Homey Flow Cards Development Tutorial

Overview

Homey flow cards are the building blocks that allow users to create automations in the Homey ecosystem. As an app developer, you can create custom flow cards that integrate your devices and services into Homey's flow system.

Types of Flow Cards

1. Trigger Cards (WHEN)

- Triggered by events (device state changes, timers, etc.)
- Start a flow when something happens
- Examples: "When motion is detected", "When temperature rises above X"

2. Condition Cards (AND)

- Check conditions before continuing a flow
- Return true/false
- Examples: "Temperature is above X", "Device is turned on"

3. Action Cards (THEN)

- Perform actions when flow executes
- Examples: "Turn on device", "Send notification", "Set temperature"

Project Setup

1. Initialize Homey App

```bash
# Install Homey CLI
npm install -g homey

# Create new app
homey app create

# Navigate to app directory
cd your-app-name
```

2. Proper App Structure with .homeycompose

File Structure Summary

To avoid confusion, here's the complete file structure for Homey apps with flow cards:

```
your-app/
├── app.js
├── app.json (generated during build)
├── .homeycompose/
│   ├── app.json (base app configuration)
│   └── flow/ (app-level flow cards)
│       ├── triggers/
│       │   └── temperature_changed.json
│       ├── conditions/
│       │   └── temperature_above.json
│       └── actions/
│           └── set_temperature.json
├── drivers/
│   └── your-device/
│       ├── driver.js
│       ├── device.js
│       ├── driver.compose.json (driver config: capabilities, class, etc.)
│       └── driver.flow.compose.json (device-specific flow cards)
└── locales/
    └── en.json
```

Key distinction:

- `driver.compose.json` = Driver configuration (capabilities, device class, pairing, etc.)
- `driver.flow.compose.json` = Device-specific flow card definitions only

```
your-app/
├── app.js
├── app.json (generated during build)
├── .homeycompose/
│   ├── app.json
│   └── flow/
│       ├── triggers/
│       │   └── temperature_changed.json
│       ├── conditions/
│       │   └── temperature_above.json
│       └── actions/
│           └── set_temperature.json
├── drivers/
│   └── your-device/
│       ├── driver.js
│       ├── device.js
│       ├── driver.compose.json
│       └── driver.flow.compose.json
├── locales/
│   └── en.json
└── assets/
```

3. Build Process

When you run `homey app run` or `homey app build`, the Homey CLI will:

1. Read all files from `.homeycompose/`
2. Merge them into a single `app.json` in the root directory
3. This generated `app.json` is what Homey actually uses

Flow Card Definition using .homeycompose

1. Base App Configuration

`.homeycompose/app.json`:

```
json
```

```
{
  "id": "com.yourcompany.yourapp",
  "version": "1.0.0",
  "compatibility": ">=5.0.0",
  "sdk": 3,
  "name": {
    "en": "Your App Name"
  },
  "description": {
    "en": "Your app description"
  },
  "category": [
    "climate"
  ],
  "permissions": [],
  "images": {
    "large": "/assets/images/large.png",
    "small": "/assets/images/small.png"
  },
  "author": {
    "name": "Your Name",
    "email": "your.email@example.com"
  }
}
```

2. Individual Flow Card Files

.homeycompose/flow/triggers/temperature_changed.json:

```
json
```

```
{
  "id": "temperature_changed",
  "title": {
    "en": "Temperature changed"
  },
  "titleFormatted": {
    "en": "Temperature changed to [[temperature]]°C"
  },
  "hint": {
    "en": "Triggered when the temperature sensor reports a new value"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "driver_id=temperature_sensor"
    }
  ],
  "tokens": [
    {
      "name": "temperature",
      "type": "number",
      "title": {
        "en": "Temperature"
      },
      "example": 23.5
    },
    {
      "name": "previous_temperature",
      "type": "number",
      "title": {
        "en": "Previous temperature"
      },
      "example": 22.1
    }
  ]
}
```

.homeycompose/flow/conditions/temperature_above.json:

```
json
```

```
{
  "id": "temperature_above",
  "title": {
    "en": "Temperature is above..."
  },
  "titleFormatted": {
    "en": "Temperature is above [[temperature]]°C"
  },
  "hint": {
    "en": "Check if the current temperature is above the specified value"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "driver_id=temperature_sensor"
    },
    {
      "name": "temperature",
      "type": "number",
      "min": -50,
      "max": 100,
      "step": 0.1,
      "placeholder": {
        "en": "Temperature"
      }
    }
  ]
}
```

.homeycompose/flow/actions/set_temperature.json:

```
json
```

```
{
  "id": "set_temperature",
  "title": {
    "en": "Set temperature"
  },
  "titleFormatted": {
    "en": "Set temperature to [[temperature]]°C"
  },
  "hint": {
    "en": "Set the target temperature for the thermostat"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "driver_id=thermostat"
    },
    {
      "name": "temperature",
      "type": "number",
      "min": 5,
      "max": 35,
      "step": 0.5,
      "placeholder": {
        "en": "Temperature"
      }
    }
  ]
}
```

Implementation in app.js

1. Basic App Class

javascript

```
'use strict';

const Homey = require('homey');

class YourApp extends Homey.App {

  async onInit() {
    this.log('Your app is running...');

    // Register flow card listeners
    this.registerFlowCards();
  }

  registerFlowCards() {
    // Register app-level flow cards (including device-filtered ones)
    this.registerAppFlowCards();

    // Device-specific cards are automatically registered by drivers
  }

  registerAppFlowCards() {
    // Register trigger cards
    this.registerTriggerCards();

    // Register condition cards
    this.registerConditionCards();

    // Register action cards
    this.registerActionCards();
  }

  registerTriggerCards() {
    // Temperature changed trigger (device-filtered at app level)
    this._temperatureChangedTrigger = this.homey.flow.getTriggerCard('temperature_changed');

    // No additional registration needed - device filter handled automatically
  }

  registerConditionCards() {
    // Temperature above condition (device-filtered)
    const temperatureAboveCondition = this.homey.flow.getConditionCard('temperature_above');

    temperatureAboveCondition.registerRunListener(async (args, state) => {
      const device = args.device;  // Device passed via device argument
      const targetTemperature = args.temperature;
      const currentTemperature = device.getCapabilityValue('measure_temperature');
```

```
    this.log(`Checking if ${currentTemperature}°C > ${targetTemperature}°C for device ${device.getName()}`);

    return currentTemperature > targetTemperature;
  });
}

registerActionCards() {
  // Set temperature action (device-filtered)
  const setTemperatureAction = this.homey.flow.getActionCard('set_temperature');

  setTemperatureAction.registerRunListener(async (args, state) => {
    const device = args.device;  // Device passed via device argument
    const temperature = args.temperature;

    this.log(`Setting temperature to ${temperature}°C for device ${device.getName()}`);

    try {
      await device.setCapabilityValue('target_temperature', temperature);
      return true;
    } catch (error) {
      this.error('Failed to set temperature:', error);
      throw new Error('Failed to set temperature');
    }
  });
}

// Helper method to trigger device-filtered flow cards
async triggerTemperatureChanged(device, temperature, previousTemperature) {
  const tokens = {
    temperature: temperature,
    previous_temperature: previousTemperature
  };

  const state = {
    device: device  // Pass device in state for filtering
  };

  // This will only trigger for flows that have this specific device selected
  await this._temperatureChangedTrigger.trigger(tokens, state);
}
}

module.exports = YourApp;
### 3. Key Differences: App-Level vs Device-Specific Flow Cards

| Aspect | App-Level with Device Filter | Device-Specific (driver.flow.compose.json) |
```

```
|--------|--------------------------|----------------------------------|
| **Definition Location** | `.homeycompose/flow/` | `drivers/*/driver.flow.compose.json` |
| **Flow Card ID** | Shared across all devices | Unique per driver |
| **Registration** | `homey.flow.getTriggerCard()` | `homey.flow.getDeviceTriggerCard()` |
| **Device Access** | Via `args.device` parameter | Via `this` (device instance) |
| **Filtering** | Runtime filtering by Homey | Compile-time filtering |
| **Flexibility** | Can filter by multiple criteria | Driver-specific only |
| **Code Maintenance** | Centralized in app.js | Distributed in device classes |
| **User Experience** | Same card works across devices | Different cards per device type |

**Example - App-Level with Device Filter:**
```javascript
// Defined in .homeycompose/flow/actions/set_temperature.json
// Works for ANY device that matches the filter
const setTempAction = this.homey.flow.getActionCard('set_temperature');
setTempAction.registerRunListener(async (args, state) => {
  const device = args.device; // User-selected device
  await device.setCapabilityValue('target_temperature', args.temperature);
});
```
```

Example - Device-Specific (in driver.flow.compose.json):

```
javascript

// Defined in drivers/thermostat/driver.flow.compose.json
// Only works for thermostat devices
const setTempAction = this.homey.flow.getDeviceActionCard('set_temperature');
setTempAction.registerRunListener(async (args, state) => {
  // 'this' is automatically the thermostat device instance
  await this.setCapabilityValue('target_temperature', args.temperature);
});
```

For device-specific cards defined in `driver.compose.json`:

```
javascript
```

```
'use strict';

const Homey = require('homey');

class TemperatureSensorDevice extends Homey.Device {

  async onInit() {
    this.log('Temperature sensor device initialized');

    // Register capability listeners
    this.registerCapabilityListener('measure_temperature', this.onTemperatureChanged.bind(this));

    // Register device-specific flow cards (from driver.flow.compose.json)
    this.registerDeviceFlowCards();
  }

  registerDeviceFlowCards() {
    // Device-specific trigger (defined in driver.compose.json)
    this._deviceTemperatureThresholdTrigger = this.homey.flow.getDeviceTriggerCard('temperature_threshold_exc

    // Device-specific condition
    const deviceTempStableCondition = this.homey.flow.getDeviceConditionCard('is_temperature_stable');
    deviceTempStableCondition.registerRunListener(async (args, state) => {
      // No device argument needed - 'this' is the device
      return this.isTemperatureStable();
    });

    // Device-specific action
    const calibrateSensorAction = this.homey.flow.getDeviceActionCard('calibrate_sensor');
    calibrateSensorAction.registerRunListener(async (args, state) => {
      const offset = args.offset;
      await this.calibrateSensor(offset);
    });
  }

  async onTemperatureChanged(value, opts) {
    const previousValue = this.getStoreValue('previous_temperature') || value;
    this.setStoreValue('previous_temperature', value);

    this.log(`Temperature changed from ${previousValue}°C to ${value}°C`);

    // Trigger app-level card (device-filtered)
    const app = this.homey.app;
    await app.triggerTemperatureChanged(this, value, previousValue);

    // Trigger device-specific card if threshold exceeded
```

```
    if (value > this.getSetting('threshold_temperature')) {
      const tokens = { threshold: this.getSetting('threshold_temperature') };
      const state = {};
      await this._deviceTemperatureThresholdTrigger.trigger(this, tokens, state);
    }
  }

  isTemperatureStable() {
    // Implementation for checking temperature stability
    const readings = this.getStoreValue('recent_readings') || [];
    if (readings.length < 5) return false;

    const variance = this.calculateVariance(readings);
    return variance < 0.5; // Stable if variance is less than 0.5°C
  }

  async calibrateSensor(offset) {
    this.log(`Calibrating sensor with offset: ${offset}°C`);
    await this.setSettings({ temperature_offset: offset });
  }
}

module.exports = TemperatureSensorDevice;
```

3. Development Workflow

Development Process:

1. Create/edit files in `.homeycompose/`
2. Run `homey app run` to build and test
3. CLI compiles `.homeycompose/` files into root `app.json`
4. App runs with the generated configuration

Important Notes:

- Never edit the root `app.json` directly - it gets overwritten
- Always work in `.homeycompose/` directory
- Use `homey app validate` to check for errors
- The generated `app.json` should be in `.gitignore`

4. Complex Flow Card Examples

`.homeycompose/flow/actions/send_notification.json`:

```
json
```

```
{
  "id": "send_notification",
  "title": {
    "en": "Send notification"
  },
  "titleFormatted": {
    "en": "Send notification [[message]] to [[user]]"
  },
  "args": [
    {
      "name": "message",
      "type": "text",
      "placeholder": {
        "en": "Notification message"
      }
    },
    {
      "name": "user",
      "type": "autocomplete",
      "placeholder": {
        "en": "Select user"
      }
    },
    {
      "name": "priority",
      "type": "dropdown",
      "values": [
        {
          "id": "low",
          "title": {
            "en": "Low"
          }
        },
        {
          "id": "normal",
          "title": {
            "en": "Normal"
          }
        },
        {
          "id": "high",
          "title": {
            "en": "High"
          }
        }
      ]
```

```
    }
  ]
}
```

Advanced Flow Card Features

1. Dynamic Arguments Based on Capability Properties

You can make flow cards more intelligent by adapting their arguments based on device capability definitions:

.homeycompose/flow/actions/set_dim_level.json:

```json
{
  "id": "set_dim_level",
  "title": {
    "en": "Set dim level"
  },
  "titleFormatted": {
    "en": "Set [[device]] to [[level]]%"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "capabilities=dim"
    },
    {
      "name": "level",
      "type": "range",
      "min": 0,
      "max": 100,
      "step": 1,
      "label": "%",
      "labelMultiplier": 100,
      "labelDecimals": 0
    }
  ]
}
```

Implementation with capability-aware logic:

```javascript
```

```
registerActionCards() {
  const setDimAction = this.homey.flow.getActionCard('set_dim_level');

  // Register argument autocomplete for dynamic min/max based on device
  setDimAction.registerArgumentAutocompleteListener('level', async (query, args) => {
    const device = args.device;
    if (!device) return [];

    // Get capability definition from device
    const capabilityOptions = device.getCapabilityOptions('dim') || {};
    const min = capabilityOptions.min || 0;
    const max = capabilityOptions.max || 1;
    const step = capabilityOptions.step || 0.01;

    // Convert to percentage for user display
    const minPercent = Math.round(min * 100);
    const maxPercent = Math.round(max * 100);
    const stepPercent = Math.round(step * 100);

    // Generate appropriate suggestions based on device capabilities
    const suggestions = [];
    for (let i = minPercent; i <= maxPercent; i += stepPercent * 10) {
      suggestions.push({
        id: i,
        name: `${i}%`,
        description: `Set to ${i}%`
      });
    }

    return suggestions.filter(s =>
      s.name.toLowerCase().includes(query.toLowerCase())
    );
  });

  setDimAction.registerRunListener(async (args, state) => {
    const device = args.device;
    const levelPercent = args.level;

    // Convert percentage back to capability value (0-1)
    const levelValue = levelPercent / 100;

    // Validate against device capability limits
    const capabilityOptions = device.getCapabilityOptions('dim') || {};
    const min = capabilityOptions.min || 0;
    const max = capabilityOptions.max || 1;
```

```
    if (levelValue < min || levelValue > max) {
      throw new Error(`Value ${levelPercent}% is outside device range ${min * 100}%-${max * 100}%`);
    }

    await device.setCapabilityValue('dim', levelValue);
  });
}
```

2. Temperature Capabilities with Device-Specific Ranges

.homeycompose/flow/actions/set_target_temperature.json:

json

```
{
  "id": "set_target_temperature",
  "title": {
    "en": "Set target temperature"
  },
  "titleFormatted": {
    "en": "Set [[device]] to [[temperature]]°C"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "capabilities=target_temperature"
    },
    {
      "name": "temperature",
      "type": "number",
      "min": 5,
      "max": 35,
      "step": 0.5,
      "placeholder": {
        "en": "Temperature"
      }
    }
  ]
}
```

Smart implementation:

javascript

```
registerActionCards() {
  const setTargetTempAction = this.homey.flow.getActionCard('set_target_temperature');

  setTargetTempAction.registerRunListener(async (args, state) => {
    const device = args.device;
    const temperature = args.temperature;

    // Get device-specific temperature ranges
    const tempCapability = device.getCapabilityOptions('target_temperature') || {};
    const min = tempCapability.min !== undefined ? tempCapability.min : 5;
    const max = tempCapability.max !== undefined ? tempCapability.max : 35;
    const step = tempCapability.step || 0.5;
    const units = tempCapability.units || '°C';

    // Validate against device-specific ranges
    if (temperature < min) {
      throw new Error(`Temperature ${temperature}${units} is below device minimum of ${min}${units}`);
    }
    if (temperature > max) {
      throw new Error(`Temperature ${temperature}${units} is above device maximum of ${max}${units}`);
    }

    // Round to device step if specified
    const roundedTemp = Math.round(temperature / step) * step;

    this.log(`Setting temperature to ${roundedTemp}${units} for ${device.getName()}`);
    await device.setCapabilityValue('target_temperature', roundedTemp);
  });
}
```

3. Enum-Based Capabilities (Mode Selection)

.homeycompose/flow/actions/set_thermostat_mode.json:

```
json
```

```
{
  "id": "set_thermostat_mode",
  "title": {
    "en": "Set thermostat mode"
  },
  "titleFormatted": {
    "en": "Set [[device]] mode to [[mode]]"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "capabilities=thermostat_mode"
    },
    {
      "name": "mode",
      "type": "autocomplete",
      "placeholder": {
        "en": "Select mode"
      }
    }
  ]
}
```

Dynamic mode selection based on device capabilities:

```javascript
```

```
registerActionCards() {
  const setModeAction = this.homey.flow.getActionCard('set_thermostat_mode');

  // Dynamic autocomplete based on device's supported modes
  setModeAction.registerArgumentAutocompleteListener('mode', async (query, args) => {
    const device = args.device;
    if (!device) return [];

    // Get supported modes from device capability
    const modeCapability = device.getCapabilityOptions('thermostat_mode') || {};
    const supportedModes = modeCapability.values || [
      { id: 'heat', title: { en: 'Heat' } },
      { id: 'cool', title: { en: 'Cool' } },
      { id: 'auto', title: { en: 'Auto' } },
      { id: 'off', title: { en: 'Off' } }
    ];

    // Filter based on query and return device-specific modes
    return supportedModes
      .filter(mode => {
        const title = mode.title[this.homey.i18n.getLanguage()] || mode.title.en;
        return title.toLowerCase().includes(query.toLowerCase());
      })
      .map(mode => ({
        id: mode.id,
        name: mode.title[this.homey.i18n.getLanguage()] || mode.title.en,
        description: `Set to ${mode.title[this.homey.i18n.getLanguage()] || mode.title.en} mode`
      }));
  });

  setModeAction.registerRunListener(async (args, state) => {
    const device = args.device;
    const mode = args.mode.id;

    // Validate mode is supported by device
    const modeCapability = device.getCapabilityOptions('thermostat_mode') || {};
    const supportedModes = modeCapability.values || [];

    const isSupported = supportedModes.some(supportedMode => supportedMode.id === mode);
    if (!isSupported && supportedModes.length > 0) {
      const supportedModeNames = supportedModes.map(m => m.id).join(', ');
      throw new Error(`Mode '${mode}' not supported. Supported modes: ${supportedModeNames}`);
    }

    await device.setCapabilityValue('thermostat_mode', mode);
```

```
  });
}
```

Built-in Capability Flow Cards vs Custom Flow Cards

Built-in Capability Flow Cards vs Custom Flow Cards

Built-in Flow Cards from Standard Capabilities

Homey automatically provides flow cards for many standard capabilities. When you add these capabilities to your device, the corresponding flow cards become available:

Automatic Flow Cards by Capability:

Capability	Automatic Trigger Cards	Automatic Condition Cards	Automatic Action Cards
`onoff`	"Turned on/off"	"Is turned on"	"Turn on", "Turn off", "Toggle"
`dim`	"Dim level changed"	"Dim level is higher/lower than"	"Set dim level to", "Increase/decrease dim level"
`measure_temperature`	"Temperature changed"	"Temperature is higher/lower than"	-
`target_temperature`	"Target temperature changed"	"Target temperature is higher/lower than"	"Set temperature to"
`locked`	"Locked/unlocked"	"Is locked"	"Lock", "Unlock"
`alarm_motion`	"Motion alarm turned on/off"	"Motion alarm is on"	-
`alarm_contact`	"Door/window alarm turned on/off"	"Door/window alarm is on"	-
`measure_humidity`	"Humidity changed"	"Humidity is higher/lower than"	-
`thermostat_mode`	"Thermostat mode changed"	"Thermostat mode is"	"Set thermostat mode"
`volume_set`	"Volume changed"	"Volume is higher/lower than"	"Set volume", "Volume up/down"
`speaker_playing`	"Started/stopped playing"	"Is playing"	"Play", "Pause", "Stop"

Automatic Flow Cards for Custom Capabilities

IMPORTANT: For ANY capability you define (including custom ones), Homey automatically creates trigger flow cards based on the capability type:

Automatic Trigger Card Generation:

Capability Type	Automatic Trigger Cards Created
`"type": "number"`	`<capability_id>_changed`
`"type": "string"`	`<capability_id>_changed`
`"type": "enum"`	`<capability_id>_changed`
`"type": "boolean"`	`<capability_id>_true` and `<capability_id>_false`

Examples of Custom Capability Triggers:

Custom number capability:

```json
{
  "type": "number",
  "title": { "en": "Pool pH Level" },
  "min": 0,
  "max": 14,
  "step": 0.1
}
```

Automatically creates trigger: `pool_ph_changed`

Custom enum capability:

```json
{
  "type": "enum",
  "title": { "en": "Pool Mode" },
  "values": [
    { "id": "heating", "title": { "en": "Heating" } },
    { "id": "cooling", "title": { "en": "Cooling" } },
    { "id": "maintenance", "title": { "en": "Maintenance" } }
  ]
}
```

Automatically creates trigger: `pool_mode_changed`

Custom boolean capability:

```json
```

```
{
  "type": "boolean",
  "title": { "en": "Filter Running" }
}
```

Automatically creates triggers: `filter_running_true` and `filter_running_false`

Using Automatic Capability Triggers

You DON'T need to define these automatic triggers in your `.homeycompose/flow/` files - they're created automatically. However, you DO need to understand how to use them:

For Number/String/Enum Capabilities:

javascript

```
class PoolDevice extends Homey.Device {

  async onInit() {
    // Get the automatic trigger (created by Homey)
    this._poolPhChangedTrigger = this.homey.flow.getDeviceTriggerCard('pool_ph_changed');

    // Register capability listener
    this.registerCapabilityListener('pool_ph', this.onPoolPhChanged.bind(this));
  }

  async onPoolPhChanged(value, opts) {
    this.log(`Pool pH changed to ${value}`);

    // Trigger the automatic flow card
    const tokens = {
      pool_ph: value  // Token name matches capability ID
    };

    const state = {};

    // Homey automatically triggers this when capability changes,
    // but you can also trigger it manually if needed
    await this._poolPhChangedTrigger.trigger(this, tokens, state);
  }

  async updatePoolPh(newValue) {
    // This will automatically trigger the 'pool_ph_changed' flow card
    await this.setCapabilityValue('pool_ph', newValue);
  }
}
```

For Boolean Capabilities:

```javascript
class PoolDevice extends Homey.Device {

  async onInit() {
    // Get the automatic boolean triggers
    this._filterRunningTrueTrigger = this.homey.flow.getDeviceTriggerCard('filter_running_true');
    this._filterRunningFalseTrigger = this.homey.flow.getDeviceTriggerCard('filter_running_false');

    // Register capability listener
    this.registerCapabilityListener('filter_running', this.onFilterRunningChanged.bind(this));
  }

  async onFilterRunningChanged(value, opts) {
    this.log(`Filter running changed to ${value}`);

    // Homey automatically triggers the appropriate flow card based on boolean value
    // You don't need to manually trigger these - Homey does it automatically

    if (value) {
      // 'filter_running_true' trigger fires automatically
      this.log('Filter started - true trigger will fire');
    } else {
      // 'filter_running_false' trigger fires automatically
      this.log('Filter stopped - false trigger will fire');
    }
  }

  async startFilter() {
    // This automatically triggers 'filter_running_true' flow card
    await this.setCapabilityValue('filter_running', true);
  }

  async stopFilter() {
    // This automatically triggers 'filter_running_false' flow card
    await this.setCapabilityValue('filter_running', false);
  }
}
```

Important Notes About Automatic Triggers

1. **No Definition Required**: Don't create JSON files for these triggers - Homey creates them automatically
2. **Automatic Firing**: These triggers fire automatically when you call `setCapabilityValue()`

3. **Token Names**: Token names in automatic triggers match the capability ID
4. **Boolean Special Case**: Boolean capabilities get TWO triggers (`_true` and `_false`) instead of one `_changed`
5. **Always Available**: These triggers are available even if you don't register any listeners

Mixing Automatic and Custom Triggers

You can combine automatic capability triggers with custom triggers:

Automatic triggers (free):

- `pool_ph_changed` (from `pool_ph` capability)
- `filter_running_true`/`filter_running_false` (from `filter_running` capability)

Custom triggers (you define):

```json
// .homeycompose/flow/triggers/pool_maintenance_required.json
{
  "id": "pool_maintenance_required",
  "title": { "en": "Pool maintenance required" },
  "tokens": [
    {
      "name": "maintenance_type",
      "type": "string",
      "title": { "en": "Maintenance type" }
    }
  ]
}
```

This gives users both standard capability-based triggers AND your custom device-specific triggers.

When to Use Built-in vs Custom Flow Cards

Use Built-in Flow Cards When:

- The generic naming works well for your device type
- You want minimal development effort
- Standard functionality is sufficient
- You're building simple devices

Example - Basic temperature sensor:

```json
```

```
{
  "capabilities": [
    "measure_temperature",
    "measure_humidity",
    "alarm_battery"
  ]
}
```

This automatically provides temperature/humidity condition cards and battery alarm triggers without any custom flow card development.

Create Custom Flow Cards When:

- You need device-specific naming (e.g., "Pool temperature" instead of "Temperature")
- You want to combine multiple capabilities in one card
- You need custom validation or complex logic
- You want to provide a branded user experience
- You need to expose device-specific features beyond standard capabilities

Hybrid Approach: Built-in + Custom Flow Cards

Most professional apps use both built-in capability cards AND custom cards:

.homeycompose/app.json:

```json
{
  "id": "com.yourcompany.poolcontroller",
  "name": {
    "en": "Pool Controller"
  }
}
```

Device with standard capabilities (gets built-in cards):

```json
```

```
{
  "capabilities": [
    "measure_temperature",
    "target_temperature",
    "onoff"
  ]
}
```

Plus custom flow cards for pool-specific features:

.homeycompose/flow/actions/start_pool_cleaning.json:

```
json

{
  "id": "start_pool_cleaning",
  "title": {
    "en": "Start pool cleaning cycle"
  },
  "titleFormatted": {
    "en": "Start [[duration]] minute cleaning cycle for [[device]]"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "driver_id=pool_controller"
    },
    {
      "name": "duration",
      "type": "number",
      "min": 10,
      "max": 120,
      "step": 10,
      "placeholder": {
        "en": "Duration (minutes)"
      }
    }
  ]
}
```

This gives users BOTH:

- Standard cards: "Turn on", "Set temperature to", "Temperature is higher than"
- Custom cards: "Start pool cleaning cycle"

Overriding Built-in Flow Card Behavior

You can override or enhance built-in capability behavior:

```javascript
```

```
class PoolDevice extends Homey.Device {

  async onInit() {
    // Standard capability listener (affects built-in flow cards)
    this.registerCapabilityListener('target_temperature', this.onTargetTemperatureChanged.bind(this));

    // Custom flow card for enhanced temperature setting
    this.registerCustomFlowCards();
  }

  async onTargetTemperatureChanged(value, opts) {
    // This affects both built-in flow cards AND your custom logic
    this.log(`Target temperature changed to ${value}°C`);

    // Custom validation for pool temperature
    if (value < 15 || value > 40) {
      throw new Error('Pool temperature must be between 15°C and 40°C');
    }

    // Custom logic for pool heating
    await this.startHeatingSystem(value);

    // Trigger custom flow card
    await this.triggerPoolTemperatureSet(value);
  }

  registerCustomFlowCards() {
    const setPoolTempAction = this.homey.flow.getActionCard('set_pool_temperature_with_timer');

    setPoolTempAction.registerRunListener(async (args, state) => {
      const temperature = args.temperature;
      const duration = args.duration;

      // Set temperature (this will trigger built-in capability flow cards too)
      await this.setCapabilityValue('target_temperature', temperature);

      // Start timer (custom functionality)
      await this.startTemperatureTimer(duration);
    });
  }
}
```

Checking Available Built-in Flow Cards

You can see what built-in flow cards are available for your device:

1. **In Homey Developer Tools:**
   - Install your app
   - Go to Flows in Homey app
   - Create new flow
   - Look for cards with your device name
2. **Programmatically check:**

```javascript
// In your app.js or device.js
async onInit() {
  // Log all available flow cards for debugging
  this.log('Available trigger cards:', this.homey.flow.getTriggerCards());
  this.log('Available condition cards:', this.homey.flow.getConditionCards());
  this.log('Available action cards:', this.homey.flow.getActionCards());
}
```

Flow Card Registration and Error Handling

What Happens Without Run Listeners?

When you define flow cards in `.homeycompose` or `driver.compose.json` but forget to register run listeners, the cards will still appear in the Homey flow editor, but they won't work properly when executed.

Behavior by Card Type:

Trigger Cards (WHEN) without listeners:

- ✅ Card appears in flow editor
- ❌ Never triggers (no code to fire them)
- ❌ Flows using them never execute
- ❌ No error shown to user - flow just "doesn't work"

Condition Cards (AND) without listeners:

- ✅ Card appears in flow editor
- ❌ Always returns `false` (or throws error)
- ❌ Flow stops at condition
- ⚠️ Error logged: "No run listener registered for condition card 'card_id'"

Action Cards (THEN) without listeners:

- ✅ Card appears in flow editor
- ❌ Action fails when executed
- ❌ Flow may stop or continue depending on error handling
- ⚠️ Error logged: "No run listener registered for action card 'card_id'"

Example: Missing Run Listener

Defined card works in editor:

```json
// .homeycompose/flow/actions/set_temperature.json
{
  "id": "set_temperature",
  "title": { "en": "Set temperature" },
  "args": [...]
}
```

But no JavaScript registration:

```javascript
// app.js - MISSING REGISTRATION
async onInit() {
  // Forgot to register the listener!
  // this.registerFlowCards();
}
```

Result when user tries to use it:

```
[log] [ManagerFlow] No run listener registered for action card 'set_temperature'
[error] Flow execution failed at action 'Set temperature'
```

Proper Registration Patterns

Complete Registration Example:

```javascript
```

```
class YourApp extends Homey.App {

  async onInit() {
    this.log('App initializing...');

    try {
      await this.registerFlowCards();
      this.log('All flow cards registered successfully');
    } catch (error) {
      this.error('Failed to register flow cards:', error);
    }
  }

  async registerFlowCards() {
    // Register all triggers
    this.registerTriggerCards();

    // Register all conditions
    this.registerConditionCards();

    // Register all actions
    this.registerActionCards();
  }

  registerTriggerCards() {
    // MUST store reference for triggering later
    this._temperatureChangedTrigger = this.homey.flow.getTriggerCard('temperature_changed');

    // Triggers don't need run listeners, but you MUST trigger them somewhere
    this.log('Temperature changed trigger registered');
  }

  registerConditionCards() {
    const tempCondition = this.homey.flow.getConditionCard('temperature_above');

    // MUST register run listener
    tempCondition.registerRunListener(async (args, state) => {
      const device = args.device;
      const threshold = args.temperature;
      const current = device.getCapabilityValue('measure_temperature');

      this.log(`Condition check: ${current}°C > ${threshold}°C = ${current > threshold}`);
      return current > threshold;
    });

    this.log('Temperature condition registered');
```

```
  }

  registerActionCards() {
    const setTempAction = this.homey.flow.getActionCard('set_temperature');

    // MUST register run listener
    setTempAction.registerRunListener(async (args, state) => {
      const device = args.device;
      const temperature = args.temperature;

      this.log(`Setting temperature to ${temperature}°C`);
      await device.setCapabilityValue('target_temperature', temperature);

      return true; // Indicate success
    });

    this.log('Set temperature action registered');
  }
}
```

Debugging Missing Registrations

1. Check Registration at Startup

javascript

```
async onInit() {
  await this.registerFlowCards();

  // Verify registrations
  this.validateFlowCardRegistration();
}

validateFlowCardRegistration() {
  const definedCards = [
    'temperature_changed',
    'temperature_above',
    'set_temperature'
  ];

  definedCards.forEach(cardId => {
    try {
      // Try to get the card - will throw if not found
      const card = this.homey.flow.getTriggerCard(cardId) ||
                   this.homey.flow.getConditionCard(cardId) ||
                   this.homey.flow.getActionCard(cardId);

      if (card) {
        this.log(`✅ Flow card '${cardId}' found and registered`);
      } else {
        this.error(`❌ Flow card '${cardId}' not found`);
      }
    } catch (error) {
      this.error(`❌ Flow card '${cardId}' registration error:`, error.message);
    }
  });
}
```

2. Common Registration Mistakes

Wrong card type:

```
javascript

// WRONG - condition card registered as trigger
this._tempCondition = this.homey.flow.getTriggerCard('temperature_above');

// CORRECT
const tempCondition = this.homey.flow.getConditionCard('temperature_above');
```

Missing device-specific registration:

```javascript
// WRONG - device cards need getDeviceActionCard
const deviceAction = this.homey.flow.getActionCard('device_specific_action');

// CORRECT - in device class
const deviceAction = this.homey.flow.getDeviceActionCard('device_specific_action');
```

Forgetting to store trigger references:

```javascript
// WRONG - can't trigger later
this.homey.flow.getTriggerCard('temperature_changed');

// CORRECT - store reference
this._temperatureChangedTrigger = this.homey.flow.getTriggerCard('temperature_changed');
```

Error Handling in Flow Cards

Robust Action Implementation:

```javascript
```

```javascript
setTemperatureAction.registerRunListener(async (args, state) => {
  try {
    const device = args.device;
    const temperature = args.temperature;

    // Validate device
    if (!device) {
      throw new Error('No device selected');
    }

    if (!device.getAvailable()) {
      throw new Error(`Device ${device.getName()} is not available`);
    }

    // Validate capability
    if (!device.hasCapability('target_temperature')) {
      throw new Error(`Device ${device.getName()} does not support temperature control`);
    }

    // Validate value
    if (temperature < 5 || temperature > 35) {
      throw new Error('Temperature must be between 5°C and 35°C');
    }

    // Execute action
    await device.setCapabilityValue('target_temperature', temperature);

    this.log(`Successfully set ${device.getName()} to ${temperature}°C`);
    return true;

  } catch (error) {
    this.error('Set temperature action failed:', error);

    // Re-throw with user-friendly message
    throw new Error(`Failed to set temperature: ${error.message}`);
  }
});
```

Robust Condition Implementation:

```javascript
```

```
temperatureCondition.registerRunListener(async (args, state) => {
  try {
    const device = args.device;
    const threshold = args.temperature;

    // Validate inputs
    if (!device) {
      this.log('No device provided for condition');
      return false;
    }

    if (!device.getAvailable()) {
      this.log(`Device ${device.getName()} not available`);
      return false;
    }

    // Get current value
    const currentTemp = device.getCapabilityValue('measure_temperature');

    if (currentTemp === null || currentTemp === undefined) {
      this.log(`No temperature reading for ${device.getName()}`);
      return false;
    }

    const result = currentTemp > threshold;
    this.log(`Temperature condition: ${currentTemp}°C > ${threshold}°C = ${result}`);

    return result;

  } catch (error) {
    this.error('Temperature condition error:', error);
    return false; // Conditions should return false on error
  }
});
```

Flow Card Lifecycle Summary

1. **Definition** (`.homeycompose/` or `driver.compose.json`) → Card appears in editor
2. **Registration** (`getTriggerCard`/`getConditionCard`/`getActionCard`) → Card becomes accessible
3. **Listener Registration** (`registerRunListener`) → Card becomes functional
4. **Execution** (User runs flow) → Your listener code executes

Missing any step means the card won't work properly!

When mixing built-in and custom cards, maintain naming consistency:

Built-in cards (automatic):

- "Turn on Pool Controller"
- "Pool Controller temperature is higher than"

Custom cards (maintain same pattern):

- "Start Pool Controller cleaning cycle"
- "Pool Controller filter needs replacement"

Performance Considerations

Built-in flow cards:

- ✅ Zero development time
- ✅ Automatically maintained by Homey
- ✅ Consistent UX across apps
- ❌ Generic naming
- ❌ Limited customization

Custom flow cards:

- ✅ Perfect naming and UX
- ✅ Device-specific functionality
- ✅ Complete control over behavior
- ❌ Development and maintenance overhead
- ❌ Need to handle validation, errors, etc.

Recommendation: Start with built-in cards for basic functionality, then add custom cards for device-specific features and enhanced user experience.

Create a utility class to simplify capability-aware flow card development:

```javascript
```

```
class FlowCardHelper {

  static getCapabilityRange(device, capability) {
    const options = device.getCapabilityOptions(capability) || {};
    return {
      min: options.min,
      max: options.max,
      step: options.step,
      units: options.units,
      decimals: options.decimals
    };
  }

  static validateCapabilityValue(device, capability, value) {
    const range = this.getCapabilityRange(device, capability);

    if (range.min !== undefined && value < range.min) {
      throw new Error(`Value ${value} below minimum ${range.min}${range.units || ''}`);
    }

    if (range.max !== undefined && value > range.max) {
      throw new Error(`Value ${value} above maximum ${range.max}${range.units || ''}`);
    }

    return true;
  }

  static roundToCapabilityStep(device, capability, value) {
    const range = this.getCapabilityRange(device, capability);
    if (range.step) {
      return Math.round(value / range.step) * range.step;
    }
    return value;
  }

  static getCapabilityEnum(device, capability) {
    const options = device.getCapabilityOptions(capability) || {};
    return options.values || [];
  }

  static generateNumberSuggestions(device, capability, query = '') {
    const range = this.getCapabilityRange(device, capability);
    const min = range.min || 0;
    const max = range.max || 100;
    const step = range.step || 1;
    const units = range.units || '';
```

```
    const suggestions = [];
    const stepSize = Math.max(step, (max - min) / 10); // Max 10 suggestions

    for (let i = min; i <= max; i += stepSize) {
      const value = Math.round(i / step) * step;
      const displayValue = `${value}${units}`;

      if (displayValue.includes(query)) {
        suggestions.push({
          id: value,
          name: displayValue,
          description: `Set to ${displayValue}`
        });
      }
    }

    return suggestions;
  }
}
```

Usage example:

```javascript
// In your flow card registration
temperatureAction.registerArgumentAutocompleteListener('temperature', async (query, args) => {
  const device = args.device;
  if (!device) return [];

  return FlowCardHelper.generateNumberSuggestions(device, 'target_temperature', query);
});

temperatureAction.registerRunListener(async (args, state) => {
  const device = args.device;
  let temperature = args.temperature;

  // Validate and round to device step
  FlowCardHelper.validateCapabilityValue(device, 'target_temperature', temperature);
  temperature = FlowCardHelper.roundToCapabilityStep(device, 'target_temperature', temperature);

  await device.setCapabilityValue('target_temperature', temperature);
});
```

1. Complete Argument Types Reference

Beyond the basic `text`, `number`, and `dropdown` types, Homey supports many specialized argument types:

Date and Time Arguments

Date picker (`"type": "date"`):

```json
{
  "name": "birthday",
  "type": "date",
  "title": { "en": "Birthday" },
  "placeholder": { "en": "18-05-1994" }
}
```

Time picker (`"type": "time"`):

```json
{
  "name": "alarm_time",
  "type": "time",
  "title": { "en": "Alarm time" },
  "placeholder": { "en": "07:30" }
}
```

Visual and Interactive Arguments

Range slider (`"type": "range"`):

```json
{
  "name": "brightness",
  "type": "range",
  "title": { "en": "Brightness" },
  "min": 0,
  "max": 1,
  "step": 0.01,
  "label": "%",
  "labelMultiplier": 100,
  "labelDecimals": 0
}
```

Color picker (`"type": "color"`):

```json
{
  "name": "background_color",
  "type": "color",
  "title": { "en": "Background color" }
}
```

Checkbox (`"type": "checkbox"`):

```json
{
  "name": "enabled",
  "type": "checkbox",
  "title": { "en": "Enabled" }
}
```

Special Token Arguments

Droptoken (token-only input):

```json
{
  "id": "logic_equals",
  "title": { "en": "Logic equals" },
  "titleFormatted": { "en": "[[droptoken]] equals [[value]]" },
  "droptoken": ["number", "string"],
  "args": [
    {
      "name": "value",
      "type": "text",
      "placeholder": { "en": "Comparison value" }
    }
  ]
}
```

Optional Arguments

Make arguments optional with `"required": false`:

```json
```

```
{
  "name": "message",
  "type": "text",
  "required": false,
  "title": { "en": "Optional message" },
  "placeholder": { "en": "Leave empty for default message" }
}
```

Handle optional arguments in code:

javascript

```
actionCard.registerRunListener(async (args, state) => {
  const message = args.message || "Default message";
  // Handle optional argument safely
});
```

Action Duration Support

For action cards, add automatic duration support:

json

```
{
  "id": "run_animation",
  "title": { "en": "Run animation" },
  "titleFormatted": { "en": "Run [[animation]] animation" },
  "duration": true,
  "args": [
    {
      "name": "animation",
      "type": "dropdown",
      "values": [
        { "id": "rainbow", "title": { "en": "Rainbow" } },
        { "id": "pulse", "title": { "en": "Pulse" } }
      ]
    }
  ]
}
```

Handle duration in code:

javascript

```
runAnimationAction.registerRunListener(async (args, state) => {
  const animation = args.animation;
  const duration = args.duration; // Duration in milliseconds, or null

  if (duration) {
    this.log(`Running ${animation} animation for ${duration}ms`);
    await this.runAnimationWithTimeout(animation, duration);
  } else {
    this.log(`Running ${animation} animation indefinitely`);
    await this.runAnimation(animation);
  }
});
```

2. Advanced Autocomplete Arguments

Basic autocomplete setup:

```
json

{
  "name": "user",
  "type": "autocomplete",
  "placeholder": { "en": "Select user" }
}
```

Implementation with filtering:

```
javascript

sendMessageAction.registerArgumentAutocompleteListener('user', async (query, args) => {
  const users = await this.getUsers();

  return users
    .filter(user => user.name.toLowerCase().includes(query.toLowerCase()))
    .map(user => ({
      id: user.id,
      name: user.name,
      description: user.email,
      // Optional: Add icon for visual enhancement
      icon: user.avatar || "/app/assets/icons/user.svg"
    }));
});
```

Advanced autocomplete with context:

```
javascript
```

```
// Autocomplete that depends on other arguments
playlistAction.registerArgumentAutocompleteListener('song', async (query, args) => {
  const selectedArtist = args.artist; // Get value from other argument

  if (!selectedArtist) {
    return []; // No songs without artist selection
  }

  const songs = await this.getSongsByArtist(selectedArtist.id);

  return songs
    .filter(song => song.title.toLowerCase().includes(query.toLowerCase()))
    .map(song => ({
      id: song.id,
      name: song.title,
      description: `${song.duration} - ${song.album}`,
      image: song.albumArt // Use image instead of icon for album art
    }));
});
```

3. Advanced Device Filtering

Beyond basic `driver_id` filtering, Homey supports complex device filters:

Multiple values with pipe (`|`):

```
json

{
  "name": "device",
  "type": "device",
  "filter": "driver_id=thermostat_v1|thermostat_v2|smart_thermostat"
}
```

Multiple properties with ampersand (`&`):

```
json

{
  "name": "device",
  "type": "device",
  "filter": "class=light&capabilities=onoff,dim"
}
```

Z-Wave specific filters:

```json
{
  "name": "device",
  "type": "device",
  "filter": "driver_id=zwave_switch&flags=zwaveMultiChannel"
}
```

Zigbee specific filters:

```json
{
  "name": "device",
  "type": "device",
  "filter": "driver_id=zigbee_sensor&flags=zigbeeSubDevice"
}
```

4. Flow Tokens and Advanced Data Passing

Local Tokens (Trigger Card Outputs)

Define tokens in trigger cards:

```json
```

```
{
  "id": "motion_detected",
  "title": { "en": "Motion detected" },
  "tokens": [
    {
      "name": "zone",
      "type": "string",
      "title": { "en": "Zone" },
      "example": "Living Room"
    },
    {
      "name": "confidence",
      "type": "number",
      "title": { "en": "Detection confidence" },
      "example": 0.95
    },
    {
      "name": "snapshot",
      "type": "image",
      "title": { "en": "Motion snapshot" }
    }
  ]
}
```

Pass tokens when triggering:

javascript

```
async triggerMotionDetected(device, motionData) {
  const tokens = {
    zone: motionData.zoneName,
    confidence: motionData.confidence,
    snapshot: motionData.snapshotImage
  };

  const state = { deviceId: device.getData().id };

  await this._motionDetectedTrigger.trigger(device, tokens, state);
}
```

Action Card Return Tokens (Advanced Flow)

Action cards can return tokens for Advanced Flow:

json

```
{
  "id": "analyze_image",
  "title": { "en": "Analyze image" },
  "tokens": [
    {
      "name": "objects_found",
      "type": "number",
      "title": { "en": "Objects found" }
    },
    {
      "name": "analysis_result",
      "type": "string",
      "title": { "en": "Analysis result" }
    }
  ]
}
```

Return tokens from action handler:

javascript

```
analyzeImageAction.registerRunListener(async (args, state) => {
  const imageToken = args.image_input;
  const imageStream = await imageToken.getStream();

  const analysisResult = await this.analyzeImage(imageStream);

  // Return tokens for Advanced Flow
  return {
    objects_found: analysisResult.objectCount,
    analysis_result: analysisResult.description
  };
});
```

Global Tokens

Create app-wide global tokens:

javascript

```
class YourApp extends Homey.App {

  async onInit() {
    // Create global token for current app status
    this._statusToken = await this.homey.flow.createToken('app_status', {
      type: 'string',
      title: { en: 'App Status' }
    });

    await this._statusToken.setValue('Initializing');

    // Create global token for device count
    this._deviceCountToken = await this.homey.flow.createToken('device_count', {
      type: 'number',
      title: { en: 'Connected Devices' }
    });

    await this.updateDeviceCount();
  }

  async updateDeviceCount() {
    const devices = this.homey.drivers.getDevices();
    await this._deviceCountToken.setValue(devices.length);
  }

  async updateAppStatus(status) {
    await this._statusToken.setValue(status);
    this.log(`App status updated to: ${status}`);
  }
}
```

Image Tokens

Working with image tokens:

javascript

```
class SecurityApp extends Homey.App {

  async captureSnapshot(camera) {
    // Create image from camera
    const imageBuffer = await camera.takeSnapshot();
    const image = await this.homey.images.createImage();
    image.setBuffer(imageBuffer, 'jpeg');

    // Trigger with image token
    const tokens = { snapshot: image };
    await this._snapshotTakenTrigger.trigger(camera, tokens, {});
  }

  registerImageAction() {
    const saveImageAction = this.homey.flow.getActionCard('save_image');

    saveImageAction.registerRunListener(async (args, state) => {
      const imageToken = args.image_input; // Image from droptoken
      const filename = args.filename;

      // Get image stream
      const imageStream = await imageToken.getStream();
      this.log(`Saving ${imageStream.contentType} as ${imageStream.filename}`);

      // Save to file
      const fs = require('fs');
      const path = require('path');
      const targetPath = path.join(this.homey.platformVersion >= 3 ? '/userdata' : __dirname, filename);
      const writeStream = fs.createWriteStream(targetPath);

      imageStream.pipe(writeStream);

      return new Promise((resolve, reject) => {
        writeStream.on('finish', resolve);
        writeStream.on('error', reject);
      });
    });
  }
}
```

5. Flow State and Trigger Validation

For triggers with arguments, validate that the current state matches user-selected criteria:

Flow state validation example:

json

```
{
  "id": "temperature_threshold",
  "title": { "en": "Temperature crosses threshold" },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "capabilities=measure_temperature"
    },
    {
      "name": "threshold",
      "type": "number",
      "min": -50,
      "max": 100
    },
    {
      "name": "direction",
      "type": "dropdown",
      "values": [
        { "id": "above", "title": { "en": "Above" } },
        { "id": "below", "title": { "en": "Below" } }
      ]
    }
  ]
}
```

Implement state validation:

javascript

```
registerTriggerCards() {
  this._temperatureThresholdTrigger = this.homey.flow.getTriggerCard('temperature_threshold');

  // Register run listener for argument validation
  this._temperatureThresholdTrigger.registerRunListener(async (args, state) => {
    // args = user-selected values { device: deviceObj, threshold: 25, direction: "above" }
    // state = current trigger state { device: deviceObj, currentTemp: 27, previousTemp: 23 }

    const userDevice = args.device;
    const userThreshold = args.threshold;
    const userDirection = args.direction;

    const stateDevice = state.device;
    const currentTemp = state.currentTemp;
    const previousTemp = state.previousTemp;

    // Only run this flow if it matches the user's device
    if (userDevice.getData().id !== stateDevice.getData().id) {
      return false;
    }

    // Check if threshold was crossed in the user-specified direction
    if (userDirection === 'above') {
      return previousTemp <= userThreshold && currentTemp > userThreshold;
    } else {
      return previousTemp >= userThreshold && currentTemp < userThreshold;
    }
  });
}

// Trigger with state validation
async onTemperatureChanged(device, newTemp, oldTemp) {
  const state = {
    device: device,
    currentTemp: newTemp,
    previousTemp: oldTemp
  };

  const tokens = {
    temperature: newTemp,
    previous_temperature: oldTemp
  };

  // This will only execute flows where the state validation returns true
```

```
  await this._temperatureThresholdTrigger.trigger(tokens, state);
}
```

6. Monitoring Flow Card Usage

Subscribe to argument changes:

javascript

```
registerTriggerCards() {
  const weatherTrigger = this.homey.flow.getTriggerCard('weather_changed');

  // Monitor when users change arguments
  weatherTrigger.on('update', async () => {
    this.log('Weather trigger arguments updated');

    // Get all current argument values
    const argumentValues = await weatherTrigger.getArgumentValues();
    this.log('Current weather locations being monitored:', argumentValues);

    // Update your app's monitoring based on new arguments
    const locations = argumentValues.map(args => args.location);
    await this.updateWeatherMonitoring(locations);
  });
}
```

7. Flow Card Management and Organization

Highlighted Flow Cards

Promote important cards to the top of the card list:

json

```
{
  "id": "emergency_stop",
  "title": { "en": "Emergency stop" },
  "highlight": true,
  "args": [...]
}
```

Best practices for highlighting:

- Only highlight 2-3 most important cards
- Choose cards users will use frequently
- Don't over-highlight - it defeats the purpose

Advanced Flow Only Cards

Restrict cards to Advanced Flow users only:

```json
{
  "id": "complex_automation",
  "title": { "en": "Complex automation sequence" },
  "advanced": true,
  "args": [...]
}
```

When to use advanced-only:

- Complex cards that might confuse basic users
- Cards requiring deep technical knowledge
- Power-user features

Deprecating Flow Cards

Gradually phase out old cards without breaking existing flows:

```json
{
  "id": "old_temperature_control",
  "title": { "en": "Set temperature (legacy)" },
  "deprecated": true,
  "hint": { "en": "This card is deprecated. Use 'Set temperature' instead." },
  "args": [...]
}
```

Deprecation strategy:

1. Mark as deprecated (still works, hidden from add list)
2. Provide migration path in hint
3. Eventually remove in major version update

8. Complete Example: Professional Weather App Flow Cards

Here's a comprehensive example combining all advanced features:

.homeycompose/flow/triggers/weather_alert.json:

```json
```

```
{
  "id": "weather_alert",
  "title": { "en": "Weather alert received" },
  "titleFormatted": { "en": "Weather alert for [[location]]: [[alert_type]]" },
  "highlight": true,
  "hint": { "en": "Triggers when severe weather alerts are issued for your location" },
  "args": [
    {
      "name": "location",
      "type": "autocomplete",
      "title": { "en": "Location" },
      "placeholder": { "en": "Enter city name" }
    },
    {
      "name": "alert_type",
      "type": "dropdown",
      "title": { "en": "Alert type" },
      "values": [
        { "id": "any", "title": { "en": "Any alert" } },
        { "id": "storm", "title": { "en": "Storm warning" } },
        { "id": "flood", "title": { "en": "Flood warning" } },
        { "id": "heat", "title": { "en": "Heat wave" } }
      ]
    }
  ],
  "tokens": [
    {
      "name": "severity",
      "type": "string",
      "title": { "en": "Alert severity" },
      "example": "severe"
    },
    {
      "name": "description",
      "type": "string",
      "title": { "en": "Alert description" },
      "example": "Heavy rain expected"
    },
    {
      "name": "alert_image",
      "type": "image",
      "title": { "en": "Weather radar image" }
    }
  ]
}
```

.homeycompose/flow/conditions/weather_condition.json:

json

```
{
  "id": "weather_condition",
  "title": { "en": "!{{Weather condition is...|Weather condition is not...}}" },
  "titleFormatted": { "en": "!{{Weather in [[location]] is [[condition]]|Weather in [[location]] is not [[condition]]}}" },
  "args": [
    {
      "name": "location",
      "type": "autocomplete",
      "title": { "en": "Location" },
      "placeholder": { "en": "Enter city name" }
    },
    {
      "name": "condition",
      "type": "dropdown",
      "title": { "en": "Weather condition" },
      "values": [
        { "id": "sunny", "title": { "en": "Sunny" } },
        { "id": "cloudy", "title": { "en": "Cloudy" } },
        { "id": "rainy", "title": { "en": "Rainy" } },
        { "id": "stormy", "title": { "en": "Stormy" } }
      ]
    }
  ]
}
```

.homeycompose/flow/actions/weather_report.json:

json

```
{
  "id": "generate_weather_report",
  "title": { "en": "Generate weather report" },
  "titleFormatted": { "en": "Generate weather report for [[location]]" },
  "advanced": true,
  "duration": true,
  "args": [
    {
      "name": "location",
      "type": "autocomplete",
      "title": { "en": "Location" },
      "placeholder": { "en": "Enter city name" }
    },
    {
      "name": "include_forecast",
      "type": "checkbox",
      "title": { "en": "Include 7-day forecast" },
      "required": false
    },
    {
      "name": "report_format",
      "type": "dropdown",
      "title": { "en": "Report format" },
      "values": [
        { "id": "summary", "title": { "en": "Summary" } },
        { "id": "detailed", "title": { "en": "Detailed" } },
        { "id": "technical", "title": { "en": "Technical" } }
      ]
    }
  ],
  "tokens": [
    {
      "name": "report_text",
      "type": "string",
      "title": { "en": "Weather report" }
    },
    {
      "name": "report_image",
      "type": "image",
      "title": { "en": "Weather chart" }
    }
  ]
}
```

This example showcases:

- **Highlighted important trigger** for weather alerts
- **Inverted condition** with natural language
- **Advanced-only action** with duration support
- **Multiple argument types** (autocomplete, dropdown, checkbox)
- **Optional arguments**
- **Image and text tokens** for rich data passing
- **Professional naming** and descriptions

3. Device-Specific Flow Cards

There are two ways to create device-specific flow cards in Homey:

A. App-Level Cards with Device Arguments (Recommended)

You can make app-level flow cards device-specific by using device arguments with filters. This is often the preferred approach:

`.homeycompose/flow/triggers/temperature_changed.json`:

```json
```

```
{
  "id": "temperature_changed",
  "title": {
    "en": "Temperature changed"
  },
  "titleFormatted": {
    "en": "Temperature of [[device]] changed to [[temperature]]°C"
  },
  "args": [
    {
      "name": "device",
      "type": "device",
      "filter": "driver_id=temperature_sensor"
    }
  ],
  "tokens": [
    {
      "name": "temperature",
      "type": "number",
      "title": {
        "en": "Temperature"
      }
    }
  ]
}
```

Multiple driver filter example:

```
json

{
  "name": "device",
  "type": "device",
  "filter": "driver_id=temperature_sensor|humidity_sensor|multi_sensor"
}
```

Capability-based filter example:

```
json

{
  "name": "device",
  "type": "device",
  "filter": "capabilities=measure_temperature"
}
```

B. True Device-Specific Cards in driver.flow.compose.json

Device-specific flow cards can also be defined in the `driver.flow.compose.json` file within each driver directory:

```
drivers/
└── temperature-sensor/
    ├── driver.js
    ├── device.js
    ├── driver.compose.json (driver configuration)
    └── driver.flow.compose.json (device-specific flow cards)
```

Important: Note the distinction between:

- `driver.compose.json` - Contains driver configuration (capabilities, class, etc.)
- `driver.flow.compose.json` - Contains device-specific flow card definitions

```
drivers/
└── temperature-sensor/
    ├── driver.js
    ├── device.js
    └── driver.compose.json
```

`drivers/temperature-sensor/driver.flow.compose.json`:

```json
```

```
{
  "id": "temperature_sensor",
  "name": {
    "en": "Temperature Sensor"
  },
  "class": "sensor",
  "capabilities": [
    "measure_temperature",
    "alarm_battery"
  ],
  "flow": {
    "triggers": [
      {
        "id": "temperature_threshold_exceeded",
        "title": {
          "en": "Temperature threshold exceeded"
        },
        "args": [
          {
            "name": "threshold",
            "type": "number",
            "min": -50,
            "max": 100,
            "step": 0.1
          }
        ]
      }
    ],
    "conditions": [
      {
        "id": "is_temperature_stable",
        "title": {
          "en": "Temperature is stable"
        }
      }
    ],
    "actions": [
      {
        "id": "calibrate_sensor",
        "title": {
          "en": "Calibrate sensor"
        },
        "args": [
          {
            "name": "offset",
            "type": "number",
```

```
        "min": -10,
        "max": 10,
        "step": 0.1
      }
    ]
  }
 ]
}
}
```

C. When to Use Each Approach

Use App-Level Cards with Device Arguments when:

- You want the same flow card to work across multiple device types
- You need centralized flow card management
- The flow card logic is generic and can be shared
- You want to filter by capabilities rather than specific drivers

Use driver.flow.compose.json when:

- The flow card is very specific to one device type
- You need different flow card IDs for different drivers
- The card behavior varies significantly between device types
- You want the cards to only appear for that specific driver

D. Device Argument Filter Examples

Filter by single driver:

```json
{
  "name": "device",
  "type": "device",
  "filter": "driver_id=my_thermostat"
}
```

Filter by multiple drivers:

```json
```

```
{
  "name": "device",
  "type": "device",
  "filter": "driver_id=thermostat_v1|thermostat_v2|smart_thermostat"
}
```

Filter by device class:

```json
{
  "name": "device",
  "type": "device",
  "filter": "class=thermostat"
}
```

Filter by capabilities:

```json
{
  "name": "device",
  "type": "device",
  "filter": "capabilities=target_temperature&measure_temperature"
}
```

Complex filter combining multiple criteria:

```json
{
  "name": "device",
  "type": "device",
  "filter": "driver_id=smart_thermostat&capabilities=target_temperature"
}
```

```javascript
```

```
class TemperatureSensorDevice extends Homey.Device {

  async onInit() {
    this.log('Temperature sensor device initialized');

    // Register capability listeners
    this.registerCapabilityListener('measure_temperature', this.onTemperatureChanged.bind(this));

    // Register device-specific flow cards
    this.registerDeviceFlowCards();
  }

  registerDeviceFlowCards() {
    // Device-specific trigger
    this._deviceTemperatureChangedTrigger = this.homey.flow.getDeviceTriggerCard('device_temperature_change

    // Device-specific condition
    const deviceTempCondition = this.homey.flow.getDeviceConditionCard('device_temperature_above');
    deviceTempCondition.registerRunListener(async (args, state) => {
      const targetTemp = args.temperature;
      const currentTemp = this.getCapabilityValue('measure_temperature');
      return currentTemp > targetTemp;
    });

    // Device-specific action
    const setDeviceModeAction = this.homey.flow.getDeviceActionCard('set_device_mode');
    setDeviceModeAction.registerRunListener(async (args, state) => {
      const mode = args.mode;
      await this.setMode(mode);
    });
  }

  async onTemperatureChanged(value, opts) {
    this.log(`Temperature changed to ${value}°C`);

    // Trigger flow card
    const tokens = { temperature: value };
    const state = {};

    await this._deviceTemperatureChangedTrigger.trigger(this, tokens, state);
  }
}
```

Error Handling and Validation

1. Input Validation

javascript

```javascript
setTemperatureAction.registerRunListener(async (args, state) => {
  const device = args.device;
  const temperature = args.temperature;

  // Validate inputs
  if (!device) {
    throw new Error('No device selected');
  }

  if (temperature < 5 || temperature > 35) {
    throw new Error('Temperature must be between 5°C and 35°C');
  }

  if (!device.hasCapability('target_temperature')) {
    throw new Error('Device does not support temperature control');
  }

  try {
    await device.setCapabilityValue('target_temperature', temperature);
    return true;
  } catch (error) {
    this.error('Failed to set temperature:', error);
    throw new Error(`Failed to set temperature: ${error.message}`);
  }
});
```

2. Async Error Handling

javascript

```
temperatureAboveCondition.registerRunListener(async (args, state) => {
  try {
    const device = args.device;
    const targetTemperature = args.temperature;

    // Check if device is available
    if (!device.getAvailable()) {
      this.log('Device is not available');
      return false;
    }

    const currentTemperature = device.getCapabilityValue('measure_temperature');

    if (currentTemperature === null || currentTemperature === undefined) {
      this.log('No temperature reading available');
      return false;
    }

    return currentTemperature > targetTemperature;

  } catch (error) {
    this.error('Error in temperature condition:', error);
    return false; // Conditions should return false on error
  }
});
```

Testing Flow Cards

1. Development Testing

javascript

```javascript
// In your app's onInit method
if (process.env.NODE_ENV === 'development') {
  this.testFlowCards();
}

async testFlowCards() {
  this.log('Testing flow cards...');

  // Test trigger
  setTimeout(() => {
    this.triggerTemperatureChanged(null, 25.5);
  }, 5000);

  // Test condition
  const conditionResult = await this.homey.flow.getConditionCard('temperature_above')
    .trigger({ device: mockDevice, temperature: 20 });
  this.log('Condition result:', conditionResult);
}
```

2. Mock Devices for Testing

```javascript
const mockDevice = {
  getCapabilityValue: (capability) => {
    if (capability === 'measure_temperature') return 23.5;
    return null;
  },
  setCapabilityValue: async (capability, value) => {
    this.log(`Mock: Setting ${capability} to ${value}`);
    return true;
  },
  hasCapability: (capability) => true,
  getAvailable: () => true
};
```

Best Practices

1. File Organization

For larger projects, organize flow cards logically:

```
.homeycompose/flow/
├── triggers/
│   ├── temperature/
│   │   ├── temperature_changed.json
│   │   ├── temperature_threshold.json
│   │   └── temperature_alert.json
│   ├── motion/
│   │   ├── motion_detected.json
│   │   └── motion_stopped.json
│   └── system/
│       ├── app_started.json
│       └── connection_lost.json
├── conditions/
│   ├── temperature_above.json
│   ├── is_night_mode.json
│   └── device_available.json
└── actions/
    ├── climate/
    │   ├── set_temperature.json
    │   └── set_mode.json
    └── notifications/
        ├── send_push.json
        └── send_email.json
```

2. Localization with .homeycompose

Combine `.homeycompose` structure with proper localization:

`locales/en.json`:

```json
{
  "flows": {
    "triggers": {
      "temperature_changed": {
        "title": "Temperature changed",
        "titleFormatted": "Temperature changed to [[temperature]]°C"
      }
    }
  }
}
```

`locales/nl.json`:

```json
```

```
{
  "flows": {
    "triggers": {
      "temperature_changed": {
        "title": "Temperatuur veranderd",
        "titleFormatted": "Temperatuur veranderd naar [[temperature]]°C"
      }
    }
  }
}
```

3. Performance Optimization

- Cache frequently accessed data
- Use debouncing for rapid triggers
- Implement proper cleanup in onUninit

```javascript
```

```
class YourApp extends Homey.App {
  constructor() {
    super();
    this._temperatureCache = new Map();
    this._debounceTimers = new Map();
  }

  debounceTemperatureChange(deviceId, temperature, delay = 1000) {
    // Clear existing timer
    if (this._debounceTimers.has(deviceId)) {
      clearTimeout(this._debounceTimers.get(deviceId));
    }

    // Set new timer
    const timer = setTimeout(() => {
      this.triggerTemperatureChanged(deviceId, temperature);
      this._debounceTimers.delete(deviceId);
    }, delay);

    this._debounceTimers.set(deviceId, timer);
  }

  async onUninit() {
    // Cleanup timers
    for (const timer of this._debounceTimers.values()) {
      clearTimeout(timer);
    }
    this._debounceTimers.clear();
  }
}
```

2. Internationalization

javascript

```
// In locales/en.json
{
  "flow": {
    "triggers": {
      "temperature_changed": {
        "title": "Temperature changed",
        "titleFormatted": "Temperature changed to [[temperature]]°C"
      }
    }
  }
}

// In locales/nl.json
{
  "flow": {
    "triggers": {
      "temperature_changed": {
        "title": "Temperatuur veranderd",
        "titleFormatted": "Temperatuur veranderd naar [[temperature]]°C"
      }
    }
  }
}
```

3. State Management

javascript

```
// Use state to pass data between flow cards
const triggerCard = this.homey.flow.getTriggerCard('motion_detected');
const state = {
  zoneId: motion.zoneId,
  deviceId: motion.deviceId,
  timestamp: Date.now()
};

await triggerCard.trigger(tokens, state);

// In condition card, access state
conditionCard.registerRunListener(async (args, state) => {
  const zoneId = state.zoneId;
  // Use state data for more context-aware conditions
});
```

4. Documentation

Always document your flow cards:

```javascript
/**
 * Temperature Changed Trigger
 *
 * Triggered when a temperature sensor reports a new value
 *
 * @param {Object} tokens - Contains temperature value
 * @param {number} tokens.temperature - The new temperature in Celsius
 * @param {Object} state - Contains device context
 * @param {string} state.deviceId - The device that triggered the change
 */
```

Debugging and Validation

1. Validation Commands

```bash
# Validate your .homeycompose structure
homey app validate

# Build and run with live reload
homey app run

# Build for production
homey app build

# Install on local Homey for testing
homey app install
```

2. Common Validation Errors

Invalid JSON syntax:

```bash
Error: Invalid JSON in .homeycompose/flow/triggers/temperature_changed.json
```

Missing required fields:

```bash
Error: Flow card 'temperature_changed' is missing required field 'title'
```

Invalid argument types:

```bash
Error: Argument 'temperature' has invalid type 'string', expected 'number'
```

3. Development Workflow Tips

1. **Use extensive logging**:

```javascript
this.log('Flow card triggered with args:', JSON.stringify(args));
```

2. **Validate .homeycompose changes**:

```bash
# After modifying flow cards
homey app validate
homey app run
```

3. **Test incremental changes**:

- Change one flow card at a time
- Test thoroughly before moving to the next
- Use `homey app log` to monitor execution

4. **Check generated app.json**:

- Verify the build process worked correctly
- Ensure all your flow cards are present
- Check for any merge conflicts or missing data

Conclusion

This comprehensive tutorial covers the complete spectrum of Homey flow card development, from basic concepts to advanced professional techniques. You now understand:

Core Concepts:

- Flow card types (triggers, conditions, actions) and their purposes
- Proper file organization using `.homeycompose` structure
- Device filtering and capability-aware development
- Built-in vs custom flow cards strategy

Advanced Features:

- Complete argument type reference (text, number, date, time, color, range, etc.)
- Inverted flow cards for natural language flows
- Image tokens and global tokens for rich data sharing
- Flow state validation and argument change monitoring
- Professional flow card management (highlighting, deprecation, advanced-only)

Development Best Practices:

- Capability-aware argument validation and ranges
- Robust error handling and user feedback
- Performance optimization and resource management
- Testing strategies and debugging techniques
- Internationalization and accessibility

Professional Architecture:

- JSON extends patterns for code reuse
- Device-specific customization strategies
- Flow card lifecycle management
- Integration with Homey's ecosystem

Creating effective Homey flow cards requires careful planning of the user experience, robust error handling, comprehensive testing, and deep understanding of Homey's flow system. Focus on making your flow cards intuitive, reliable, and professionally integrated with the platform.

The techniques covered in this tutorial will help you create flow cards that feel native to Homey while providing powerful, device-specific functionality that enhances users' home automation experiences. Remember to leverage automatic capability flow cards where appropriate, and layer on custom cards for specialized features and branded experiences.

Your flow cards are often the primary way users interact with your app's functionality - investing in their quality and user experience will significantly impact your app's success and user satisfaction.