# API

# Integrations basics

Integration is anything which enables chatbot to connect with outside world. It could be initiated by the chatbot using step on certain point in the chatbot tree or in the opposite by some external system by triggering some of bot APIs.

# Integration step

Integration step could be added to any place of chatbot tree and is triggered in the moment when any user is passing given point in the conversation. It can call any of supported APIs, pass parameters inside (static or dynamic based on data we already know about the user using variables) and store results for further usage.

## Use-cases

- Fetching some data from system to show (in mesage, carousel, ...) or use for further decisions in the bot flow (for example show available job positions) - examples of use:
  - Show carousel of available job positions in ATS system
  - Show carousel of recommended products in eshop
  - Get maximum loan value based on values entered by the user
- Storing some data from bot to external system (add new candidate to selected position using data user entered)
  - Store candidate application to ATS system with all gathered information (name, email, phone, position ID)
  - Order some selected product in e-commerce platform
  - Create support ticked based on gathered clues in servicedesk systems like Jira
- Triggering some action in external system (trigger remote modem restart in internet connection helpdesk bot)

## Architecture

Every integration step type consists of two parts. First of them is JSON schema which describes what input parameters of which types are expected, which of them are required and what output structure should be expected. Second part is the code itself which implements:

- How is bot authenticated (in the most cases using some predefined token stored in the bot settings)

- How are parameters transformed and attached to the call
- How are output data transformed before saved into output variable
- Error handling
  - show fatal bot error to the user and stop conversation
  - trigger some predefined dialog
  - just log error and save some flag to user data variable so some decision could be made in the tree based on it

In ideal case, integration step performs just a single API call but in some advanced scenarios it is often needed to perform multiple calls and combine input/output data to fulfill desired goal.

# Advanced topics

## Timing

Integration step could behave in three different ways from the point of view of timing and performance.

- **Synchronous call**
  - User needs to wait for completion of API call during conversation
  - All data returned from API could be saved and used further
  - Data are not stored in bot database so filtering/searching should be performed on target system and just required subset of data should be returned
  - System should be stable and response should be fast (max. few seconds) to not let user wait too long
- **Asynchronous call**
  - Fire the request but do not wait for the result and continue immediatelly
  - Can be used to trigger some not-critical action where we do not need to know the result
  - For example track event into some analytics system
- **Call on background**
  - Data could be downloaded on background and stored into bot's database
  - They are later accessed locally by integration step during conversation quickly without need of any other system
  - Bot needs to filter/search data by itself which has limited performance (max. thousands of rows) and features (only filter by by tags or by locality radius)
  - Works also fine for slow and unstable data sources
  - There could be several ways how to trigger background data update
    - Periodical pull (reliable but consumes an unnecessarily large amount of resources)
    - Push (external server pushes data to the bot when some change occurs - always all of them or incrementally)
    - Pull on change (external server just signals bot that there is some change and bot do the same as during periodical pull)

## Level of abstraction

It is very important to consider which level of abstraction use during the process of integration step design. For example imagine that we have some system called "MyCRM" with a lot of modules with REST API which can perform insert/update/delete operations on any of them when every module has different set of fields. Goal of prepared integration is just to add customer with given name and email. Now we have at least three levels of abstraction to choose from:

- **Action level**
  - The most specific, single purpose, very easy usage, need to change the code during every change of logic
  - Integraton called `mycrm-add-contract`
  - Parameters `name=...` and `email=...`
  - Returns `id=...` which could be used directly
- **Protocol level**
  - The most generic, versatile, difficult usage, do not need to change the code when logic changes, API needs to support such generic format
  - Integration called `rest`
  - Parameters `endpoint=mycrm.com/api/v1/customers`, `method=post`, `body={"name":"...","email":"..."}`
  - Returns `{"response": {"body": {"id":"...", ...}, ...}, ...}` which needs to be parsed
  - For such purposes, there is already an integration step called HTTP GET/POST request
- **Service level**
  - Balance between two levels above
  - Integraton called `mycrm`
  - Parameters `module=customers`, `action=add`, `body={"name":"...","email":"..."}`
  - Returns `{"id":"...", ...}` which needs some simple parsing

It could not be said in a generic way which of these levels is correct. It always depends on given use-case, specs of used API and plans for future modifications of logic by non-programmers.

# Channels

Channel is used to transfer messages from bot the user and back. We have some channel integrations built in the product itself, but it is possible to integrate any custom chat channel with proper API. There are three ways how to do it:

- Feedyou will implement connector to present API of the new channel
- Channel will implement connection to generic Microsoft Bot REST API which is already supported (more info here)

- Channel will be part of some channel-aggregation platform which is already supported by Feedyou such as MS Bot Service or Mluvii

Some chat channels also support transferring events during the same "pipe" which is used for messages. Example could be WebChat component which allows:

- during initialization of chat component on the web page, any data we know about the user (such as email, selected page, auth token, ...) could be passed in and then can be further used in the chatbot tree for decision or passed into other integrations (for example auth token passed from the web page could be used for API authentication with user-scoped access permissions)
- when any event happens for page (for example user clicks some element, fill form field, etc.), webchat can send event to the bot to save some user data, start selected dialog and so on
- in the opposite when user reaches some point in the tree, it is possible to send event from bot to the webchat component and trigger predefined listener to perform any code in the page (for example close chat, reload page, ...)

# Dialog trigger

External system can trigger selected dialog for given user by calling Trigger API which has following parameters:

- Selected user address (probably previously stored in some output integration, some segmentation filter could be also available in the future instead of single ID)
- Dialog ID to trigger
- Optionally list of user data to preset before triggering dialog

You can read more in Trigger API documentation.

# User data export

There is an API exposed by every running chatbot which could be used to retrieve all stored data for every user. This API is mainly used for analytic purposes (such as how many users have passed certain point in the tree - and thus have given storage filled) but also could be used for anything else. There are available basic parameters that could be used to filter out only given subset of users.

You can read more in Export API documentation.

# Preset integrations

There is already a lot of integrations already implemented in Feedbot Designer. You can check this spreadsheet to find out more. It is important that every system could be integrated in large number of different ways with different options so it is needed to consider given use-case carefully.

# API automation platforms

Not all integrations needs to be programmed directly in the bot's code. Different kinds of automation/integration platforms or tools could be used to connect different services including bot to implement needed logic. Good examples of these platforms could be:

- Integromat
- Zapier
- Microsoft Azure Logic App
- ... and many others

# Adding new integrations

Following list tells what is often needed to have to allow new integration to be implemented:

- Well described use-case incl. the way how integration should be used in the chatbot tree such as input/output variables and how they will be used on other parts of the tree
- API specification incl. authentication (if target system currently do not have any and is going to prepare some just for the chatbot, take a look at the example how we think this API should look like )
- Example credentials to be able to test integration during implementation against real system

# Export API

Base URL for export API is bot-specific `https://feedbot-${BotId}.azurewebsites.net`. Please use the `?code=...` query param or `x-functions-key: ...` HTTP header to perform token based authentication.

## Get users data

> Calling this endpoint without timestamp filters can return very large amount of data and affect bot performance. Regular daily fetching of data from the past day is recommended approach.

`GET /api/management/export/userData`

Returns list of objects containing all user-scoped data which bot persists. Optional filters parameters:

- `notEmpty` use comma separated list of fields (in *camelCase* format) which should not be empty to include given user in the output list
- `fromTimestamp` and `toTimestamp` second-based UNIX timestamp to limit users in the output based on the date of last message
- `where...` where `...` is name of field which should be equal to the parameter value

| |
|---|
| `userId` |
| `timestamp` |
| `... all storages in camelCase ...` |

## Examples

- `GET https://feedbot-${BotId}.azurewebsites.net/api/management/export/userData?notEmpty=phone&fromTimestamp=1559901734` to get data of all users who have interacted with the bot after `06/07/2019 10:02:14` and have passed through question with `phone` storage
- `GET https://feedbot-${BotId}.azurewebsites.net/api/management/export/userData?whereGender=male&&code=${FunctionKey}` to get data of all male users

## Get users data snapshot

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/userDataSnapshot`

Returns list of objects containing all user-scoped data snapshots (generated by the "clear user data" action) which bot persists. Usage and all params are the same as in "Get users data" API.

| |
|---|
| `userId` |
| `timestamp` |
| `... all storages in camelCase ...` |

## Get NLP logs

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/nlpLog`

Returns list of objects containing all inputs entered by the user which was not possible to process using conversation tree. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| `processed` |
| --- |

## Get outgoing emails

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/outgoingEmail`

Returns list of objects containing all emails sent by the chatbot incl. recipient, subject, body. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| `userId` |
| --- |
| `timestamp` |
| `body` |
| `from` |
| `recipients` |
| `replyTo` |
| `stepId` |
| `subject` |

## Get outgoing SMS

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/outgoingSms`

Returns list of objects containing all SMS messages sent by the chatbot. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| |
|---|
| `link` |
| `message` |
| `provider` |
| `recipients` |

## Get fuzzy question answers log

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/fuzzyQuestionAnswerLog`

Returns list of objects containing all utterances processed by the fuzzy question answer plugin. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| |
|---|
| `userId` |
| `timestamp` |
| `intent` |
| `score` |
| `text` |
| `stepId` |
| `dialogId` |
| `matched` |
| `otherClassifications` |

## Get integration log

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/integrationLog`

Returns list of objects containing all integration invocations. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| |
|---|
| `userId` |
| `timestamp` |
| `url` |
| `method` |
| `requestBody` |
| `requestHeaders` |
| `responseStatus` |
| `responseBody` |
| `duration` |

## Get URL tracker

> This API method is available since bot hosting version v1.7.474

`GET /api/management/export/urlTracker`

Returns list of objects containing row for every click on URL button in bot by every user including target URL and domain. Desired time range could be specified using `fromTimestamp` and `toTimestamp` second-based UNIX timestamp.

| |
|---|
| `userId` |
| `timestamp` |
| `domain` |
| `url` |

# Trigger API

Pro vyvolání určitého dialogu pro vybraného uživatele ve virtuálním asistentovi iniciované externím systémem je možné využít Trigger API.

Každý bot běží na své doméně (většinou ve formátu `https://feedbot-${BOT_ID}.azurewebsites.net`) a je chráněn klíčem buď v `?code=...` URL query parametru nebo `x-functions-key: ...` HTTP hlavičce.

`POST /api/management/trigger/${ADDRESS_JSON_AS_BASE64}/${DIALOG_ID}/${MODE?}`

- parametry volání jsou:
  - adresa uživatele (objekt s propertami `bot`, `user`, `channelId` a `conversation?`) ve formátu JSON řetezce zakódovaného pomocí BASE64
  - ID dialogu, který se má spustit
  - volitelně mód spuštění, aktuálně podporovaný jen "clear", který nejprve smaže pozici uživatele ve stromu a pak teprve spustí dialog
- v těle požadavku může být JSON objekt s dodatečnými informacemi, které se uloží do dat uživatele
- reakci na ni je potřeba předem definovat v Designeru, kde je možné případně pomocí proměnných využít dodatečné informace

## Příklad použití

- uživatel si zažádá o převod kreditů na své věrnostní kartě na účet, přičemž tato operace může trvat několik minut
- virtuální asistent pomocí integračního kroku zavolá systém zákazníka vč. předání ID uživatele pro zahájení převodu
- uživatele informuje, že převod započal + spustí timeout na 10 min, který by uživatele informoval, že převod se nepovedlo dokončit v případě, že ze systému nepřijde do té doby odpověď
- pokud se převod povedl, systém zavolá na botovi Trigger API `POST /api/trigger/5ca39df/transfer-complete?code=...` s detaily transakce jako JSON tělem požadavku, což spustí dialog, který potvrdí uživateli že se vše povedlo a zobrazí detaily transakce
- pokud se převod nepovedl, systém zavolá na botovi Trigger API `POST /api/trigger/5ca39df/transfer-error?code=...` a virtuální asistent na to zareaguje chybovou zprávou s instrukcemi jak pokračovat dále

# External API recommendations

Do virtuálního asistenta je možné připravit libovolné množství "integračních kroků", což jsou body konverzačního stromu, kdy se virtuální asistent pomocí API integruje s jiným systémem za účelem získání nějakých dat nutných pro svou funkci nebo naopak nějaká data, která od uživatele zjistil do systému posílá.

Ve formátu a struktuře API se virtuální asistent může poměrně dost přizpůsobit, pokud by ale API vznikalo až kvůli němu, je možné se držet následujících doporučení, aby implementace byla co nejplynulejší.

## Doporučené vlastnosti API

- Autentizace ideálně jedním z následujících způsobů
  - username+password jako BASE64 string v hlavičce `Authorization: Basic ...`
  - token v hlaviččce `Authorization: Bearer ...`
    - může být jeden předem daný pro celého virtuálního asistenta
    - nebo pokud bot např. běží uvnitř systému, kam se uživatel přihlašuje, může být tento token předán botovi vždy se scopem daného uživatele, takže bot bude mít maximálně jeho oprávnění a ne vyšší
  - standardní OAuth2 `client_credentials` grant (tedy separátní požadavek pro získání a prodloužení accces tokenu, který se pak použije v `Authorization: Bearer ...` s každým požadavkem
- Ideálně se držet REST pravidel pro používané HTTP metody
  - **GET** pro získání dat
  - **POST** pro založení nového záznamu nebo spuštění nějaké akce (není idempotentní)
  - **PUT** pro aktualizaci existujícího záznamu (jeho identifikátor ideálně jako parametr URL)
  - **PATCH** pro částečnou aktualizaci existujícího záznamu (jeho identifikátor ideálně jako parametr URL)
  - **DELETE** pro odstranění existujícího záznamu (jeho identifikátor ideálně jako parametr URL)
- Tělo požadavku i odpovědi jako JSON
- Pokud to daná metoda vyžaduje, filtrování a řazení dat implementovat jako parametry URL adresy, nebo jako vlastní `X-...` HTTP hlavičky
- Maximální čas odezvy by bylo dobré držet pod 3 sekundami, protože v některých use-casech se API může volat synchronně v reálném čase a uživatel bude na výsledek čekat

## Ukázka

Virtuální asistent může pomocí API získat ze závislého systému seznam událostí, které má naplánované. ID uživatele jsou předány např. do WebChat komponenty uvnitř nějakého interního systému, kde je uživatel již autentizován. Data bot využije k zobrazení "carouselu" s naplánovanými událostmi, kde si

uživatel jednu z nich může vybrat a pokračovat nějakou další akcí.

```
GET https://somedomain.cz/api/v1/user/33/events

Authorization: Bearer ...
Content-type: application/json

→ 200 [{"id":"1", "name":"Onboarding meeting", "date":"2020-05-07T08:22:30.871Z"}]
```

Na konci komunikace může virtuální asistent odeslat informace o nově získaném uživateli do CRM. Součástí požadavku můžou být všechna data, která virtuální asistent o uživateli posbíral.

```
POST https://somedomain.cz/api/v1/user

Authorization: Bearer ...
Content-type: application/json
{"email": "user@example.com", "name":"Jack", "surname":"User", "interestedInProductIds":
[244, 234]}

→ 200 {"id":"433"}
```

# Custom channel API

API documentation

# JSON

JSON (JavaScript Object Notation) je nejčastěji používaný formát pro reprezentaci dat - používá se pro specifikaci API, configů, nebo se do něj dá ukládat celý bot - což mimo jiné náš případ

JSON slouží zejména k popisování vlastností objektů. Objekt je v tomto případě jakákoli věc o které chceme zjistit jaké vlastnosti má. Vlastnosti objektu pak popisujeme takto:

`"klíč": "hodnota"`

Například pokud bychom popisovali vlastnost auta tak jeho objekt mohli popsat takto:

```
{
  "color": "blue",
  "doorCount": 3,
  "brand": "Toyota",
  "engine": {
    "fuel": "petrol",
    "engineCapacity": 1.3,
    "cylinderCount": 4
  },
  "assets": ["AC", "GPS"]
}
```

Zde popisujeme vlastnosti auta jako jsou barva, počet dveří atd., kromě toho je zde zanořený objekt s vlastnostmi motoru a pole s doplňky auta - klimatizace a GPS

> Objekty a pole mohou obsahovat další objekty a pole, toto může sloužit k oddělení kontextu dat

# Typy v JSONu

- String
  - Textový řetězec (to co je v uvozovkách)
  - "Petr", "I", "Hello world!"
-

Number
  - Číslo včetně desetinných, záporných a vědeckých zápisů
  - 1, -10, 2.5, 1.2e10
- Boolean
  - Informace zda je daná vlastnost pravdivá/nepravdivá
  - true / false
- Null
  - Žádná hodnota
- Array
  - Pole/Seznam položek
  - [1, 2, 3], ["blue", "red", "green"]
- Object
  - Objekt s popisem vlastností
  - {"key": "value"}{"name": "Pavel", "age": 31}

# Přístup k hodnotám

K hodnotám v JSONu lze přistupovat přes tečkovou notaci, tedy pokud budeme mít například tento JSON:

```
{
  "color": "blue",
  "doorCount": 3,
  "brand": "Toyota",
  "engine": {
    "fuel": "petrol",
    "engineCapacity": 1.3,
    "cylinderCount": 4
  },
  "assets": ["AC", "GPS"]
}
```

a budeme chtít zjistit obsah motoru, tak toho docílíme tímto řádkem:

`car.engine.engineCapacity`

pokud budeme chtít zjistit jaká je první položka v seznamu doplňků, tak toho docílíme tímto řádkem:

`car.assets[0]`

První položku získáme přístupem do indexu 0, protože JavaScript čísluje pole od 0

# Podklady pro vytvoření integračního kroku

Ahoj, jsem tvůj checklist k tomu, aby se výroba integračního kroku obešla bez zbytečných prodlev. Drž se mě a za hodinu můžeš mít perfektní integrační krok a všichni budou nadšený . A to je skvělý, no ne?

Takže co teda budem potřebovat?

## 1. API URL klienta

Když chceme někam sáhnout pro data, nebo je někam zapsat, musíme vědět kam - k tomu nám slouží API url, která nás navede na rozhraní klienta, přes které nám poskytne data, nebo nám umožní je zapisovat.

## 2. API Key klienta

Protože bezpečnost je důležitá a klient ve většině případů nebude chtít, aby mu kdokoli mohl sáhnout do databáze, bude mít svoje API nějak zabezpečené, ve většině případů se bude jednat o zabezpečení s Bearer klíčem. Ten budeme muset od klienta dostat, abychom byli schopni zavolat API.

Příklad klíče: `Authorization: Bearer abcdef1234`

## 3. Specifikace API

Specifikace API je potřeba, abychom věděli z jakého koncového bodu, jaké informace posílat, nebo stahovat. Jedná se v podstatě o takovou "dohodu" mezi námi a klientem jaká data, a v jakém formátu, očekáváme. Součástí specifikace může, ale nemusí být API URL a API klíč, na toto není pravidlo a každý klient specifikaci pojímá jinak, proto je dobré se zeptat, zda tyto informace specifikace obsahuje, pokud neobsahuje tak je potřeba o ně požádat.

## 4. Příklad volání

Ke konečnému otestování je potřeba mít nějaká data, na kterých si dokážeme zjistit, jak získaná data reprezentovat v botovi. Tedy například pokud se jedná o bota pro zásilkovou službu a klient chce zobrazovat data o zásilce na základě zadaného čísla zásilky, tak budeme potřebovat nějakou testovací

zásilku abychom si ověřili, zda volání proběhlo v pořádku.

# Campaigns API

API specification