "payload": {}             }           }         ]       }     }   </pre>	<div style="display: flex; justify-content: space-between; align-items: center;"> <span style="font-weight: bold;">Preview</span> <span style="margin: 0 10px;">Run</span> <span></span> </div> <div style="margin-top: 10px;"> <div style="border: 1px solid #ccc; padding: 5px; display: inline-block; margin-bottom: 10px;">MY_FIRST_SCREEN</div> <div style="border: 1px solid #ccc; border-radius: 10px; padding: 10px; text-align: center;"> <span style="font-size: 1.2em; font-weight: bold;">First Screen</span> </div> <div style="border: 1px solid #ccc; border-radius: 10px; padding: 10px; text-align: center;"> <span style="font-size: 1.2em; font-weight: bold;">Continue</span> </div> <div style="text-align: right; margin-top: 10px;"> <small>Dikelola oleh bisnis. <a href="#">Pelajari selengkapnya</a></small> </div> </div>

## Properties

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

## Static properties

Static properties are simple. You set static properties once and they never change. Here is an example (see **text** and **label** properties of **TextHeading** and **Footer** components). Static properties is the simplest way to start building your Flow. You can always replace them later with dynamic content.

Flow JSON	Preview
<pre>{   "version": "7.3",   "screens": [     {       "id": "DEMO_SCREEN",       "title": "Demo Screen",       "terminal": true,       "success": true,       "layout": {         "type": "SingleColumnLayout",         "children": [           {             "type": "TextHeading",             "text": "This is a static h           },           {             "type": "Footer",             "label": "Static footer lat             "on-click-action": {               "name": "complete",               "payload": {}             }           }         ]       }     }   ] }</pre>	<p>Run</p> <p>Demo Screen</p> <p>This is a static heading</p> <p>Static footer label</p> <p>Dikelola oleh bisnis. <a href="#">Pelajari selengkapnya</a></p>

## Dynamic properties

[Aplikasi Saya](#)[Tindakan yang diperlukan](#)[Dokumen](#)

dynamic and static variant of the property together, you will get a compilation error. The dynamic data reference mechanism works with the following data types:

- string
- number
- boolean
- object
- array

You can dynamically reference these data types in all the components of Flow JSON. There are two types of dynamic properties:

- **Form properties** - `"${form.field_name}"` (data entered by the user in input fields). This is used to provide access to information that the user entered on the screen.
- **Screen properties** - `"${data.field_name}"` (data provided for the screen). This is used to provide access to information that is passed down by the server or the `navigate` action from the previous screen.

## Nested Expressions

Supported starting with Flow JSON version 6.0

Nested expressions allow conditionals and string concatenation to be created and used in components` properties (except `name` and `type` properties). Dynamic and static properties work as stated in this document in the previous sections, to enable nested expressions you need to wrap the property with backticks (`). Check below how to use it with all the available operations (code snippets provided). The available operations are:

- Equality comparisons (`==`, `!=`)
- Math comparisons (`<`, `<=`, `>`, `>=`)
- Logical comparisons (`&&`, `||`)
- String concatenation
- Math operations (`+`, `-`, `/`, `%`)

Note that in order to be able to use backticks as part of a string, you should add two back slashes (\\\) before if.

```
{
  "type": "TextBody",
```



## Equality comparisons

Operators	Types allowed	Return type
<code>==, !=</code>	strings, numbers and booleans  both sides should have the same type (validated during Flow JSON creation time)	<code>boolean</code>

Code snippet with strings:

```
{
  "type": "TextBody",
  "text": "Your first name should be different from your last name.",
  "visible": "`${form.first_name} == ${form.last_name}`"
}
```

Code snippet with booleans:

```
{
  "type": "TextBody",
  "text": "You have not accepted the T&C!",
  "visible": "`${form.accept} != true`"
}
```

Code snippet with numbers:

```
{
  "type": "TextBody",
  "text": "You are 18!",
  "visible": "`${form.age} == 18`"
}
```

## Math comparisons



Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

&lt;, &lt;=, &gt;, &gt;=

numbers

boolean

Code snippets:

```
{
  "type": "TextBody",
  "text": "You are above 18!",
  "visible: "`${form.age} > 18`"
}
```

```
{
  "type": "TextBody",
  "text": "You are above or at 18!",
  "visible: "`${form.age} >= 18`"
}
```

### Logical comparisons

Operators	Types allowed	Return type
&&,	booleans	boolean

Code snippets:

```
{
  "type": "TextBody",
  "text": "You have accepted the T&C and subscribed for our newsletter!",
  "visible: "`${form.accept} && ${form.subscribe}`"
}
```

Code snippet combined with other expressions:

```
{
  "type": "TextBody",
  "text": "You are above 18 and have accepted our T&C!",
  "visible: `(${form.age} > 18) && ${form.accept}`"
}
```



Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```

    "text": "You provided at least your first or last name.",
    "visible": "`(${form.first_name} != '') || (${form.last_name} != '')`"
}

```

## String Concatenation

Operators	Types allowed	Return type
No special sign is required, just add space between the expression members	strings, numbers and booleans	string

Code snippets:

```
{
  "type": "TextBody",
  "text": "'Hello ' ${form.first_name}'"
}
```

```
{
  "type": "TextBody",
  "text": "${form.first_name} ' you are ' ${form.age} ' years old.'"
}
```

## Math operations

Operators	Types allowed	Return type
+, -, /, %	numbers	number

Note that in case of division or modulo for **zero** or **null** value, the result will be **zero** to avoid **not a number** issue. Code snippets:

```
{
  "type": "TextBody",
  "text": "'You were born on either ' (2024 - ${form.age}) ' or ' (2023 - ${for
}
```

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```
"text": ``The amount per person is: ' ${data.total} / ${form.group_size}```
```

## Declaring screen properties (No endpoint example)

If a screen expects dynamic data, declare it inside the **data** property. Data declaration uses the standard JSON Schema. A simple Flow example would replace **text** with dynamic data coming from the message payload:

**Flow JSON**

```
{
  "version": "7.3",
  "screens": [
    {
      "id": "MY_FIRST_SCREEN",
      "title": "MY_FIRST_SCREEN",
      "terminal": true,
      "success": true,
      "data": {
        "hello_world_text": {
          "type": "string",
          "__example__": "Hello World"
        }
      },
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "TextHeading",
            "text": "${data.hello_world_text}"
          },
          {
            "type": "Footer",
            "label": "Static footer label",
            "on-click-action": {
              "name": "complete",
              "payload": {}
            }
          }
        ]
      }
    }
  ]
}
```

**Preview**

Run

MY\_FIRST\_SCREEN

**Hello World**

Static footer label

Dikelola oleh bisnis. [Pelajari selengkapnya](#)

A few things have been added:

Aplikasi Saya

Tindakan yang diperlukan

Dokumen



2. Inside the data field we declared `hello_world_text`. This is the data that we expect to receive for screen.

- `hello_world_text` follows the JSON Schema specification to declare the expected type, in this example it is a string.
- `__example__` field serves as mock data for the template, which is useful while you're developing your template without WhatsApp Flows Data Endpoint integration. **This field is mandatory.**

3. In `TextHeading` we've referenced the data via dynamic data reference syntax.  `${data}` represents an object that came from the WhatsApp Flows Data Endpoint or `navigate` actions in case of Flow without endpoint. You can treat it as a screen state that was set after the response is received.

4. Property of the state can be accessed using the following pattern - `"${data.property_name}"`

## Declaring screen properties (Endpoint powered example)

If you want to power the screen by endpoint data, the example above will slightly change.

1. We've added `data_api_version`, `routing_model` and `data_channel_uri` to indicate that the flow is connected to server.
2. We've added `data_api_version` and `routing_model` to indicate that the flow is connected to server. `data_channel_uri` should also be added if the Flow JSON version is less than **3.0**.
3. Flow expects to receive a payload from flow data server containing `hello_world_text` field.

Flow JSON	Preview	Run
<pre>{   "version": "7.3",   "data_api_version": "3.0",   "routing_model": {     "MY_FIRST_SCREEN": []   },   "screens": [     {       "id": "MY_FIRST_SCREEN",       "title": "MY_FIRST_SCREEN",       "terminal": true,       "success": true,       "data": {         "hello_world_text": {           "text": "Hello World"         }       }     }   ] }</pre>	<p><b>MY_FIRST_SCREEN</b></p> <p><b>Hello World</b></p>	<input type="button" value="Run"/>



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

```
        ,
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "TextHeading",
            "text": "${data.hello_world}"
          },
          {
            "type": "Footer",
            "label": "Static footer label",
            "on-click-action": {
              "name": "data_exchange",
              "payload": {}
            }
          }
        ]
      }
    }
```

Static footer label

## Forms and Form properties

The use of the Form component is optional starting from [Flow JSON versions 4.0](#). This means that you can submit user-entered data without the need to wrap your components inside a Form component any more.

To get and submit the data entered from users, Flow JSON uses a straightforward concept from HTML - Forms.

HTML Form example:

```
<form>
  <label for="first_name">First name</label><br>
  <input type="text" id="first_name" name="first_name"><br>
  <label for="last_name">Last name</label><br>
  <input type="text" id="last_name" name="last_name">

  <input type="radio" id="html" name="fav_language" value="HTML">
  <label for="html">HTML</label><br>

  <input type="radio" id="css" name="fav_language" value="CSS">
  <label for="css">CSS</label><br>

  <input type="radio" id="javascript" name="fav_language" value="JavaScript">
  <label for="javascript">JavaScript</label>
</form>
```

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen****Flow JSON**

```
{
  "version": "7.3",
  "screens": [
    {
      "id": "FORM_EXAMPLE",
      "title": "Demo Screen",
      "terminal": true,
      "success": true,
      "data": {},
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "Form",
            "name": "user_data",
            "children": [
              {
                "type": "TextInput",
                "required": true,
                "label": "First name",
                "name": "first_name"
              },
              {
                "type": "TextInput",
                "required": true,
                "label": "Second name",
                "name": "second_name"
              },
              {
                "type": "CheckboxGroup"
                "name": "favourite_languages"
                "label": "Favourite Languages (optional)"
                "data-source": [
                  {
                    "id": "javascript",
                    "title": "JavaScript"
                  }
                ]
              }
            ]
          }
        ]
      }
    }
  ]
}
```

**Preview**

Run

**Demo Screen**

Favourite Languages (optional)

Javascript

CSS

HTML

**Submit data**

Dikelola oleh bisnis. [Pelajari selengkapnya](#)

Alternative way to implement form starting from **Flow JSON versions 4.0** is as follows:

**Flow JSON**

```
{
  "version": "7.3",
  "screens": [
    {
      "id": "FORM_EXAMPLE",
      "title": "Demo Screen",
      "terminal": true,
      "success": true,
      "data": {},
      "layout": {
        "type": "Form"
        "name": "user_data"
        "children": [
          {
            "type": "TextInput"
            "label": "First name"
            "name": "first_name"
          },
          {
            "type": "TextInput"
            "label": "Second name"
            "name": "second_name"
          },
          {
            "type": "CheckboxGroup"
            "name": "favourite_languages"
            "label": "Favourite Languages (optional)"
            "data-source": [
              {
                "id": "javascript",
                "title": "JavaScript"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

**Preview**

Run

**Demo Screen**



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

<pre>     "type": "TextInput",     "required": true,     "label": "First name",     "name": "first_name"   },   {     "type": "TextInput",     "required": true,     "label": "Second name",     "name": "second_name"   },   {     "type": "CheckboxGroup",     "label": "Favourite Languages (optional)",     "name": "favourite_languages",     "data-source": [       {         "id": "javascript",         "title": "Javascript"       },       {         "id": "css",         "title": "CSS"       }     ]   } </pre>	<input type="text"/>  <input type="text"/>  <p>Favourite Languages (opsional)</p> <p>Javascript</p> <p>CSS</p> <p>HTML</p> <p><b>Submit data</b></p>
---	--

## Using Form properties

Using example above, we can reference form properties using a `"${form.field_name}"` binding. This type of binding uses `name` property of the interactive inputs to reference its value. You can use form values to submit the data to a flow data server or pass it another screen.

### Passing data to the next screen:

```
{
  "type": "Footer",
  "label": "Submit data",
  "on-click-action": {
    "name": "navigate",
    "next": { "type": "screen", "name": "NEXT_SCREEN" },
    "payload": {
      "name": "${form.first_name}",
      "lang": "${form.favourite_language}"
    }
  }
}
```

### Submitting data to the server:

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```

"label": "Submit data",
"on-click-action": {
  "name": "data_exchange",
  "payload": {
    "name": "${form.first_name}",
    "lang": "${form.favourite_language}"
  }
}
}

```

## Building forms guidelines

- In order to build Forms in Flow JSON you need to use Form components then provide the **name** and **children** properties
- Children properties must be an array of Form components
- Each Form component has its own property model, however the **name** property is required in all of them

Interactive components can not be used outside forms.

Component	Can Be Used Outside Forms? (Before Flow JSON 4.0)	Can Be Used Outside Forms? (Flow JSON 4.0+)
Text (TextHeading, TextSubheading, TextCaption, TextBody)	✓	✓
TextInput	✗	✓
TextArea	✗	✓
CheckboxGroup	✗	✓
RadioButtonsGroup	✗	✓



Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

	Aplikasi Saya	Tindakan yang diperlukan	Dokumen
Footer	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
OptIn	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Dropdown	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
EmbeddedLink	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
DatePicker	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

## Form configuration

Initial values of inputs can be initialised using `init-values` property. `error-messages` property allows you to set custom error for input. This is useful when you use Flow Data Endpoint to receive user data and you want to indicate that certain fields are incorrect.

Attribute	Description
<code>init-values</code>	<p>&lt;key, value&gt; object where</p> <p>key - Field Name in Component</p> <p>value - Field Initial Value</p> <p>type - String, Array&lt; String &gt; or Dynamic <code>init-values=" \${data.init_values} "</code></p>

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

<b>error-messages</b>	<p>&lt;key, value&gt; object where</p> <p>key - Field Name in Component</p> <p>value - Error Message</p> <p>type - String or Dynamic <b>error-messages=</b> "\${data.error_messages}"</p>
-----------------------	---

You set **init-values** by specifying the field name in the respective component, then mapping it to your desired value.

The data type for **init-values** must match that of the component as outlined below.

Component	<b>init-values</b> data type
CheckboxGroup	Array of Strings
RadioButtonsGroup	String
Text Entry	String
Dropdown	String

For example, if you have the field **first\_name** in one **TextInput** component, the field **second\_name** in another **TextInput** component you would set the **init-values** like so:

**Flow JSON**

```
{
  "version": "7.3",
  "screens": [
    {
      "id": "DEMO_SCREEN",
      "title": "Demo Screen",
      "terminal": true,
      "success": true,
      "data": {},
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "TextEntry",
            "label": "First name",
            "name": "first_name",
            "initValue": "John"
          },
          {
            "type": "TextEntry",
            "label": "Last name",
            "name": "second_name",
            "initValue": "Doe"
          }
        ]
      }
    }
  ]
}
```

**Preview**

Run



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

```

    "name": "user_data",
    "init-values": {
      "first_name": "Jon",
      "second_name": "Joe"
    },
    "children": [
      {
        "type": "TextInput",
        "required": true,
        "label": "First name",
        "name": "first_name"
      },
      {
        "type": "TextInput",
        "required": true,
        "label": "Second name",
        "name": "second_name"
      },
      {
        "type": "Footer",
        "label": "Submit data",
        "on-click-action": {
          "type": "submit"
        }
      }
    ]
  }
}

```

Second name —  
Joe

Submit data

Starting from **Flow JSON versions 4.0**, the utilization of the Form component has become optional. In the event that you opt not to use the Form component, you can still initialize the initial values of inputs by employing the **init-value** property, and set custom errors for each input with **error-message** property. Here is an example showcasing how to initialize values without utilizing the Form component:

## Flow JSON

```

{
  "version": "7.3",
  "screens": [
    {
      "id": "DEMO_SCREEN",
      "title": "Demo Screen",
      "terminal": true,
      "success": true,
      "data": {},
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "TextInput",
            "required": true,
            "label": "First name",
            "name": "first_name",
            "init-value": "Jon"
          },
          {
            "type": "TextInput",
            "required": true,
            "label": "Second name",
            "name": "second_name"
          }
        ]
      }
    }
  ]
}

```

## Preview

Run

Demo Screen

First name —  
Jon

Second name —  
Joe



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

```
{  
  "type": "Footer",  
  "label": "Submit data",  
  "on-click-action": {  
    "name": "complete",  
    "payload": {  
      "name": "${form.first_r  
    }  
  }  
}
```

## Global dynamic and form properties

Supported starting with Flow JSON version 4.0

Starting from **Flow JSON Version 4.0**, you can use **Global Dynamic Referencing** feature. It has the following syntax:

```
 ${screen.<screen_name>.({form | data}).<field-name>}
```

1. **screen** - global variable that gives access to screen storage
2. **screen\_name** - name of the screen to refer
3. **(form | data)** - type of storage you want to reference - Form / Dynamic data
4. **field-name** - name of the field you want to reference

## Where it can be used

1. Component properties that support **dynamic** data
2. Action payloads
3. Screen titles
4. Conditional components

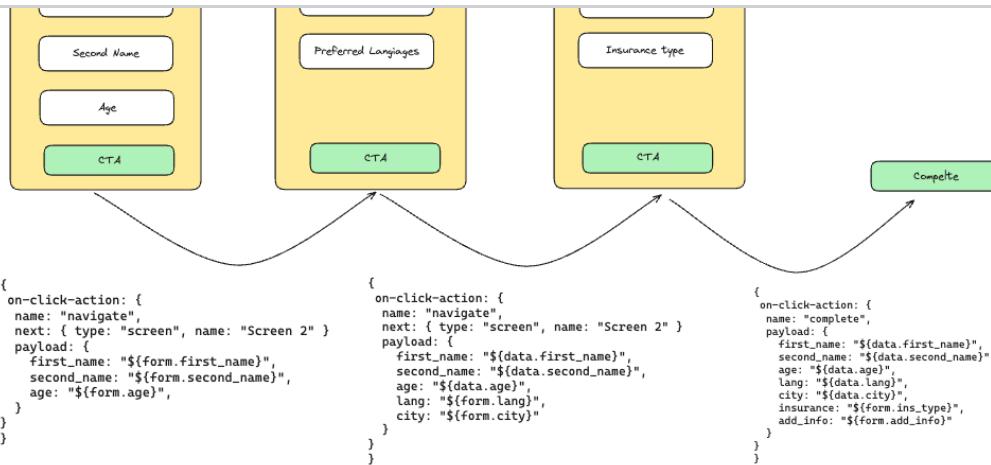
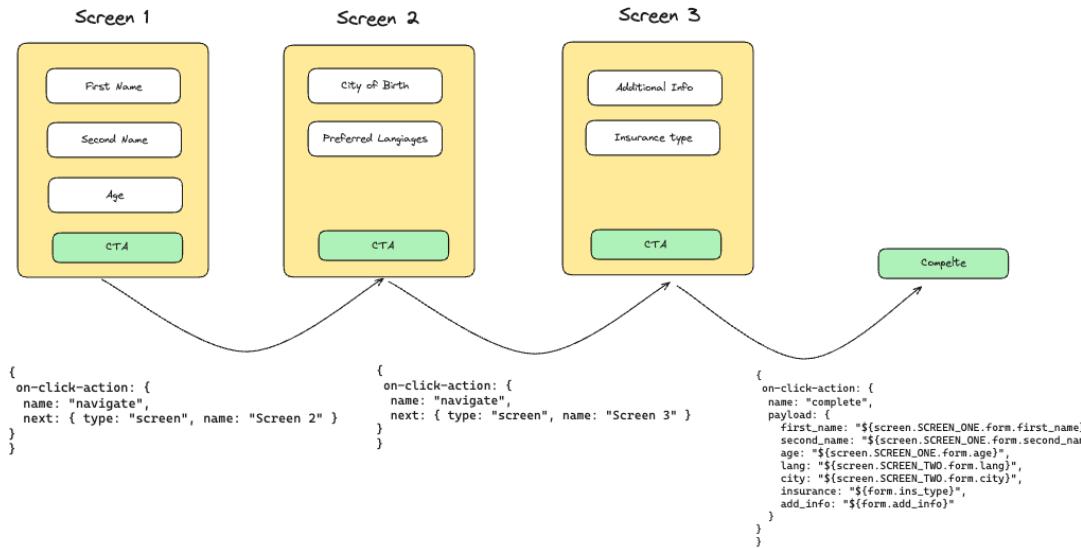
## Example 1 - Carrying data forward



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

**Before****After**

Before **Flow JSON Version 4.0**, to transfer data from one screen to another you would need to use **navigate** action. For instance to transfer the data from **SCREEN\_ONE** to **SCREEN\_TWO**, you would write the following:

```
"on-click-action": {
    "name": "navigate",
    "next": {
        "type": "screen",
        "name": "SCREEN_TWO"
    },
    "payload": {
        "field1": "${data.field_one}",
        "field2": "${form.field_two}"
    }
}
```

In Flow JSON V4.0 you don't need to transfer the data via **navigate** since all data now is globally accessible, so instead you can keep **payload** as empty {}



Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```
"next": {
    "type": "screen",
    "name": "SCREEN_TWO"
},
"payload": {}
```

And on **SCREEN\_TWO** you can reference it as:

```
{
  "type": "TextBody",
  "text": "${screen.SCREEN_ONE.data.field1}"
}
```

## Example 2 - No screen **data** declaration for global fields

When you use global fields on the screen, you don't need to specify them in the **data** model. Global fields utilise data-model of the parent screens. See example below, we use

**\${screen.SCREEN\_ONE.form.field1}**  and  **\${screen.SCREEN\_ONE.data.field2}**  on  **SCREEN\_TWO** , since the data comes from  **SCREEN\_ONE** , we keep  **data**  model as empty on  **SCREEN\_TWO**

<b>Flow JSON</b> <pre>{   "version": "7.3",   "screens": [     {       "data": {         "field2": {           "type": "string",           "__example__": "data"         }       },       "id": "SCREEN_ONE",       "layout": {         "type": "SingleColumnLayout",         "children": [           {             "type": "TextInput",             "name": "field1",             "label": "Enter your name"           },           {             "type": "Footer",             "label": "CTA",             "on-click-action": {               "name": "next"             }           }         ]       }     }   ] }</pre>	<b>Preview</b> <div style="display: flex; justify-content: space-between;"> <span>Run</span> <span><input type="button" value=""/></span> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">SCREEN_ONE</div> <div style="border: 1px solid #ccc; padding: 10px; background-color: #f0f0f0; border-radius: 10px; text-align: center;"> <span>Screen 1</span>   <input type="text" value="Enter your name (optional)"/> </div>
--	---



Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```

        },
        "payload": {}
    }
]
},
"title": "Screen 1"
},
{

```

### Example 3 - Forward references

When you use global fields on the screen, you can also reference "future" screens. The only caveat is that you need to handle empty values. This can be done with **Conditional Rendering** components. See example below:

1. **SELECT\_SERVICES** screen references the data from **SELECT\_INSURANCE**
2. When value from **SELECT\_INSURANCE** screen is empty - we display **You haven't selected any insurance type**
3. When value is not empty, based on selected value - we display different text via **Switch** statement

**Flow JSON**

```
{
  "version": "7.3",
  "routing_model": {
    "SELECT_SERVICES": [
      "SELECT_INSURANCE"
    ]
  },
  "screens": [
    {
      "terminal": true,
      "id": "SELECT_SERVICES",
      "title": "Select services",
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "TextSubheading",
            "text": "Select insurance t
          },
          {
            "type": "If",
            "condition": "(${screen.SELECT_INSURANCE} === '')",
            "then": [
              {
                "type": "TextSubheading",
                "text": "Choose insurance type
              }
            ]
          }
        ]
      }
    }
  ]
}
```

**Preview**

Run

**SELECT\_SERVICES**

Select services

**Select insurance type**

You haven't selected any insurance type

**Choose insurance type**



Aplikasi Saya

Tindakan yang diperlukan

Dokumen

```
{
  "type": "TextBody",
  "text": "You've s
}
],
"standard": [
{
  "type": "TextBody"
}
```

## Actions

Flow JSON provides a generic way to trigger asynchronous actions handled by a client through interactive UI elements. The following actions are supported:

Flow JSON Reference	Description	Payload Type
<a href="#">data_exchange</a>	Sending Data to WhatsApp Flows Data Endpoint	Customizable JSON payload on data exchanges { [key:string]: any }
<a href="#">navigate</a>	Triggers the next screen with the payload as its input. The CTA button will be disabled until the payload with data required for the next screen is supplied.	Static JSON payload
<a href="#">complete</a>	Triggers the termination of the Flow with the provided payload.	Static JSON payload
<a href="#">update_data</a>	Triggers an immediate update to the screen's state, reflecting user input changes.	Static JSON payload
<a href="#">open_url</a>	Triggers a link to open in the device's default web browser.	No payload is accepted by the <a href="#">open_url</a> action. It only accepts a <a href="#">url</a> property (i.e. the URL of the link to open).

[Aplikasi Saya](#)[Tindakan yang diperlukan](#)[Dokumen](#)

This action is a primary way to navigate between the screens of the flow. The data that's passed as **payload** of this action will be available on the next screen through dynamic data referencing -  `${data.field_name}`. You shouldn't use it on the Footer of a terminal screen because that will prevent the flow from terminating.

### When to use

Use this action when you need to transition to another screen.

### Example

```
{
  "type": "Footer",
  "label": "Continue",
  "on-click-action": {
    "name": "navigate",
    "next": { "type": "screen", "name": "NEXT_SCREEN" },
    "payload": {
      "name": "${form.first_name}",
      "lang": "${form.favourite_language}"
    }
  }
}
```

### complete action

Terminates the flow and sends the response message to the chat thread. The business will receive the termination message bubble on the webhook, together with the `flow_token` and all of the other parameters from the payload. More information can be found [here](#).

### When to use

Use this action on **terminal** screen as a last interaction of the user. Once triggered, the flow will be terminated and entered data will be submitted via webhook.

We strongly recommend to only include data inputted by the user in the Flow's completion payload, and to keep the payload size to a minimum. Avoid leveraging the completion payload to send base64 images.

### Example

```
{
  "type": "Footer",
```

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

```
"payload": {
  "discount_code": "${data.discount_code}",
  "items": "${form.selected_items}"
}
```

## Example of flow using **navigate** and **complete** actions

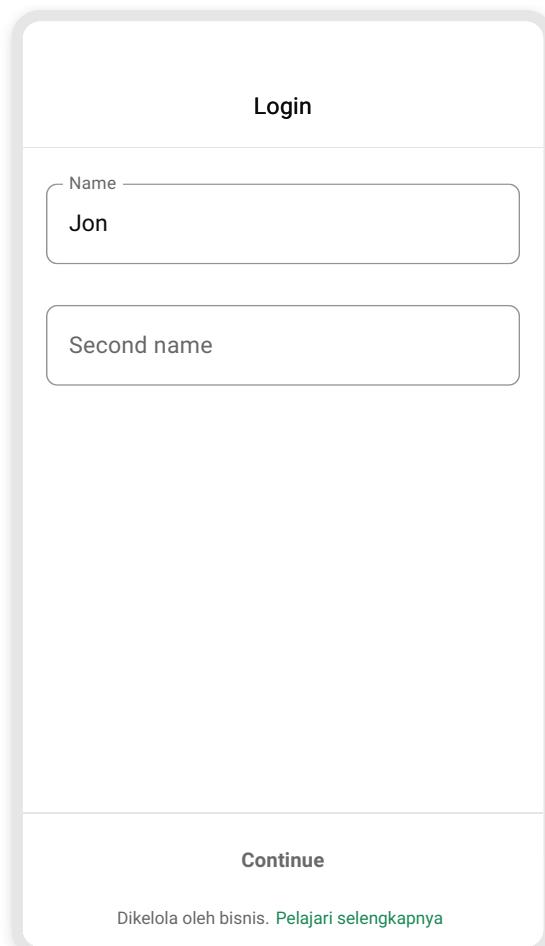
### Flow JSON

```
{
  "version": "7.3",
  "screens": [
    {
      "id": "LOGIN",
      "layout": {
        "type": "SingleColumnLayout",
        "children": [
          {
            "type": "Form",
            "name": "flow_path",
            "init-values": {
              "name": "Jon"
            },
            "children": [
              {
                "type": "TextInput",
                "name": "name",
                "required": true,
                "input-type": "text",
                "label": "Name"
              },
              {
                "type": "TextInput",
                "name": "second_name",
                "required": true,
                "input-type": "text",
                "label": "Second name"
              }
            ],
            "on-click-action": {
              "name": "navigate",
              "next": {
                "type": "screen",
                "screen": "Dashboard"
              }
            }
          }
        ]
      }
    }
  ]
}
```

### Preview

Run

LOGIN



## data\_exchange action



## When to use

You can only use this action if your Flow is powered by a Data Endpoint. Use this action when you need to submit data to your server before transitioning to the next screen or terminating the flow. Your server could then decide on the next step and provide the input for it.

## Example

```
{
  "type": "Footer",
  "label": "Submit data",
  "on-click-action": {
    "name": "data_exchange",
    "payload": {
      "discount_code": "${data.discount_code}",
      "items": "${form.selected_items}"
    }
  }
}
```

## update\_data action

This action is supported from Flow JSON version 6.0 onwards.

This allows us to update the state of the current screen based on user interactions. For example, when a user selects a value from a dropdown, another dropdown on the screen can be updated immediately. The data that's passed as payload of this action can have multiple key-value pairs, where the key references the dynamic data of the screen where we are using this action. The value of the payload can be form/data, and can be referenced using the global dynamic referencing syntax.

## When to use

Use the update\_data action to dynamically update the content displayed on a screen based on user interactions, without the need to navigate away or refresh the screen. This action is particularly beneficial in scenarios where:

- 1. Immediate Response Required:** You need to update elements on the same screen in response to user inputs. In the provided example, selecting a country from the dropdown triggers an immediate update to the state dropdown. This ensures that users experience no delay in seeing relevant options, enhancing the responsiveness of the application.
- 2. Dynamic Data Handling:** This action excels in scenarios where data relationships are dynamic yet predefined within the components data-source. In the example, each country object

Aplikasi Saya

Tindakan yang diperlukan

**Dokumen**

times and server dependency.

**3. Reusable Templates:** Utilizing update\_data promotes the reuse of flow components across different data contexts. The state dropdown in the example is a single component that is repopulated with different data depending on the country selected. This approach minimizes the need for multiple distinct templates, simplifying the application structure and reducing development effort.

## Examples

### RadioButtonsGroup Example

Flow JSON	Preview
<pre>{   "version": "6.0",   "screens": [     {       "id": "ADDRESS_SELECTION",       "layout": {         "type": "SingleColumnLayout",         "children": [           {             "type": "RadioButtonsGroup",             "name": "select_country",             "label": "Select country:",             "data-source": "\${data.country}"           },           {             "type": "RadioButtonsGroup",             "name": "select_states",             "label": "Select state:",             "visible": "\${data.state_visible}",             "data-source": "\${data.states}"           },           {             "type": "RadioButtonsGroup",             "name": "pincode",             "label": "Select pincode:",             "visible": "\${data.pincode_visible}",             "data-source": "\${data.pincode}"           },           {             "type": "Footer",             "label": "Complete",             "on-click-action": {               "name": "complete",               "payload": {}             }           }         ]       }     }   ] }</pre>	<p><b>Preview</b></p> <p>Run</p> <div style="border: 1px solid #ccc; padding: 10px; width: fit-content;"> <p style="text-align: center;">Address selection</p> <p>Select country: (opsional)</p> <p>USA</p> <p>Canada</p> <p style="text-align: right;">Complete</p> <p>Dikelola oleh bisnis. <a href="#">Pelajari selengkapnya</a></p> </div>

[Aplikasi Saya](#)[Tindakan yang diperlukan](#)[Dokumen](#)

This action is supported from Flow JSON version 6.0 onwards.

This action opens the URL of a website that loads in the device's default mobile web browser when the URL text in the Flow is tapped by the user. It can be used only with the [EmbeddedLink](#) and [OptIn](#) components.

### When to use

Use this when you want to redirect the user to an external link outside WhatsApp. For example, to open the "Terms and Conditions" link.

### EmbeddedLink Example

```
{  
  "type": "EmbeddedLink",  
  "text": "This is an external link.",  
  "on-click-action": {  
    "name": "open_url",  
    "url": "https://www.whatsapp.com/"  
  }  
,
```

### OptIn Example

```
{  
  "type": "OptIn",  
  "label": "I agree to the terms.",  
  "name": "T&Cs",  
  "on-click-action": {  
    "name": "open_url",  
    "url": "https://www.whatsapp.com/"  
  }  
,
```

## Components

A comprehensive list of components with code examples is available [here](#).

## Static Validation Errors



## Limitations

Flow JSON content string is limited and cannot exceed 10 MB.

### WhatsApp Flows

[Get Started](#)

[Guides](#)

[Reference](#)

#### Flow JSON

[Flows API](#)

[Error Codes](#)

[Flows Encryption](#)

[Versioning](#)

[Metrics API](#)

[Webhooks](#)

[Lifecycle of a Flow](#)

[Components](#)

[Media Upload Components](#)

[Playground](#)

[Help - Status and Support](#)

[Changelog](#)