YuGiOh-Utilities

Andrea Pepe   
*Faculty of Computer Engineering, University of Rome "Tor Vergata"*  
Rome, Italy  
pepe.andmj@gmail.comMatteo Ciccaglione  
*Faculty of Computer Engineering, University of Rome "Tor Vergata"*  
Rome, Italy  
matteociccaglione@gmail.com

*Abstract*— This paper aims to give an overview of “YuGiOh-Utilities”, a marketing utility distributed system for the KONAMI trading card game Yu-Gi-Oh. The system allows a user to track the trend of market prices for a card of interest, monitored through one of the seven available different price sources. The user can also sell or buy cards by interacting with other users in the system, taking advantage of the matching function, which makes it possible to meet and deal with other users interested in the same card in the same price range. The user can complete the purchase directly through our system, using the PayPal payment channel, or withdraw the credit accumulated thanks to his sales to his PayPal account.

# Backend Architecture

The backend is designed and implemented following a microservices architecture, including as many as seven microservices. On the client side, an application for Android devices has been created.

Going into more details on the backend architecture, the microservices created are the following:

* User Management: microservice used to manage all user information, such as personal data, access credentials, card currently in tracking, sale/purchase transactions and available credit. This microservice is also responsible for authenticating a user to the system.
* Database: microservice used for storing and managing all cards currently released for the TCG Yu-Gi-Oh, as well as minimum price updates provided by Cardmarket [1], TcgCardGame [2], Amazon [3] and Ebay [4].
* Tracking: microservice used for automatic tracking of price trends at the minimum and maximum market prices using the YuGiOh Prices REST service [5].
* Chat: microservice used to manage a chat bot, which allows users to conclude negotiations for the purchase/sale of a card and provide their shipping address.
* Paypal: microservice responsible for interacting with the PayPal payment channel.
* Matching: microservice used to match users interested in the purchase of a specific card with users interested in the selling of that card with similar prices.
* EmailSender: microservice responsible for sending messages, via e-mail, to the users.

# Microservices Details

A graphic representation of the system architecture is available at the link <https://github.com/AndreaPepe/yugioh-utilities-docs/blob/main/img/YuGiOh-Utilities%20Microservices%20Architecture.png>.

For internal communication among microservices, the gRPC middleware has been chosen, because it provides more flexibility in terms of programming language choices for the microservices’ implementations. Some of them also communicate using message queues or publish/subscribe systems.

## User Management

The microservice is responsible for user registration, login and account management. It relies on a MySQL DBMS, where the data regarding the user accounts are stored. In particular, beyond the personal information, the microservice keeps track of the current in-tracking card for each user. Currently, the system does not support more than one card in tracking per user and no more than one price source. It also interacts with a Kafka cluster to receive price updates for the tracked cards (each card represents a topic), published by the tracking microservice. Moreover, the service is responsible of the management of the sell/purchase transactions and of the credit accumulated by the users, keeping, for each user, an history of his transactions.

User management is implemented separating the transactions logic from the personal account information logic: in particular, it provides two different gRPC services, exposing specific functions to other microservices of the system.

The service is implemented in Python language and performs a perioding task, polling Kafka about topics in which users are interested. When it receives prices updates, it compares them with a price threshold selected by the user; if the threshold is exceeded, the service generates some email messages that will be sent to the users, in order to notify them; these messages are put in an AWS SQS queue and will then be delivered by the EmailSender microservice.

Payment functionalities are implemented using local transactions, which exploit DBMS stored procedures, involved in a global SAGA transaction.

## Database

This microservice has the role of keeping a database of all the existing YuGiOh TCG cards. It not only stores them, but periodically checks if newer cards have been released or some of them has been updated, in order to always have a freshly updated database. In particular, every two hours, a check for newer versions of the database is done and, in case of positive response, the cards information will be downloaded using the REST API of the external service YGOPRODeck [6], stored in local database and then propagated to the Tracking microservice. The microservice is developed in Go language and exposes gRPC functions that allow the search of card in the database, giving also links to download card image. Because the database is really simple (only one table), SQLite3 has been used, in order to avoid DBMS overhead. The SQLite file is embedded inside the microservice container, which makes it stateful, although it is actually not replicated.

## Tracking

The Tracking microservice is designed with a master-worker architecture: there is a loadbalancer component which receives updates from the Database microservice (consisting in cards information, like the IDs) and divides the data equally for the workers, assigning them a pool of cards of which they have to track the prices. Communication between the loadbalancer and the workers happens through gRPC. Since Docker Swarm orchestration automatically have a load-balancing mechanism for replicated services, the loadbalancer component was not strictly needed, but it has been designed and implemented to increase decoupling among microservices and have more control on the cards to be tracked, filtering out the malformed ones.

Both loadbalancer and workers are written in Go and are stateless microservices, since they do not store any data. The tracking is done using REST API of the external service YuGiOhPrices [7], and publishing them through Kafka. Workers interface with Kafka using the “Sarama” library [8].

Moreover, when the loadbalancer sends a pool of cards to a worker, it checks for communication error and, if something goes wrong, it retries to send data for at most ten times. Workers, even if there are no updates from the Database service, periodically perform tracking operation every 30 minutes; if there are errors during the execution of this task, they retry to do it at most three times. In this way, eventual consistency is achieved about prices updates.

## Chat

Chat microservice is the component of the system that allow negotiations between users interested in the purchase or selling of a card. The communication is not direct between two users, but it’s done with an intermediary bot (the Chat microservice). It must be assumed that two users have an active chat only if they have been matched by the Matching microservice; therefore, in this scenario, there will be a seller user and a buyer user. Message has been implemented using the AWS SQS service: the chat microservice has its own queue, used to enqueue messages it has to deliver to different users; for each user that has at least one chat, there will be a dedicated SQS queue, where the Chat microservice will deliver the messages intended for him.

The chat interaction always starts with a message for the buyer, informing the user that a matching for the purchase of the card he was interested in has been found at a given price. Now, the buyer can choose to accept or refuse the offer. If he accepts, then the seller will receive a similar message and he will accept or refuse the transaction. If both users accept, the chat microservice will send another message to the buyer, asking him to specify the desired shipping address. Once the microservice will receive the response, the specified address will be communicated to the seller and a transaction will be created by invoking, through gRPC, a specific service of the User Management microservice, allowing the buyer to proceed with the payment.

The Chat microservice is written in Python and saves metadata about chats in a MySQL DBMS.

## Paypal

The Paypal microservice plays the role of system component responsible for payment by the users, through the PayPal [9] external service. Indeed, each user, involved as a buyer in a transaction, can use a PayPal account to make the payment. The amount of money will not directly be delivered to the seller account, but the buyer user will actually send money to the YuGiOh-Utilities system, which owns a PayPal business account. The credit due to the seller will be kept in the system until he explicitly requests to withdraw it; if so, the system will generate a PayPal payout for the user, on the PayPal account associated with the email indicated by him at request time.

It is important to note that all taxes imposed by PayPal, such as “goods and services”, are charged to users. Furthermore, for each credit withdrawal from the system, an amount equal to 2% of the requested credit (already taxed by PayPal) will be kept by the system as earnings.

Of course, each withdrawal request will involve several operations: first of all, a user can not withdraw more money than he has on his credit. If there is enough credit, then the requested amount should be subtracted from the user account. All these operations are performed by the User Management microservice, triggered by the Paypal microservice using gRPC. This structure avoids having a high coupling between microservices and increases their coherence and cohesion. This interaction between the two microservices is handled using the SAGA pattern.

Paypal microservice also relies on a MySQL DBMS, in order to keep track of the users’ payments. Going into more details, a payment of a buyer user is managed as a two-step process: first, when the user makes a request to pay, a Paypal “order” will be created and appropriate metadata, such as the transaction identifier for which he is paying, will be stored in the database. From the creation of the order, the user has 48 hours to complete the payment, triggering the execution of a PayPal “capture” of the order. The system will receive a notification of the payment by redirecting the user to an appropriate endpoint exposed by the system itself, and, in case of success, will increase the credit of the seller account.

Clearly, the interaction with the PayPal external service can result into failures, but, about payment captures, it is robust to replicated requests, so a user will never pay for the same transaction more than once. About withdrawal, the consistency is obtained using SAGA.

The microservice has been developed in Python language, mostly because the PayPal Developer REST API documentation provides some goods examples in such programming language and also to reuse the code structure used in other microservices for the interaction with the DBMS. Actually, the payment using PayPal is only possible in a developer sandbox, using several fake accounts created on purpose.

## Matching

The Matching microservice is the component that registers the user’s intention to sell or buy cards. It not also does that, but, every time a user registers as interested in buying or selling a card, the microservice will try to match the requesting user with other users. For example, if a user shows his interest in buying a card A at the price of X euros, the Matching microservice will try to find a user who, instead, sells card A at about the same price X (in particular, the system will search for sellers of the card A at a price between X – 3.50€ and X + 3.50€). If such a user is found, a matching is created and the Chat microservice will be asked to create a chat between the two matched users, so they can proceed with negotiations. Moreover, it will use AWS SQS to place an email message, informing of the matching, for each user, in the EmailSender queue.

When two users are matched, their visibility will be set to “invisible”, so they can not be matched with other users if they already have a matching. In case of unsuccessful negotiations, both users will become again “visible” and the system will try to rematch them with different users. Clearly, if instead the negotiations are successful, the involved users will be removed from the database.

When requesting to sell or buy a card, a user can decide to not specify any price: in this case, the price will be the minimum market price, retrieved from Kafka, using Cardmarket as price source.

The microservice stores data using SQLite3 in a file embedded in the container hosting the single instance of the microservice. This makes the microservice a stateful one. So, in order to have more than one replica of it, a consistency protocol has been implemented, making use of the Kafka system. Each instance, at start-up, generates a UUID; each write operation on the local database file, is published on Kafka and marked with the UUID. Each instance will receive messages from Kafka, and will redo the same operation indicated in the message, except for the ones marked with its UUID. With such a system, assuming no crashes, eventual consistency is achieved.

This microservice has also been developed in Python language, in order to reuse some previously available code for the management of data with SQLite3. Such technology has been chosen because data to be saved have a fairly simple structure and a DBMS is not strictly required for performing necessary operations.

## EmailSender

The EmailSender microservice is only responsible for sending e-mail messages to users. It periodically fetches messages from a dedicated AWS SQS queue and then process and sends them to users through Outlook SMTP server, using the javax [10] library. The microservice is completely stateless and is written in Java, exploiting reusability of code already developed in previous context. For dependency management, the Maven framework has been adopted.

# API Gateway

The system exposes its functionalities through a REST API Gateway server. This component represents the access point to the system for the client and is implemented in NodeJS, using the “Express” [11] library. First of all, the API Gateway performs the authentication of the users, generating and storing session cookies in a Redis database. When a user logs successfully into the system, he will receive the cookie to use in following requests. Each cookie will expire after a 24 hours period and the user will need to log in again. Almost every invokable API requires a valid cookie, otherwise the requested operation will not be performed. Internally, each API invokes, using gRPC, functions exposed by the previously described microservices. Particularly, the API Gateway directly communicates with User Management, Paypal, Matching, Chat and Database microservices.

It also offers the possibility to generate a new password, in case the user forgot it: the API Gateway will generate an OTP valid for at most 1 hour, that will be sent to the client through the EmailSender microservice. Once the user will submit the OTP, a new password will be pseudo-randomly generated and also communicated to the user via e-mail.

As previously mentioned, the API Gateway also exposes an endpoint that will receive requests when a user captures a PayPal order.

Relying on a separated and shareable Redis database, the API Gateway is easily replicable, increasing system scalability and availability.

# Microservices Patterns

Two different patterns for microservice architecture have been used in the development of the system.

## Database per service

The database per service pattern is used for stateless microservices relying on a MySQL DBMS: these microservices are User Management, Paypal and Chat. Each of them uses a dedicated schema on the same shared DBMS, avoiding direct access to other services’ data. So, the schema-per-service variant of the pattern has been adopted, which have a sufficiently low overhead with respect to the database-server-per-service variant. The application of this pattern allows to be more robust in accessing data, ensuring that each service accesses correctly only data for which it has permissions. It also increases maintainability and decoupling of the different microservices.

## Saga pattern

The saga pattern has been used for managing the payment use-cases involving the Paypal and the User Management microservices. In particular, when a user makes a request to retrieve some of the credit he has on his account, the Paypal microservice will be invoked. In chain, it will trigger a local transaction on the User Management service, which will check if the user has enough credit; if so, the amount of money requested will be subtracted from the total available credit and the control will return to the Paypal service, which, in turn, will perform the real payment operation to the user PayPal account. If this last operation succeeds, the whole saga is completed successfully; otherwise, the Paypal microservice will make the User Management service rollback the previous operation, bringing back the user’s credit to the initial amount.

A similar interaction happens when a buyer completes a payment, in order to increase the seller’s credit amount.

The usage of this pattern was strictly necessary in the described scenario, to avoid inconsistent behaviours, such as a decreasing of the credit of a user without the execution of an effective payment and vice versa.

It should be noticed that the operation interacting with the external PayPal service is always the last step of the saga. This choice has been done in order to not have to invoke rollbacks on PayPal: if the operation succeeds, the saga ends and there will never be the need to rollback it; if the operation fails, the only operations to be rollbacked will be the previous ones, on which the system has full control. In this way the system does not depend on the rollback operations exposed by PayPal, potentially subject to changes.

# Client

The realized client consists in an Android application, written in Kotlin language, using the new Jetpack Compose toolkit for building native UI, developed by Google and adopted as new standard for Android development. A mobile client app was preferred, as it better suited the needs of the typical YuGiOh player, who constantly wants to stay updated on card prices and always have quick access to the system.

The application allows the user to interact with the backend system and exploit the offered features, using REST API exposed by the API Gateway. In addition to the backend functionalities, the app permits to store a local collection of cards, helping the user to manage his card set. The user can arbitrarily add or delete cards from the collection, in order to keep track of the cards he owns. The added cards will be shown with their image, so as to be easily recognizable.

The app stores necessary information in a local database, using the Room library [12] and SQLite; more specifically, a new database for each logged user will be created. One of the key information stored is the cookie received from the API Gateway following a successful login request; such cookie will then be used in the following requests, in order to be always authenticated.

For the chat use-case, as previously described, each user has a dedicated AWS SQS queue. In order to send answers to the bot, the user puts messages in the bot SQS queue. Although, this operation requires AWS credentials, that the application obtains from the API Gateway after a successful login.

# Deployment

The system backend as been deployed containerizing each microservices instance in a Docker container. Moreover, there is a container to host Kafka, another which contains Zookeeper, used by Kafka itself, and other two containers for the shared database services: Redis and MySQL DBMS.

All the containers are deployed on several AWS EC2 virtual machine instances. Empirically, it has been seen that four EC2 instances of type “t3.large” are sufficient and their load is quite balanced, with 12 tracking worker instances and only two replicas of the matching microservice.

The containers are orchestrated using the Docker Swarm [11] framework: one of the four EC2 instances is selected as the swarm manager, while other instances join the swarm as workers. The deployment of the containers over the nodes of the swarm is automatically managed by the Docker Swarm framework, which tries to balance the workload of each node. Containers can communicate each other using an overlay network built on the nodes of the swarm. Moreover, volumes have been used to permits containers to access the AWS credentials needed to do their job and to have persistency of data, in particular of data written through the MySQL DBMS.

One of the advantages of the Docker Swarm framework is that it automatically performs disaster recovery, restarting instances that, for some reason, crashed. In this way, the number of desired replicas specified always match the real number of available instances for each service.

On the other hand, with respect to Kuberneetes [12], automatic scaling of services is not performed.

Furthermore, a dedicated DNS name has been assigned to the API Gateway, so it is accessible from public internet. SSL certificates have also been generated and used to encrypt the communication between clients and the system access point, using the HTTPS protocol. This significantly improves the global security of the system.

## Some Common Mistakes

* The word “data” is plural, not singular.
* The subscript for the permeability of vacuum **0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter “o”.
* In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
* A graph within a graph is an “inset”, not an “insert”. The word alternatively is preferred to the word “alternately” (unless you really mean something that alternates).
* Do not use the word “essentially” to mean “approximately” or “effectively”.
* In your paper title, if the words “that uses” can accurately replace the word “using”, capitalize the “u”; if not, keep using lower-cased.
* Be aware of the different meanings of the homophones “affect” and “effect”, “complement” and “compliment”, “discreet” and “discrete”, “principal” and “principle”.
* Do not confuse “imply” and “infer”.
* The prefix “non” is not a word; it should be joined to the word it modifies, usually without a hyphen.
* There is no period after the “et” in the Latin abbreviation “et al.”.
* The abbreviation “i.e.” means “that is”, and the abbreviation “e.g.” means “for example”.

An excellent style manual for science writers is [7].

# Using the Template

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper; use the scroll down window on the left of the MS Word Formatting toolbar.

## Authors and Affiliations

**The template is designed for, but not limited to, six authors.** A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

### For papers with more than six authors: Add author names horizontally, moving to a third row if needed for more than 8 authors.

### For papers with less than six authors: To change the default, adjust the template as follows.

#### Selection: Highlight all author and affiliation lines.

#### Change number of columns: Select the Columns icon from the MS Word Standard toolbar and then select the correct number of columns from the selection palette.

#### Deletion: Delete the author and affiliation lines for the extra authors.

## Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is “Heading 5”. Use “figure caption” for your Figure captions, and “table head” for your table title. Run-in heads, such as “Abstract”, will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced. Styles named “Heading 1”, “Heading 2”, “Heading 3”, and “Heading 4” are prescribed.

## Figures and Tables

#### Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation “Fig. 1”, even at the beginning of a sentence.

1. Table Type Styles

| Table Head | Table Column Head | | |
| --- | --- | --- | --- |
| Table column subhead | Subhead | Subhead |
| copy | More table copya |  |  |

1. Sample of a Table footnote. (*Table footnote*)
2. Example of a figure caption. (*figure caption*)

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity “Magnetization”, or “Magnetization, M”, not just “M”. If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write “Magnetization (A/m)” or “Magnetization {A[m(1)]}”, not just “A/m”. Do not label axes with a ratio of quantities and units. For example, write “Temperature (K)”, not “Temperature/K”.

##### Acknowledgment *(Heading 5)*

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks ...”. Instead, try “R. B. G. thanks...”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

##### References

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first ...”

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors’ names; do not use “et al.”. Papers that have not been published, even if they have been submitted for publication, should be cited as “unpublished” [4]. Papers that have been accepted for publication should be cited as “in press” [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

1. G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*
2. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
3. I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
4. K. Elissa, “Title of paper if known,” unpublished.
5. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
6. Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
7. M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

**IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove template text from your paper may result in your paper not being published.**