University of Trento
Department of Computer Science
Service Design and Engineering

Report — Wishlist Inviter

Marcel Pascal Stolin
`marcelpascal.stolin@studenti.unitn.it`

28th August 2022

1 Introduction

The goal of this project is to present the learning of the course *Service Design and Engineering*. The requirements are to develop a project that consists of multiple microservices and use techniques and technologies presented during the course. The idea of the project is an application where the user can create wishlists, and invites other persons to buy an item from that wishlist via email. The content of a wishlist is based on a vendor wishlist (e.g. Amazon.com). To make a wishlist available, the application extracts the content from a given wishlist URL and saves it in a structured format to the database.

2 Design

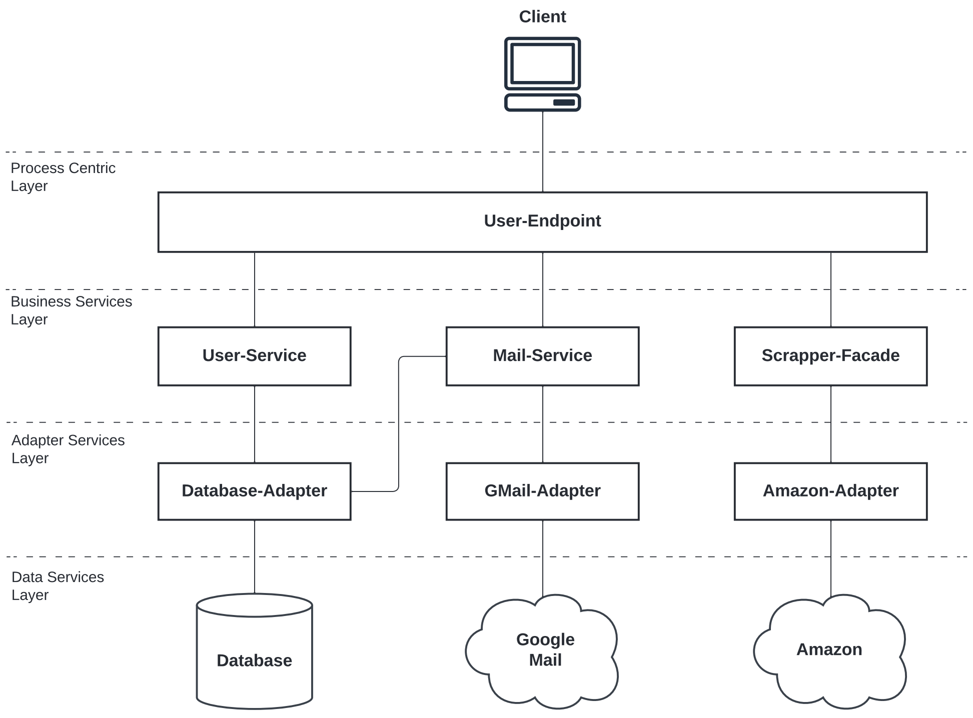

Figure 1: Micro service architecture

Figure 1 illustrates the architecture of the application. It consists of four different layers, each serving a different purpose. Each layer (except the Data Layer) includes services that are designed to solve a specific task and be self-describing and self-continued. The architecture is designed as a choreography. No service takes responsibility for the whole process and services are aware of collaborating.

2.1 Process Centric Layer

The Process Centric Layer consists only of the *User-Endpoint*, which is responsible to handle all client-specific requests. Therefore, it serves as a gateway to all other services and forwards/sends requests to all services of the underlying Business Services Layer.

2.2 Business Services Layer

The Business Services Layer consists of three services, the *User-Service*, *Mail-Service*, and *Scraper-Facade*. All three services coordinate business processes (e.g. sending a mail) and overall determine what the application is capable of. None of these services have direct access to modify data.

User-Service: The *User-Service* is responsible to handle all user-related tasks. This includes: Creating a new user, deleting an existing user, adding items to a user's wishlist, updating/deleting items from a user's wishlist, and requesting information about a user and/or its items. It is able to send requests to the *Database-Adapter*.

Mail-Service The *Mail-Service* is responsible to handle all email-related tasks. At the current version, this only includes sending invitations. It receives the needed data to generate an e-mail object and forwards it to the *GMail-Adapter*.

Scraper-Facade The *Scraper-Facade* handles all requests for extracting data from a vendor. It hides the complexity of composing multiple vendor-specific adapter services. Therefore, it forwards the request to the specific vendor adapter service and responds with its result. In the current version, only Amazon wishlists are supported. In the future, this service should include other vendors (e.g. Walmart, Media World).

2.3 Adapter Services Layer

The Adapter Services Layer includes all services that utilize physical data storage or provides an interface to an existing external service.

Database-Adapter: The *Database-Adapter* has direct access to the Database and performs CRUD operations. It abstracts its usage by providing a REST API.

GMail-Adapter: The *GMail-Adapter* abstracts the usage of working with the Google-Mail SMTP server directly.

Amazon-Adapter: The *Amazon-Adapter* is a web scraper that can extract data from Amazon wishlists. It makes the extracted data available as JSON objects through a REST API.

2.4 Data Layer

The Data Layer includes all physical data stores and external services, including a Postgres Database, an Amazon Clone, and the Google-Mail SMTP server. The database of choice is a Postgres DB. It serves as the persistent storage for this project. Mails are sent via the Google-Mail SMTP server[1]. To simplify the implementation of this project, two Amazon wishlists have been exported from Amazon.com. They are being served via an NGINX server and try to be as identical to the original Amazon website as possible. The reason is explained in Section 3.5.

3 Implementation

This section explains important details about the project implementation, the documentation, as well as the project structure.

3.1 Development

Most services of this project are developed using the Go programming language, except the *Amazon-Adapter* that is implemented in Python because of its ecosystem advantages. To build web services using Go, the chi[2] router was chosen because of its idiomatic and lightweight nature. Additionally, GORM[3] is used as an ORM to work with the database. The *Amazon-Adapter* is implemented using FastAPI[4] as the web

[1]`https://developers.google.com/gmail/imap/imap-smtp`
[2]chi — `https://go-chi.io/`
[3]GORM — `https://gorm.io/`
[4]FastAPI — `https://fastapi.tiangolo.com/`

framework, and Beautiful Soup 4[5] to extract data from HTML documents. All services provide a REST API for communication. Furthermore, to provide consistency and isolation each service runs in its own container using Podman[6]. There is no graphical user interface provided for this API. However, a Postman[7] collection is provided to test and use the API of each service.

3.2 Security

To secure access to resources, all services use JSON Web Tokens (JWT). To authenticate, a client has to send a `POST` request to the `users/auth` endpoint of the *User-Service*. This request requires a JSON body containing the ID of the user who wants to authenticate. Then it returns an access token that is meant to be used to authenticate for all other endpoints. The given access token is valid for 24 hours. All services use the same secret. Therefore, tokens get forwarded between services.

3.3 Database

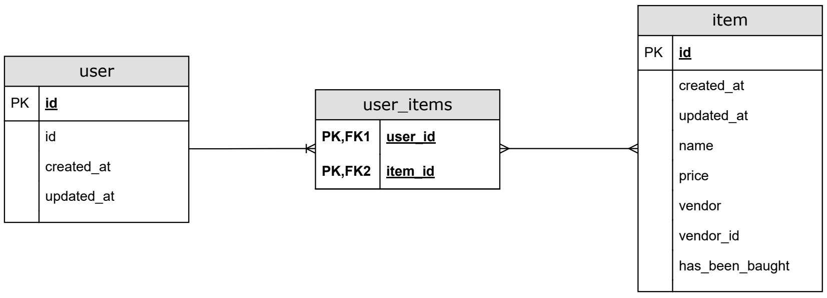

Figure 2: Database entity-relationship-model

Figure 2 illustrates the entity-relation-model diagram of the database. There exist two models, a user, and an item. A single user owns multiple items. To simplify the database model, an item can only be owned by a single user. A user only exists of an ID that identifies a user and is unique for each one. The *User-Service* creates new user instances and randomly creates a `UUIDv4` for a new user. An item is identified by its ID. Furthermore, the item model also saves the vendor ID which refers to the ID that was given by the vendor. Important to mention is also the `has_been_bought` property, the reason why an item can only be owned by a single user. It represents if an item has been bought by a buyer or not.

3.4 Project Structure

The project repository is available at `https://github.com/mstolin/present-roulette`. It is a single multi-module repository that contains all source files and the documentation of this project. To manage a multi-module Go application, the workspace feature is used. A `README.md` file is provided in the root directory, that explains additional documentation on how to set up the application. Furthermore, each service provides documentation explaining its purpose and development details. Additionally, the `docs/` folder provides the documentation for the REST API of each service, using the apiary[8] format.

3.5 Problems

To simplify the extracting of Amazon wishlists, the *Amazon-Clone* has been created. The reason is, that Amazon does not allow extracting its data and tries to prevent this through bot detection algorithms (artificial intelligence, IP banning, captcha, etc.). There exist web spider frameworks[9] that try to bypass these detection algorithms, however, it would have increased the complexity of the *Amazon-Adapter* significantly. Furthermore, these web spider frameworks foster a asynchronous communication and are more effective in combination with asynchronous communication constructs like publish/subscribe or message queues instead of REST.

[5]Beautiful Soup 4 — `https://beautiful-soup-4.readthedocs.io/`
[6]Podman — `https://podman.io/`
[7]Postman — `https://www.postman.com/`
[8]apiary — `https://apiary.io/`
[9]Scrapy — `https://scrapy.org/`