

UNIVERSITÀ POLITECNICA DELLE MARCHE

INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

PROGETTO DATA SCIENCE 2021-2022

---

**Implementazione di un chatbot tramite  
RASA Framework per ottenere  
informazioni sul mondo dei videogames**

---



*Studenti:*

MASSIMO CIAFFONI

SIMONE CAPPANERA

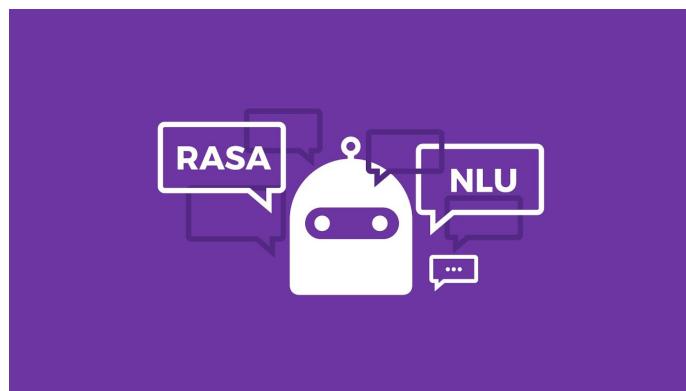
# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Implementazione . . . . .	3
1.2	Intents . . . . .	4
1.3	Entities . . . . .	5
1.3.1	Generazione degli esempi per le entities . . . . .	5
1.3.2	Utilizzo degli slot della memoria . . . . .	6
<b>2</b>	<b>Domain</b>	<b>7</b>
<b>3</b>	<b>Stories e Rules</b>	<b>9</b>
<b>4</b>	<b>Custom Actions</b>	<b>11</b>
4.1	Search Game . . . . .	11
4.2	Search Platforms . . . . .	14
4.3	Search Stores . . . . .	14
4.4	Search Screenshots . . . . .	15
4.5	Get Trailer . . . . .	16
4.6	Get Metacritic . . . . .	17
4.7	Search Publisher . . . . .	18
4.8	Reset Index . . . . .	19
4.9	Reset Slot . . . . .	20
<b>5</b>	<b>Esempi di utilizzo</b>	<b>21</b>

# Capitolo 1

## Introduzione

Per molti videogiocatori non è facile ottenere informazioni sui propri giochi preferiti in maniera intuitiva e veloce, in quanto è necessario recuperare le diverse informazioni. Per questo motivo si è deciso di sviluppare un chatbot tramite il framework **Rasa** e l'API gratuita **RAWG** per rendere disponibili le informazioni in un'unica fonte.



Le tecnologie utilizzate per questo progetto sono:

- **Rasa:** Framework open source Python per lo sviluppo e il rilascio di chatbot. <https://rasa.com/>
- **Ngrok:** Servizio di reverse proxy che espone all'esterno un servizio attivo in locale. <https://ngrok.com/>
- **RAWG:** Web API che fornisce dati aggiornati sui videogames. <https://rawg.io/apidocs>

## 1.1 Implementazione

La progettazione del chatbot è iniziata individuando quali sono le informazioni di maggiore interesse per un videogiocatore in modo da definire i diversi intent. Tale fase è stata influenzata dalle informazioni disponibili nell'API. Per ognuno degli intent sono stati definiti degli esempi di richieste piuttosto verosimili, in modo da definire un set di training adeguato. Il file di configurazione *config.yaml* (fig. 1.1) è stato modificato per fare in modo di integrare nella pipeline i classificatori di entities *Regex Entity Extractor* e *CRFEntityExtractor*. Per quanto riguarda invece le policy, è stata definita una regola che definisce un action di default eseguita dal chatbot nel caso in cui lo stesso non riesca ad interpretare la richiesta fatta dall'utente.

```
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: en

pipeline:
  # No configuration for the NLU pipeline was provided. The following default pipeline was used to train your model.
  # If you'd like to customize it, uncomment and adjust the pipeline.
  # See https://rasa.com/docs/rasa/tuning-your-model for more information.
  - name: WhitespaceTokenizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 8
  - name: DIETClassifier
    entity_recognition: False
    use_masked_language_model: False
    epochs: 300
    constrain_similarities: true
  - name: RegexEntityExtractor
    case_sensitive: True
    use_lookup_tables: True
    use_regexes: True
    use_word_boundaries: True
  - name: LexicalSyntacticFeaturizer
    features:
      - [EOS, digit]
      - [EOS, title, upper, low]
  - name: "CRFEntityExtractor"
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 200
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.4
    ambiguity_threshold: 0.1

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  # No configuration for policies was provided. The following default policies were used to train your model.
  # If you'd like to customize them, uncomment and adjust the policies.
  # See https://rasa.com/docs/rasa/policies for more information.
  - name: MemoizationPolicy
  - name: RulePolicy
    core_fallback_threshold: 0.4
    core_fallback_action_name: "action_my_fallback"
    enable_fallback_prediction: True
  - name: UnexpectTEDIntentPolicy
    max_history: 10
    epochs: 200
  - name: TEDPolicy
    max_history: 10
    epochs: 200
    constrain_similarities: true
```

Figura 1.1: File di configurazione del chatbot

**N.B:** Data la mole di istanze di entità da estrarre, i classificatori sono stati addestrati per un numero di epoche maggiori e i classificatori di entity sono stati scelti dopo una serie di test per verificare quale pipeline fosse la più accurata.

## 1.2 Intents

Analizzando le richieste possibili sull'API sono stati definiti diversi intent:

- **searchgame:** Richieste di informazioni generiche su un particolare videogioco. Il chatbot fornirà all'utente la data di uscita, gli sviluppatori, il publisher ed una breve descrizione del gioco. Esempi di input:
  - *What can you tell me about The Witcher 3?*
  - *Show me general info of this game*
  - *Show me info about Borderlands*
- **platform:** Richiesta al chatbot delle piattaforme dove il videogioco è disponibile. Esempi di input:
  - *On which platforms Overwatch is available?*
  - *Show me platforms*
  - *What are the platforms of God of War?*
- **screenshots:** Richiesta di screenshots di un particolare videogioco. Esempi di input:
  - *Show me some screens of this game*
  - *Can you show me some screens of Paladins?*
  - *I'd like to see some screenshots of Crash Bandicoot*
- **trailer:** Richiesta di trailer relativi ad un particolare videogioco. Esempi di input:
  - *Show me the trailer of this game*
  - *Trailer of Yakuza Kiwami*
  - *Show me the trailer of God of War Ragnarok*
- **stores:** Richiesta di informazioni sui diversi store in cui è possibile acquistare un particolare videogioco. Esempi di input:
  - *Where can I buy this game?*

- *On which store Final Fantasy VII is sold?*
  - *Where can I buy Tomb Raider?*
- **ratings:** Richiesta dello score metacritic di un videogioco e visualizzazione dell'ESRB tag assegnato. Esempi di input:
  - *Show me ratings of Borderlands 2*
  - *Can I get the score of this game?*
  - *ratings of Sekiro*
- **searchpublisher:** Richiesta di informazioni generiche di un particolare publisher. Esempi di input:
  - *What can you say me about Annapurna Games publisher?*
  - *How many games Square Enix published?*

Oltre agli intent specifici per la gestione delle richieste, sono stati definiti degli intent generici in grado di effettuare del semplice "*chitchatting*".

## 1.3 Entities

Per i diversi intent sono state utilizzate diverse entità descritte di seguito:

- **game:** entity per l'estrazione del nome del videogame di cui si vogliono richiedere le informazioni. I vari esempi da inserire nel file *nlu.yaml* sono stati estratti dall'API facendo una richiesta generica e salvando i risultati in un file *yaml*.
- **publisher:** entity per l'estrazione di un publisher di cui si vogliono ottenere delle informazioni.

### 1.3.1 Generazione degli esempi per le entities

Data l'elevata mole di videogame e di publisher esistenti, per ottenere un dataset abbastanza rappresentativo si sono creati degli script Python in grado di estrarre il maggior numero di dati dall'API. La query definita, infatti, permette di ottenere come risultato la lista di tutti i game/publisher presenti nel database. Per ogni elemento della lista lo script ottiene il nome e lo salva in un file *yaml*, il quale sarà utilizzato dal chatbot durante l'addestramento dei classificatori.

### 1.3.2 Utilizzo degli slot della memoria

Tutte le varie action/intent utilizzano lo slot della memoria in modo tale da permettere all'utente di chiedere informazioni su un particolare videogame senza dover ogni volta specificarne il nome. Oltre agli slot relativi alle entità, sono stati definiti degli slot custom, utili per la gestione delle diverse custom actions. Sono stati definiti tre custom slots:

- **index:** slot di tipo float per gestire la ricerca di un videogame. In alcuni casi, poiché l'API restituisce una lista di risultati, quello voluto dall'utente potrebbe non essere il primo della lista. Se ciò dovesse accadere, il valore di *index* verrà incrementato e la ricerca del videogame restituirà l'elemento successivo della lista (in ogni caso l'action custom di ricerca di un videogame permette di scorrere la lista solo fino al quinto elemento).
- **gameid:** slot di tipo float per memorizzare l'id di un videogame. Molto spesso, una volta trovato un videogame, l'utente vorrà ottenere ulteriori informazioni. In questo caso, l'id risultante dalla prima query di ricerca viene salvato in uno slot, in modo tale da semplificare le richieste successive (in questi casi, infatti, non è necessario ricercare il gioco nuovamente e dunque viene risparmiata una richiesta all'API).
- **current\_search:** slot di tipo text per gestire interazioni particolari in cui l'utente non interagisce con i bottoni mostrati dal chatbot, ma effettua una nuova richiesta testuale. In questi casi, poiché il *gameid* non è pari a zero, la richiesta effettua un controllo tra il nuovo nome cercato e quello precedente; se i due non coincidono allora il *gameid* sarà aggiornato con quello relativo alla nuova ricerca.

Nel momento in cui il videogioco restituito dal chatbot sia quello effettivamente richiesto dall'utente, il valore dello slot *index* sarà resettato a zero tramite una custom action. In questo modo richieste successive di videogame diversi restituiranno sempre il primo risultato della lista.

# Capitolo 2

## Domain

Il file *domain.yaml* permette di specificare tutti gli intent, le entities e le varie action che il chatbot deve gestire. In particolare:

- Per ognuno degli intent che il chatbot deve gestire si è specificato l'uso delle entities tramite l'opzione booleana *use\_entites*.
- Per ognuna delle entità definite si è creato uno slot di memoria, il quale specifica il tipo di entità da salvare nello slot e un valore iniziale di default. In particolare, gli slot delle entità *game* e *publisher* sono stati definiti di tipo text con valore iniziale pari a None; mentre gli slot custom *index* e *gameid* sono stati definiti di tipo float con un valore iniziale pari a 0.
- Oltre alle custom action, sono state definite delle risposte che il chatbot fornisce all'utente in particolare situazioni. Infatti, oltre alle risposte generiche, sono state definite delle responses in cui il chatbot può mostrare all'utente le sue funzionalità e una response che permette l'utilizzo di bottoni, in modo tale da garantire un corretto funzionamento del chatbot nel caso in cui l'utente voglia ottenere più informazioni su un particolare videogame.

```

utter_greet:
- text: Hey! How are you?
utter_cheer_up:
- image: https://images6.alphacoders.com/116/1162254.jpg
| text: 'Here is something to cheer you up:'
utter_did_that_help:
- text: Did that help you?
utter_happy:
- text: Great, carry on!
utter_goodbye:
- text: Bye
utter_iamabot:
- text: I am a bot, powered by Rasa.
utter_init:
- text: Do you want some information about videogames world?
utter_games:
- text: How can i help you?
utter_send_help:
- text: |
  I can show info about:
  GAMES:
    - General information about videogame.
    - Score and tag of the game.
    - Screenshot of the game.
    - Platforms where the game is avaivable.
    - Stores where the game is sold.
    - Trailer of the game.
  PUBLISHERS:
    - General information about publisher.

utter_fallback:
- text: Sorry! I couldn't get you!
utter_more_info:
- text: I'm glad that I help you. I can show you other info (type help to see how can i help you).
utter_other_info:
- text: Do you need other info about this game?
buttons:
- title: "Yes"
  payload: "/affirm"
- title: "No"
  payload: "/deny"
utter_tell_me_more:
- text: What do you wanna know about {game}?
..

```

Figura 2.1: Responses definite nel domain

# Capitolo 3

## Stories e Rules

Il file *stories.yaml* contiene delle possibili sequenze con cui gli utenti possono interagire con il chatbot. Oltre alle sequenze classiche già definite di default dal framework **Rasa**, si sono create per ognuna delle custom action relative ai videogames due diverse stories. La prima tipologia definisce il comportamento del chatbot nel caso di una normale interazione con l'utente, che una volta risposto al chatbot con il suo mood chiede delle informazioni generali su un particolare videogame, per poi richiedere, successivamente, ulteriori informazioni su un altro aspetto. Nel secondo caso, le stories definiscono una richiesta di un videogame che può essere susseguita da altre possibili richieste su tutte le informazioni che il chatbot può fornire. Un'ulteriore storia è stata aggiunta per gestire il caso in cui l'utente non ottenga come primo risultato il videogame voluto e dunque il chatbot effettua una nuova ricerca restituendo l'elemento successivo nella lista dei risultati. Infine, per quanto riguarda la ricerca di un publisher, si è creata una semplice story che definisce come normalmente un utente dovrebbe interagire con il chatbot per richiedere informazioni su un particolare publisher. Le stories definite sono sufficienti per gestire la maggior parte delle richieste degli utenti, in quanto il chatbot andrà a riconoscere non solo le sequenze complete definite dalle stories, ma anche tutte le possibili sottosequenze.

Nel file *rules.yaml*, invece, sono definite tutte le regole che il chatbot deve obbligatoriamente seguire. In particolare, in questo file, sono state definite delle regole custom, le quali dopo una richiesta di informazioni particolari eseguono correttamente una action. Nel caso in cui l'utente abbia espresso la volontà di terminare la ricerca, il chatbot effettuerà un reset degli slot e verrà riavviato mostrando all'utente il messaggio di start; mentre, nel caso in cui l'utente richieda ulteriori informazioni, il chatbot gli chiederà quali sono queste altre informazioni di cui necessita. Altre custom action sono state definite in modo tale che il reset degli indici o una nuova ricerca siano effettuati in maniera corretta.

```
- story: search_game path 1
  steps:
    - intent: greet
    - action: utter_greet
    - intent: mood_great
    - action: utter_happy
    - intent: utter_init
    - intent: affirm
    - action: utter_games
    - intent: search_game
    - action: action_search_game
    - intent: affirm
    - action: action_reset_index
    - action: utter_more_info
    - intent: platforms
    - action: action_search_platforms
    - action: utter_other_info
    - intent: deny
    - action: action_reset_slots
    - action: utter_start
```

Figura 3.1: Esempio di story

```
- rule: Show other info after store search
  steps:
    - intent: store
    - action: action_search_store
    - action: utter_other_info

- rule: Show other info after platform search
  steps:
    - intent: platforms
    - action: action_search_platforms
    - action: utter_other_info

- rule: Show other info after screenshot search
  steps:
    - intent: screenshots
    - action: action_search_screenshots
    - action: utter_other_info

- rule: Show other info after trailer search
  steps:
    - intent: trailer
    - action: action_get_trailer
    - action: utter_other_info
```

Figura 3.2: Esempi di rules

# Capitolo 4

## Custom Actions

Di seguito sono descritte nello specifico le diverse custom action definite per il chatbot.

### 4.1 Search Game

L'action di ricerca di un videogioco estrae dagli slot il nome del videogioco e l'indice di ricerca. Una volta estratte le entità, il chatbot effettua una richiesta all'API presso la route `/games`: la query è effettuata inserendo il valore dell'entità *name* nell'attributo *search* e specificando un valore di *search\_precise* pari a true, in modo da ottenere il risultato più corretto possibile. Dopo aver ottenuto il risultato, l'action estrae l'elemento all'i-esima posizione della lista (l'indice è specificato dallo slot *index*) ed estrae diverse informazioni come il nome completo e l'id del videogioco, i generi con cui il videogioco può essere classificato, la data di rilascio e infine un'immagine di background da restituire all'utente. A questo punto, l'action effettua una seconda richiesta al database, eseguendo in questo caso una query con l'id del videogame. Così facendo è possibile ottenere ulteriori informazioni specifiche da mostrare all'utente, come i publisher, gli sviluppatori ed una breve descrizione del videogame.

```

name = tracker.get_slot('game')
indice = tracker.get_slot('index')
print(name)
print(indice)
if(indice> 4):
    output="I didn't find your game try to ask me again later"
    dispatcher.utter_message(text=output)
    return[AllSlotsReset()]

r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
if r.status_code == 200:
    data = r.json()
    #print(len(data['results']))
    if indice > len(data['results']):
        output="I couldn't find the game. Try to use less characters"
    else:
        name = data['results'][indice]['name'] #nome videogame
        if 'None' in nome:
            output="I do not know anything about , what a mistery! Are you sure it is correctly spelled?"
            dispatcher.utter_message(text=output)
            return [AllSlotsReset()]
        else:
            image = data['results'][indice]['background_image']
            release_date = data['results'][indice]['released']
            genres = data['results'][indice]['genres']
            generi = []
            string_genres = ""
            for elem in genres:
                generi.append(elem['name'])
            for elem in generi:
                string_genres=string_genres+elem+" "
            id_game = data['results'][indice]['id']
            print(id_game)
            r2= requests.get(url='https://api.rawg.io/api/games/{}?key=bbac0252b5ed4a2b8286472063cb2dfe'.format(id_game))
            game_data = r2.json()
            print(game_data)
            try:
                publishers = game_data['publishers'][0]['name']
                print(publishers)
            except:
                publishers = "Not Available"
            try:
                developers = game_data['developers'][0]['name']
                print(developers)
            except:
                developers = "Not Available"
            print(publishers)
            print(developers)
            description=cleanhtml(game_data['description'])
            output="{} it's an {} game and it was developed by {} and published by {} on {}. {} . \n \n Is this the game you wanted? ".format(nome,
                string_genres,developers, publishers, release_date,description,image)
    else:
        output = "I do not know anything about , what a mistery! Are you sure it is correctly spelled?"
        dispatcher.utter_message(text=output)
return[SlotSet("index", indice+1), SlotSet("game_id", id_game), SlotSet("current_search", nome)]

```

Una volta terminata la ricerca, l'action modifica il valore degli slot, settando il *gameid* e incrementando di uno il valore dell'indice, in modo tale da gestire nuovamente la richiesta in caso il risultato voluto non sia il primo della lista. Ad esempio, la figura 4.1 mostra un caso possibile in cui la ricerca corretta sia la seconda della lista, infatti esistono due giochi con il nome *God of War* usciti rispettivamente nel 2005 e nel 2018.

**N.B.:** Nel caso in cui il risultato della ricerca risulti corretto, l'indice viene resettato tramite la action *reset\_index*.

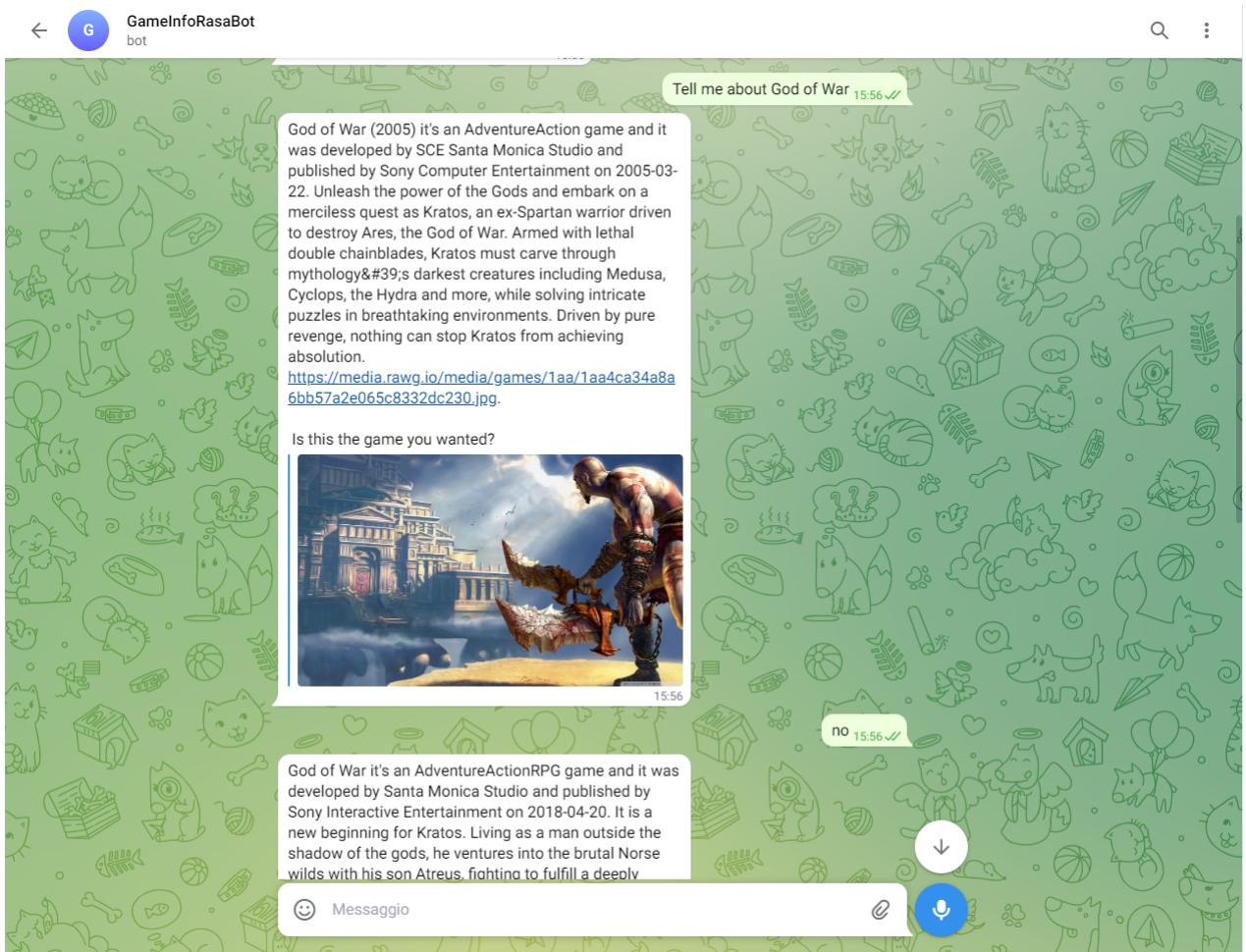


Figura 4.1: Esempio di ricerca non corretta

## 4.2 Search Platforms

L'action, una volta ottenuto il nome del videogame, effettua una ricerca tramite l'API in modo da ottenere una lista di piattaforme in cui il videogame è disponibile. Una volta ottenuto il risultato, il chatbot restituisce le varie piattaforme sotto forma di stringa.

```
class ActionSearchPlatforms(Action):

    def name(self) -> Text:
        return "action_search_platforms"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('game')
        indice = tracker.get_slot('index')
        if indice > 0:
            indice=indice -1
        print(name)
        print(indice)

        r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))

        if r.status_code == 200:

            data = r.json()
            nome = data['results'][indice]['name'] #nome videogame
            if 'None' in nome:
                output="I do not know anything about , what a mistery!? Are you sure it is correctly spelled?"
            piattaforme = []
            platforms = data['results'][indice]['platforms']
            if 'None' in platforms:
                output="I do not know anything about , what a mistery!? Are you sure it is correctly spelled?"
            else:
                for elem in platforms:
                    piattaforme.append(elem['platform']['name'])
            string_platforms = '\n'.join(str(elem) for elem in piattaforme)
            print(string_platforms)
            output="{} has been published on the following platforms: {}".format(nome, string_platforms)
        else:
            output = "I do not know anything about , what a mistery!? Are you sure it is correctly spelled?"
        dispatcher.utter_message(text=output)
        return []
```

Figura 4.2: Action di ricerca delle piattaforme di gioco

## 4.3 Search Stores

L'action, una volta estratto il nome, il *gameid* e la *current\_search*, effettua una query presso la route `/stores`, usando il *gameid* corrente come campo di ricerca, in modo da ottenere l'elenco degli store dove il videogame è venduto. In ogni caso, l'action effettua prima un confronto tra l'entità *name* e *current\_search*, in modo tale da gestire il caso in cui l'utente abbia specificato un gioco diverso e dunque l'id debba essere aggiornato. Una volta ottenuta la risposta, si estraggono tutti i link degli store in cui il gioco è disponibile e si restituiscono all'utente. L'action è in grado di gestire tutte le situazioni. Infatti, nel caso in cui la richiesta

degli store non sia preceduta da una ricerca del videogame (dunque il game id estratto sia pari a zero), verrà prima effettuata tale ricerca e estratto il rispettivo id, poi, successivamente, sarà effettuata nuovamente la ricerca per estrarre i link dei vari stores. In questo modo, come anticipato precedentemente, non è necessario effettuare di nuovo l'estrazione dell'id in quanto già salvato nello slot.

```
class ActionSearchStoreLink(Action):
    def name(self) -> Text:
        return "action_search_store"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('game')
        indice = tracker.get_slot('index')
        if indice > 0:
            indice=indice - 1
        game_id=tracker.get_slot('game_id')
        current=tracker.get_slot('current_search')
        print(name)
        if game_id==0:
            if name!=current:
                r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                if r.status_code ==200:
                    data = r.json()
                    game_id=data['results'][indice]['id']
                else: output = "I couldnt find any store which sells the game"
            else:
                if name==current:
                    r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                    if r.status_code ==200:
                        data = r.json()
                        game_id=data['results'][indice]['id']
                    else: output = "I couldnt find any store which sells the game"
                r = requests.get(url="https://api.rawg.io/api/games/{}/stores?key=bbac0252b5ed4a2b8286472063cb2dfe".format(game_id))

                if r.status_code ==200:
                    data = r.json()
                    store_urls = []
                    stores = data['results']
                    print(stores)
                    for elem in stores:
                        store_urls.append(elem['url'])
                    store_url='\n'.join(str(elem) for elem in store_urls)
                    output = "The game is sold in the following stores: {} ".format(store_url)
                else:
                    output = "I couldnt find any store which sells the game"
        dispatcher.utter_message(text=output)
        return [SlotSet("current_search", name), SlotSet('game_id', game_id)]
```

Figura 4.3: Action di ricerca degli store che vendono un videogame

## 4.4 Search Screenshots

L'action, in maniera simile alle precedenti, estraе le informazioni dagli slot ed effettua la ricerca degli screenshots del videogame presso la route /screenshots, utilizzando come campo della ricerca il game id. Una volta ottenuta la risposta dall'API, l'action estraе tutti gli screen ancora disponibili, infatti nella risposta è presente un campo *is\_deleted* che permette di verificare se i link delle immagini di cui si è fatta richiesta sono visualizzabili. Una volta salvati in un array, tutti i link vengono generati sotto forma di immagini nel client di Telegram grazie all'utilizzo della funzionalità *utter\_message* con specifica image. Anche in questo caso, l'action

gestisce entrambe le situazioni in cui la richiesta degli screenshots sia effettuata prima o dopo la ricerca del videogame.

```
class ActionSearchScreenshots(Action):
    def name(self) -> Text:
        return "action_search_screenshots"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('game')
        indice = tracker.get_slot('index')
        if indice > 0:
            indice=indice -1
        game_id=tracker.get_slot('game_id')
        current=tracker.get_slot('current_search')
        print(game_id)
        print(name)
        print(current)
        if game_id==0:
            if name!=current:
                r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                if r.status_code ==200:
                    data = r.json()
                    game_id=data['results'][indice]['id']
                else:
                    output = "I couldnt find any screenshots"
                    dispatcher.utter_message(text=output)
            else:
                if name!=current:
                    r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                    if r.status_code ==200:
                        data = r.json()
                        game_id=data['results'][indice]['id']
                    else:
                        output = "I couldnt find any screenshots"
                        dispatcher.utter_message(text=output)
        r3 = requests.get(url="https://api.rawg.io/api/games/{}/screenshots?key=bbac0252b5ed4a2b8286472063cb2dfe".format(game_id))

        if r3.status_code ==200:
            data3 = r3.json()
            image = []
            images = data3['results']
            #print(images)
            for elem in images:
                if elem['is_deleted'] == False:
                    image.append(elem['image'])
            print(image)
            if image is Empty:
                output = "I couldnt find any screenshots"
                dispatcher.utter_message(text=output)
            for elem in image:
                dispatcher.utter_message(image=elem)
        else:
            print('here')
            output = "I couldnt find any screenshots"
            dispatcher.utter_message(text=output)
        return [SlotSet("current_search", name), SlotSet('game_id', game_id)]
```

Figura 4.4: Action di ricerca di screenshots di un videogame

## 4.5 Get Trailer

L'action, una volta estratto il game id, effettua una query di ricerca presso la route `/movies`. Dopo aver ottenuto la risposta dall' API, il chatbot inserisce i diversi link dei vari trailer in un array e li mostra all'utente sotto forma di stringa. Nel caso in cui l'API non abbia trovato alcun risultato, il chatbot comunica all'utente che non sono presenti trailer per quel particolare videogame all'interno del database.

In ogni caso, il chatbot è in grado di gestire le diverse situazioni esattamente come descritto nelle action precedenti.

```

class ActionGetTrailer(Action):
    def name(self) -> Text:
        return "action_get_trailer"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('game')
        indice = tracker.get_slot('index')
        if indice > 0:
            indice=indice -1
        game_id=tracker.get_slot('game_id')
        current=tracker.get_slot('current_search')
        print(name)
        print(game_id)
        if game_id==0:
            if name==current:
                r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                if r.status_code ==200:
                    data = r.json()
                    game_id=data['results'][indice]['id']
                else: output = "I couldnt find any trailer of the game"
            else:
                if name==current:
                    r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                    if r.status_code ==200:
                        data = r.json()
                        game_id=data['results'][indice]['id']
                    else: output = "I couldnt find any trailer of the game"
                r = requests.get(url='https://api.rawg.io/api/games/{}/movies?key=bbac0252b5ed4a2b8286472063cb2dfe'.format(game_id))

                if r.status_code ==200:
                    data = r.json()
                    trailer_urls = []
                    trailers = data['results']
                    print(trailers)
                    for elem in trailers:
                        print(elem['data'][['max']])
                        trailer_urls.append(elem['data'][['max']])
                    trailer_url='\n'.join(str(elem) for elem in trailer_urls)
                    if trailer_url=='':
                        output= "I couldnt find any trailer of the game"
                    else:
                        output = "Here some trailers of this game: {}".format(trailer_url)
                else:
                    output = "I couldnt find any trailer of the game"

        dispatcher.utter_message(text=output)
        return [SlotSet("current_search", name), SlotSet('game_id', game_id)]

```

Figura 4.5: Action di ricerca di trailer di un videogame

## 4.6 Get Metacritic

L'action, dopo aver ottenuto il game id, effettua una richiesta al dettaglio presso la route `/games`, sia per estrarre, se disponibile, il metacritic score (il metacritic score è un punteggio da 0 a 100 che mostra la media, ottenuta dal videogame, delle recensioni delle testate giornalistiche), sia per estrarre l'ESRB rating (l'ESRB classifica i videogiochi in base al loro contenuto, attraverso fasce d'età e commenti sul tipo di contenuto). Anche in questo caso l'action gestisce le varie possibili interazioni con l'utente in maniera simile a come fatto finora.

```

class ActionMetacritic(Action):

    def name(self) -> Text:
        return "action_metacritic"
    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('game')
        indice = tracker.get_slot('index')
        if indice > 0:
            indice=indice -1
        game_id=tracker.get_slot('game_id')
        current=tracker.get_slot('current_search')
        print(name)
        print(game_id)
        if game_id==0:
            if name!=current:
                r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                if r.status_code ==200:
                    data = r.json()
                    game_id=data['results'][indice]['id']
                else: output = "I couldnt find any info of the game"
            else:
                if name!<current:
                    r=requests.get(url='https://api.rawg.io/api/games?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
                    if r.status_code ==200:
                        data = r.json()
                        game_id=data['results'][indice]['id']
                    else: output = "I couldnt find any info of the game"
                r = requests.get(url="https://api.rawg.io/api/games/{}?key=bbac0252b5ed4a2b8286472063cb2dfe".format(game_id))
                data = r.json()
                nome=data['name']
                metacritic= data['metacritic']
                if metacritic is None:
                    metacritic="Not available"
                ratings = []
                if data['esrb_rating'] is None:
                    ratings_data='Not Available'
                else:
                    ratings_data = data['esrb_rating']['name']
                print(ratings_data)
                output="{} has a metacritic score of {} and has been tagged with: {}".format(nome,metacritic, ratings_data)
                dispatcher.utter_message(text=output)
        return [SlotSet("current_search", name), SlotSet('game_id', game_id)]

```

Figura 4.6: Action di ricerca relativo allo score di metacritic

## 4.7 Search Publisher

L'action, una volta estratto il publisher dallo slot, esegue una query all'API presso la route `/publishers`, estraendo dal risultato alcune informazioni come il nome completo del publisher, l'id, il numero di giochi pubblicati ed un'eventuale lista di top games dell'azienda. Con queste informazioni, l'action esegue due nuove query: la prima, utilizzando l'id del publisher, ottiene una descrizione aggiuntiva da mostrare all'utente, mentre la seconda utilizza l'id del primo elemento dell'array top games per estrarre il nome del videogioco più rappresentativo dell'azienda. Infine, il chatbot mostra all'utente una descrizione del publisher con il numero di giochi pubblicati e se presente il nome di uno dei top games.

```

class ActionSearchPublisher(Action):

    def name(self) -> Text:
        return "action_search_publisher"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        name = tracker.get_slot('publisher')
        print(name)
        r=requests.get(url='https://api.rawg.io/api/publishers?key=bbac0252b5ed4a2b8286472063cb2dfe&search={}&search_precise=true'.format(name))
        if r.status_code == 200:
            data = r.json()
            nome = data['results'][0]['name']
            print(nome)
            if 'None' in nome:
                output = "I do not know anything about , what a mistery!? Are you sure it is correctly spelled? Try to use the complete name of the publisher"
                dispatcher.utter_message(text=output)
                return [AllSlotsReset()]
            description=""
            games_count = data['results'][0]['games_count']
            top_game = data['results'][0]['top_games'][0]
            publisher_id=data['results'][0]['id']
            print(publisher_id)
            r2=requests.get(url='https://api.rawg.io/api/publishers/{}?key=bbac0252b5ed4a2b8286472063cb2dfe'.format(publisher_id))
            if r2.status_code == 200:
                data2 = r2.json()
                print(data2)
                description = cleanhtml(data2['description'])
                print(description)
                r3= requests.get(url='https://api.rawg.io/api/games/{}?key=bbac0252b5ed4a2b8286472063cb2dfe'.format(top_game))
                if r3.status_code == 200:
                    data3 = r3.json()
                    game = data3['name']
                    output="{} {} realeased {} games and one of the most known game of this company is: {}".format(description,nome, games_count, game)
                else:
                    output = "{} {} realeased {} games.".format(description,nome,games_count)
            else:
                output = "I do not know anything about , what a mistery!? Are you sure it is correctly spelled?"
            dispatcher.utter_message(text=output)
            return [SlotSet('publisher',None)]

```

Figura 4.7: Action di ricerca del publisher

## 4.8 Reset Index

L'action *reset index* viene chiamata solo nel caso in cui l'utente, una volta ottenuti i risultati di una ricerca, confermi la correttezza della stessa.

```

class ActionResetIndex(Action):

    def name(self):
        return "action_reset_index"

    def run(self, dispatcher, tracker, domain):
        return [SlotSet("index", 0)]

```

Figura 4.8: Action ResetIndex

## 4.9 Reset Slot

L'action resetta tutti gli slot ai valori di default. Essa è eseguita nel caso in cui l'utente abbia espresso la volontà di aver terminato la ricerca di informazioni su un particolare videogame. In questo modo, le richieste successive riandranno a salvare nuovi valori negli slot ed eseguire correttamente nuove richieste di informazioni su altri videogame da parte dell'utente.

```
class ResetSlot(Action):

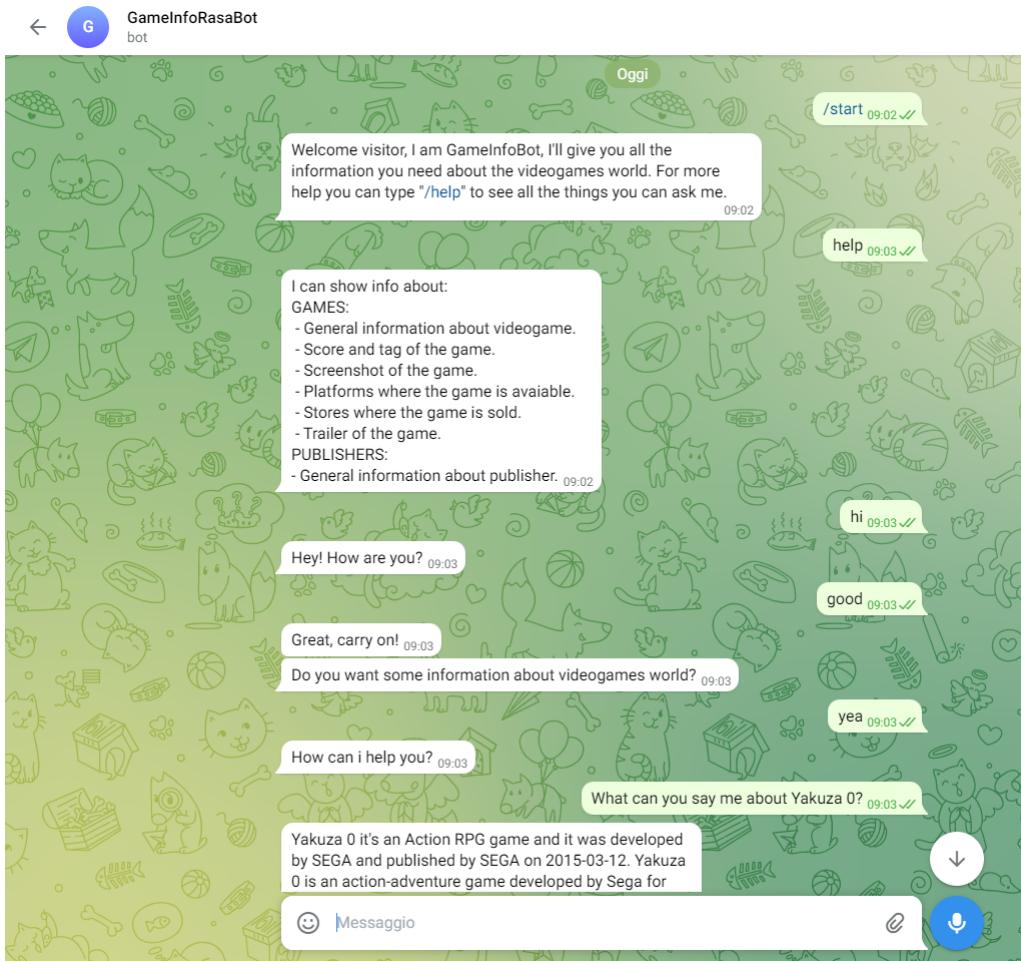
    def name(self):
        return "action_reset_slots"

    def run(self, dispatcher, tracker, domain):
        print("Reset degli slot")
        return [AllSlotsReset()]
```

Figura 4.9: Action ResetSlot

# Capitolo 5

## Esempi di utilizzo





GameInfoRasaBot  
bot

Yakuza 0 it's an Action RPG game and it was developed by SEGA and published by SEGA on 2015-03-12. Yakuza 0 is an action-adventure game developed by Sega for Microsoft and PlayStation in 2015.

#### Plot

The game takes place in 1988 in Tokyo. The main characters are Kazuma Kiryu and Goro Majima, who are forced to participate in the power struggle of the Japan mafia organizations. Kiryu is framed for a murder and pursued by a huge mafia family. On the loose, the protagonist tries to prove innocent and reveal the truth behind the so-called Empty Lot conflict. The aim is to follow the plot and make choices based on different strategies.

#### Gameplay

The game is played from a third-person perspective. The player can explore the world freely. The action is performed in Kamurocho, which is the representation of Kabukichō, Tokyo's district. During the game, the player resembles the usual mechanics: walking, listening to dialogues and watching cinematics pieces. The action includes side-quests which let the protagonists earn more money and special Completion Point currency; each CP lets the player gain additional bonuses, upgrades, and certain items. The main gameplay modification is battle control. It lets the player switch action styles in the middle of the fight.

#### Reception

Yakuza 0 is a prequel of Yakuza, video game series first released in 2005. The original was acknowledged for its innovative gameplay. The prequel followed the success and became favorable as well.

<https://media.rawg.io/media/games/ca1/ca16da30f86d8f4d36261de45fb35430.jpg>.

Is this the game you wanted?



Messaggio



← G GameInfoRasaBot  
bot

Yakuza 0 is a prequel of Yakuza, video game series first released in 2005. The original was acknowledged for its innovative gameplay. The prequel followed the success and became favorable as well.  
<https://media.rawg.io/media/games/ca1/ca16da30f86d8f4d36261de45fb35430.jpg>.

Is this the game you wanted?



09:03

I'm glad that I help you. I can show you other info (type help to see how can i help you).

09:04

yes 09:04 ✓

show me some screens 09:04 ✓

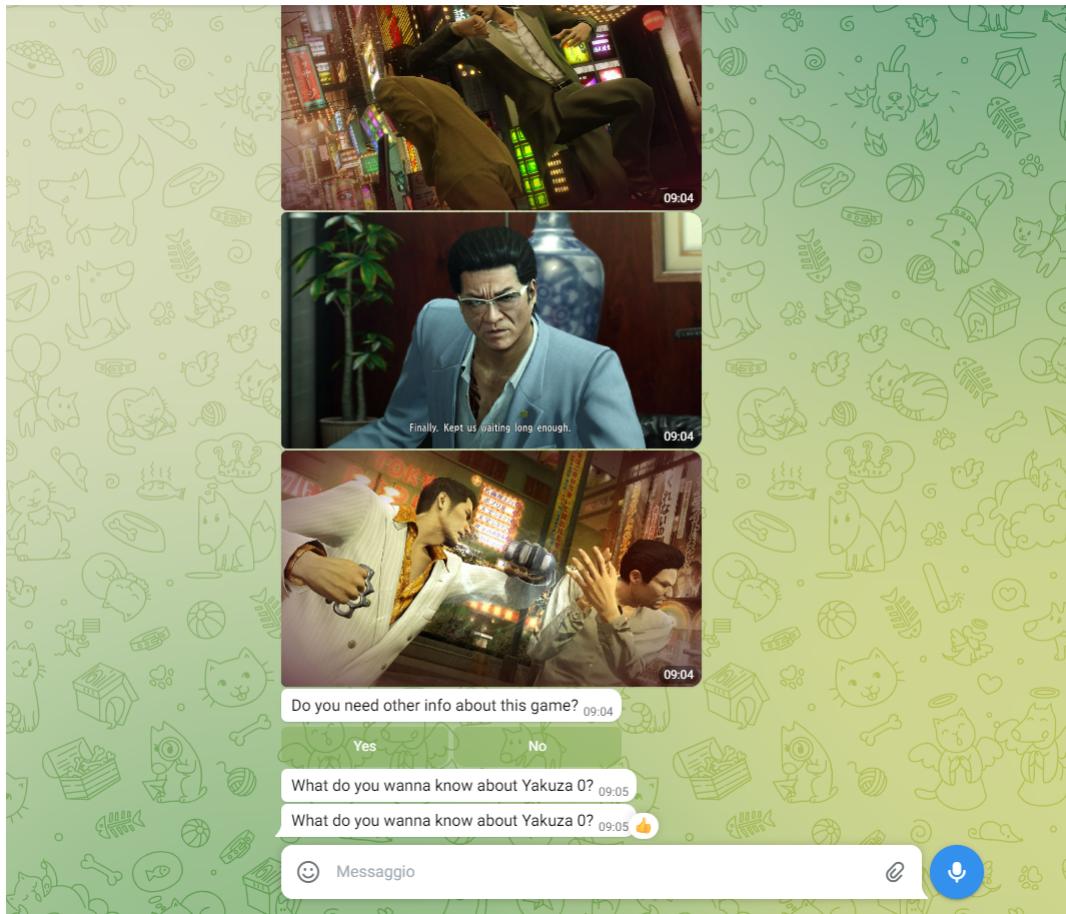


09:04





GamelInfoRasaBot  
bot





GameInfoRasaBot  
bot

Finally. Kept us waiting long enough. 09:04

Do you need other info about this game? 09:04

Yes No

What do you wanna know about Yakuza 0? 09:05

What do you wanna know about Yakuza 0? 09:05

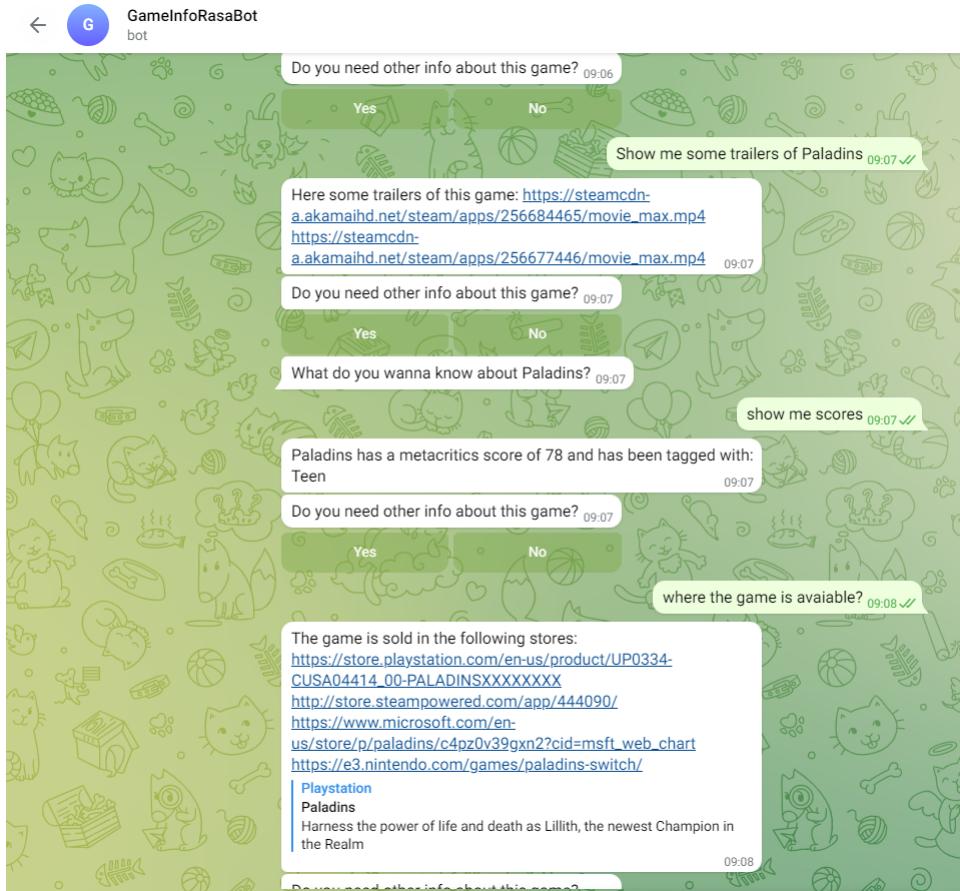
Where can i buy the game? 09:06

The game is sold in the following stores:  
[https://www.microsoft.com/en-us/p/yakuza-0/9npp17lhj3mk?cid=msft\\_web\\_chart](https://www.microsoft.com/en-us/p/yakuza-0/9npp17lhj3mk?cid=msft_web_chart)  
[https://store.playstation.com/en-us/product/UP0177-CUSA05070\\_00-YAKUZA0AMERICA00](https://store.playstation.com/en-us/product/UP0177-CUSA05070_00-YAKUZA0AMERICA00)  
<https://store.steampowered.com/app/638970/>

09:06

Do you need other info about this game? 09:06

Yes No



 GamelInfoRasaBot  
bot

Do you need other info about this game? 09:08

Yes No

Paladins has been published on the following platforms: PC Xbox One PlayStation 4 Nintendo Switch macOS 09:08

platforms 09:08 ✓

Do you need other info about this game? 09:08

Yes No

Welcome visitor, I am GamelInfoBot, I'll give you all the information you need about the videogames world. For more help you can type "/help" to see all the things you can ask me. 09:09

How many games Square Enix published? 09:09 ✓

Square Enix is a Japanese video game publisher.  
History  
Square Enix was formed on April 1st, 2003 with the merging of Square Corporation and Enix. Since its creation, Square Enix purchased several more companies, that mainly were arcade machine developers, such as Taito Corporation and Tecmo.  
Operations  
Square Enix boasts a number of its own IPs. The company acts mainly as a video game developer, rather than a publisher. However, through multiple friendly company takeovers, Square Enix became one of the biggest franchise holders.  
Square Enix main titles are Final Fantasy, Kingdom Hearts, Tomb Raider, Deus Ex, Hitman as well as Japanese-only game series. There is no particular focus or interest in just one genre. The company's policy is to create new titles in any direction possible, which makes Square Enix the biggest game-related company in Japan and one of the biggest publishers in the world. Square Enix released 469 games and one of the most known game of this company is: Tomb Raider (2013). 09:09

 Messaggio  