CS302 Final Project Report

Tony Hoare

Bryan Lee Min Yuan, Christopher Lim Sheng Yong, Ko Hui Ning,
Justina Ann Wong, Pang Huan Shan Shawn

Singapore Management University

{bryan.lee.2019, sylim.2019, huining.ko.2019,
justinawong.2019, shawn.pang.2019}@scis.smu.edu.sg

Abstract – People love digital collectibles and kopi (Singapore slang for local coffee). We created Kopi Time to satisfy both interests – a marketplace for people to buy, sell and collect coffee collectibles. In addition, this paper introduces the novel use of a framework-independent Saga Engine to handle microservice coordination and compensations.

I. THE PROBLEM

In recent years, there has been an explosive rise in popularity of digital collectibles and coffee. However, there is no solution that simultaneously satisfies both trends – there is no platform for kopi collectibles.

II. THE SOLUTION

The team is proud to present Kopi Time – a bustling digital marketplace for vibrant kopi collectibles, where users can buy, sell, and collect one-of-a-kind coffee art[1][2][3].

Figure 1. Sample of Kopi Collectibles

III. KEY SCENARIOS

A. *Seller: Auction*

Sellers can put their own collectibles up for auction. To do this, a series of calls are made, namely:

(1) get seller and item information from user and inventory services;
(2) create a listing on listing service;
(3) start a timeout job to count down to the auction expiration on auction service; and
(4) inform the inventory service that the item is currently pending auction.

B. *Buyer: Bid*

Buyers can submit bids for collectibles which are up for auction. To do this, a series of calls are made, namely:

(1) get bidder information from user service;
(2) update listing service on change of highest bidder and offer (if any); and
(3) deduct the bid amount from the user's account in user service.

C. *Market: Reconcile*

As the market closes, sellers are matched with the highest bidders, successful buyers receive their kopi collectibles, and all involved parties receive an email notification of the auction outcome.

(1) auction service initiates expiration when the timeout job for a listing has ended;
(2) listing service is updated to indicate that a listing has been sold;
(3) auction service obtains the highest bidder and offer amount from listing service;
(4) inventory service is updated to reflect the new owner of the sold item;
(5) user service is updated for all unsuccessful buyers to have their bid amount returned; and
(6) notification service is called to send email notifications to all involved parties on auction outcome.

IV. MICROSERVICE ARCHITECTURE

Kopi Time is built using a layered, service-oriented architecture: UI, gateway, composite, atomic and wrapper layers. All our services are written in Python or TypeScript.

---

[1] Leettari. (2020, November 8). Squidward Coffee Animation GIF. Tenor. Retrieved November 15, 2021, from https://tenor.com/search/coffee-animations-gifs.

[2] Squarespace. (2021). Squarespace Rotating Coffee Cup. Squarespace. Retrieved November 16, 2021, from https://images.squarespace-cdn.com/content/v1/60d038c7c9c09c1fa7dbbde1/1624342467860-GBMAX9TZ0JFKBDXQCSP2/coffee-cup-illustration-sofia-varano-perth.

[3] Dribble. (2021). Rotating Coffee GIF. Dribble. Retrieved November 16, 2021, from https://cdn.dribbble.com/users/1054552/screenshots/2532001/media/f8375f75a642f66c898bdfc987956b0d.gif.

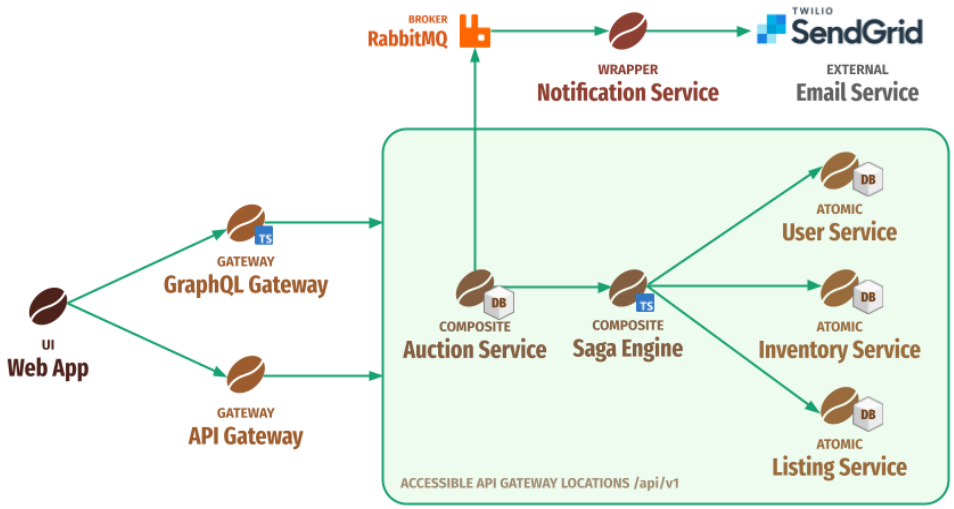

Figure 2. Architecture diagram of Kopi Time

A. *Web App (UI)*

Our web application is built with Next.js. The application is server-side rendered for a quick initial page load. Authentication is managed with client-side cookies and server-side sessions. For security, clients do not have direct access to session information. Instead, authenticated requests are proxied through the application server which then injects sensitive session data before passing the request onto the backend infrastructure. Each page on our web application is also supported by the GraphQL gateway.

B. *GraphQL Gateway*

Our GraphQL gateway is a unified data querying interface for our web application. The web application has pages that require data from multiple microservices joined together. For example, on the /listings page, listing information is required from the listing service, but each listing also needs a reference to the item being listed from the inventory service. Additional information is also required for the current user's details and inventory. Therefore, GraphQL was introduced to support the data requirements of our web application. Refer to the section on self-directed research for more information.

C. *API Gateway*

Similar to the GraphQL gateway, our NGINX API gateway acts as a reverse proxy and provides a single point of access for external clients to access Kopi Time's services. Unlike the GraphQL gateway, it handles all other requests apart from fetching data (GETs) and it does not perform API composition. The API gateway does not contain any edge functions, but nonetheless, it loosens coupling as clients do not need to know where each microservice is at. This encapsulates the internal structure of Kopi Time and protects the internal microservices from being accessed directly, while enabling extensibility for future design decisions such as toggling access to certain endpoints, performing edge functions, and applying the strangler pattern with API versioning.

D. *Auction (Composite)*

The composite auction service is responsible for three functions in relation to the key scenarios described earlier. It has one endpoint exposed for clients to auction an item and another for clients to bid on an existing auction. It also has an internal function inaccessible to clients to expire an existing auction once it expires. For all three functions, the auction service will then craft a sequential saga transaction request in JSON format which is then sent to the Saga Engine. The Saga Engine will then perform the necessary steps to adjust the monetary balance of the involved parties (user service), reserve and transfer ownership of transacted items (inventory service) and update the listing details of the auction (listing service).

The auction service uses the privileged role of "SERVICE" when performing some of these actions (e.g., adjusting the balance of the user). More about the "SERVICE" role is discussed in the JWT User Authentication section under self-directed research. The auction service will then trigger the notification service through AMQP to notify all involved users. With brevity in mind, the exact implementation details and steps taken by the auction service are documented in comment blocks within the source code of the auction service's *src/app.py* file.

Internally, the auction service uses Advanced Python Scheduler (APScheduler) to run a background scheduler with a MySQL database as a persistent datastore to schedule auction expiration jobs whenever a new auction is created. For scheduling consistency, all timestamps involved in Kopi Time microservices are in UTC, except for the frontend web application where it is localised. When an auction expires, the scheduled auction expiry job will trigger the expiry function as discussed earlier.

E. Saga Engine (Composite)

The Saga Engine is an orchestration engine that handles transactions across multiple microservices. The primary goal is to provide a single framework-agnostic implementation of the saga pattern. Refer to the section on self-directed research for more information.

F. User (Atomic)

The user service has three primary functions. Firstly, it issues tokens via its login endpoint for system-wide use, which checks the validity of a submitted username and password before creating and returning the JWT (more on JWTs under self-directed research). The user service also allows user creation via a registration endpoint. It rejects users that attempt to register with an existing username, as the username is used as a primary key within the Kopi Time ecosystem. Finally, the user service allows user information such as user email and account balance to be read and updated by users and other services.

G. Inventory (Atomic)

The inventory service handles simple CRUD operations, such as getting, adding, and deleting inventory. There are two methods that support the auction service: (1) reserve an item to prevent duplicate listings and (2) transfer an item between owners. As we are dealing with collectibles, we decided to make each item one-of-a-kind.

H. Listing (Atomic)

The listing service is primarily concerned with keeping track of the status of listings that have been created. To do this, the service has three main functions. Firstly, the service allows for creation of new listings and includes in the listing information the seller, item ID, highest bidder, and highest bid. Secondly, the service also has an endpoint to update the highest bidder whenever a new bid for the listing is received. This endpoint also handles situations where the incoming bid is higher than the auto win amount and marks the item as sold. Lastly, the service handles the logic of updating expired auction listings based on their outcome. Listings which expire while having at least one bidder will be marked as sold to the highest bidder, while listings without bidders will be marked as closed.

I. Notifications (Wrapper)

The notification service consumes AMQP messages in JSON format from a notifications queue bound to the notifications exchange on the RabbitMQ broker. The service differentiates messages via their routing keys, which specify the type of notification being sent – successful bid, auto-win, item sold, etc. The notification service then selects the email template meant for the requested notification type and populates it with the details specified in the message payload. To send the emails, the notification service uses an external API called SendGrid and injects the necessary credentials via environment variables. The notification service acts as a wrapper for this external API. This helps to decouple the external API from our internal systems, making it easy to maintain and extend our notification service in the future.

V. DEVOPS TOOLS & PRACTICES

Gene Kim's "Three Ways" were incorporated in several different forms throughout our development process, giving us a deeper appreciation of their efficacy.

In achieving *flow*, we found a Kanban board particularly useful for visualising how work was flowing. The board allowed us to: (1) track the status of various tasks, (2) increase awareness of what each team member was working on at each point of time, and (3) gain a sense of progress on the system as a whole. Early on, we also realised that multiple team members would need to use different service endpoints locally, e.g., for service integration. However as more services were added, it started taking extensive time and effort for each member to update their own API client environments with changes made to other services' APIs. Hence, we elevated this constraint by creating a shared Postman workspace, such that each developer could update the shared workspace with the changes they made, thereby allowing other developers to jump straight into development and integration.

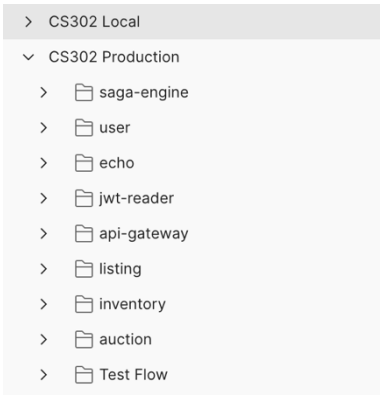

Figure 3. Shared Postman Workspace

To ensure fast and constant *feedback*, each CI pipeline was set up concurrently with its corresponding service to ensure that feedback would be provided throughout the development process. Fast feedback was also enabled via our use of pre-commit hooks (*flake8*, *pylint*, *black* and *isort* for Python, and *eslint* and *prettier* for TypeScript), which allowed us to perform static analysis on fresh code even before a commit was made. Local testing was also enabled via Docker Compose, which allowed us to perform integration and component testing before committing any code. This immediate feedback mechanism ensured that the team only committed and pushed clean, working, and well-formatted code as far as possible. Our team would also swarm to solve

problems together via Telegram or by hopping on a quick Discord call if multiple services were involved in the problem, instead of passing the problem from person to person (much like in a tiered support model). In building our microservices, we also started integration between services early on rather than developing each service in silos, allowing us to understand the endpoints and architecture that would need to be added and fixed on each service to enable coordination between services. This helped to ensure that we would not have to modify the services extensively at the end during a last-minute integration session.

Our team also supported a safety culture for *continual learning*, where bugs/issues found were brought up factually and energy was focused on discussing the fixes necessary rather than on pushing blame around.

VI. CI/CD PIPELINES

For each microservice repository in Kopi Time, we set up individual CI/CD pipelines with four stages: static analysis, test, release and deploy. Due to the purpose of the microservices and time constraints, some repositories do not have certain pipeline stages; in particular, the frontend web application, GraphQL gateway and notification service do not have the test stage, and the API gateway does not have the static analysis stage. The entire pipeline is automatic, meaning that there are no manual stages in the pipeline. The pipeline is triggered by a developer when their code is pushed to the main branch of the microservice repository.

At the static analysis stage, we run various language-specific static analysis jobs including *flake8* and *pylint* for microservices written in Python, and *eslint* and *prettier* for microservices written in TypeScript. These static code analysis jobs help ensure that our codebase meets coding standards and provides feedback on whether there are code smells present, so that we can take necessary measures to fix them before deployment.

At the test stage, we run integration tests on our atomic services (user, inventory, and listing services), component tests on our composite services (auction and saga), and a configuration test for the NGINX API gateway. For the component test in the auction service, we created WireMock test stubs with canned Saga Engine responses. We performed this sort of black box testing on our composite services as it made testing for us closer to being more realistic, while still ensuring that we are testing this service in isolation without spinning up the real services. This allowed us to take advantage of building loosely coupled services with their own individual pipelines.

At the release stage, the Docker image is built and pushed into the container registry of each microservice repository on GitLab. Every time the pipeline is triggered, a new image is built with a new image hash/ID, producing smaller and more reliable releases.

At the deploy stage, we used a custom cloud deploy job (see open-source contribution in the self-directed research section) to automatically process the deployment of the Docker image into the Elastic Container Service (ECS) cluster on Amazon Web Services (AWS). The cloud deploy script accesses the CI/CD variables configured in the repository to determine the ECS cluster, service, task definition and container name to update. An IAM user role is configured with permission to update ECS task definitions. The script uses the user role to update ECS task definitions and roll-out updates to the ECS services. Once the task definition has been updated and the service is rolled out, the container will privately authenticate with GitLab using a configured deploy token stored in AWS Secrets Manager to pull the image stored in the microservice's container registry on GitLab during the release stage.

If any stage in the pipeline fails, the entire pipeline will fail, preventing potentially bad code from being deployed into production. We would then have to rectify the issue and make another commit to trigger the pipeline once more.

VII. DEPLOYMENT TO PRODUCTION

Locally, we set up a Docker Compose environment to run Kopi Time. More details on how to run this can be found in the API documentation or within the README file of the "kopi-time" repository itself.

For production, we created an ECS cluster on AWS to host Kopi Time's microservices, with each microservice defined as an EC2 task definition and configured as an EC2 ECS service. We use an S3 bucket to store our production environment file for our microservices, which includes database connection details, a production JWT secret key, service account credentials, RabbitMQ credentials, SendGrid API credentials and template information, time zone information, and URLs that point to our various services. We also set up a production MySQL database using Amazon Relational Database Service (RDS). When using the services on AWS, we constantly monitored our billing amount to ensure that our team does not exceed the allocated budget of USD 278.

VIII. SELF-DIRECTED RESEARCH

Through the journey of developing Kopi Time, our team conducted self-directed research on various concepts – some of which have already been introduced and briefly discussed in the above sections, including the Saga Engine and GraphQL gateway. The following subsections will further flesh out the details of each self-directed research concept we have explored.

A. *JWT User Authentication*

We implemented JWTs at the service level through the Flask-JWT-Extended package to keep security close to each service. Initially, we explored JWT authentication on the API Gateway with Kong but felt that a service-level implementation would offer greater flexibility and simplicity

for implementing different levels of authorization at the path-level. The user service generates all JWTs, which are used for authorization on each service. All JWTs are signed with a secret to prevent tampering, thereby ensuring data integrity of the JWT tokens. This secret is shared by all Kopi Time microservices and used to verify the JWT signature.

When a user logs in with their username and password, a fresh JWT is generated and returned to the web application. This JWT includes the email, role, and balance of the user for use by other services. There are currently two roles – "USER", a logged-in user, and "SERVICE". The latter is only obtainable by a Kopi Time service which uses the service username and password which are defined on user service start-up. These roles are used to provide access control for certain actions. For example, only a service is allowed to make a transfer between two users when a bid expires. For development purposes, all services share the same account, but this is easily modifiable to provide more granular access control.

B. CloudWatch

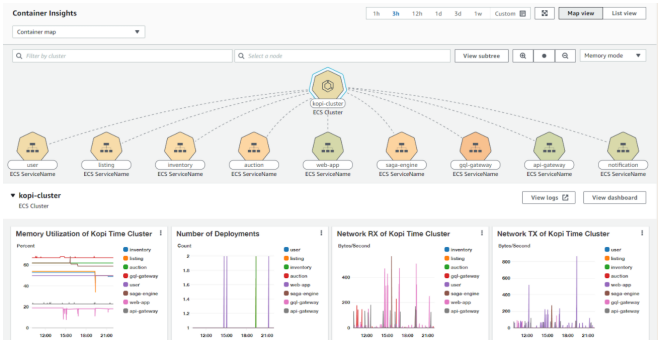

Figure 4. Container map of Kopi Time cluster on AWS

a) Monitoring: ECS has service-related telemetry data readily available on CloudWatch, including CPU and memory utilisation, network traffic, and a lot more. We were able to monitor the CPU and memory utilisation using a heatmap showing all our services running in the cluster. This then paved the way to proactive alerting where we took the metrics and created anomaly detection alarms that sent us email alerts whenever it detected abnormal memory utilization within our services. Using this insight, we discovered a low-memory issue on the frontend web application, which we then allocated more memory to.

b) Logging: We also implemented centralised logging on all our microservices with sidecar containers defined in each of our ECS task definitions. These sidecar log routers helped our team tremendously as they automatically instrumented and collected the output logs of our main microservice containers, simply by defining environment variables in the task definition. The logs can then be easily accessed as individual log streams on CloudWatch. This value-added to our project – especially so for a complex system like Kopi Time – because it gave us greater visibility into our system. We managed to see problems as they occur, and it led to faster feedback.

C. Open-Source Contribution

Whenever we updated a microservice and the deploy job kicked in, we expected that GitLab's ECS *cloud deploy* image would properly update the image tag of our container in the ECS task definition with the new one. However, it overwrites the image tags of *all* containers present in the task definition, including the sidecar log router. This breaks the ECS service when it attempts to run the log router sidecar container using the main container's image.

The problem is that GitLab's script lacked the feature to indicate which container to update when deploying. We went to the root of the problem and added that feature into GitLab's cloud deploy script and made this update public by opening a merge request on GitLab's official cloud deploy repo, so that developers can globally benefit from our local discovery. This fixed the problem we were facing, and we are now using this updated cloud deploy script for all our microservice deployments.

D. Production WSGI

Knowing that Flask's built-in servers are not production-ready, we also ran our services on a production-quality WSGI server called Waitress.

E. GraphQL

Traditionally, developers had two main options to retrieve data from multiple microservices: (1) make multiple requests through the Internet to each microservice and join data locally; or (2) develop specific REST endpoints on a gateway for each data join required. Both options are not ideal. Making multiple requests through the Internet introduces additional latency for each request made, thereby degrading the user experience. An alternative is to develop specific REST endpoints for each data join required. However, this tightly couples the development of the querying application (frontend) to the backend. Changes to the frontend are blocked by changes to the backend, and the backend must maintain endpoints for each specific join of data required on the frontend.

GraphQL solves both issues by allowing a single request to define the data joins required between data objects from different microservices. This allows data to be joined within the fast internal network instead of over the Internet and served through one request. This also moves the responsibility of defining the structure and joins of data closer to the frontend.

F. Saga Engine

The Saga Engine provides a framework-agnostic implementation of the saga pattern.

Each transaction is treated as a flow of data through different microservices. To remain framework-agnostic,

transactions are defined in a superset of JSON named Saga Flow Definition (SFD). SFD allows us to describe how data from one service is passed to other services. In addition, SFD also supports conditional dispatching, custom failure conditions, and custom authentication methods per service. More details about SFD can be found on the README of the Saga Engine repository.

The Saga Engine then defines a unique ID for the transaction and executes the transaction by dispatching requests to other microservices. This unique ID is logged to provide better process visibility in the log files. Currently, only synchronous HTTP dispatches are supported. However, we plan on supporting AMQP messaging in the future.

The main limitation of the Saga Engine is that it cannot define custom compensation logic. Microservices being called by the Saga Engine must provide a pre-defined endpoint to receive compensation requests. Unfortunately, this couples the microservice to the Saga Engine. In addition, microservices must handle concurrency issues such as race conditions and deadlocks internally. We plan on developing this idea further to find solutions to these problems.

IX. CRITICAL REFLECTIONS

A. What Went Well

The entire development process was largely smooth as the team put in regular work throughout the duration of the project. We had weekly meetings to discuss next steps and review the work done during the week, enabling clear communication and a steady rate of progress. The weekly labs also served as a great starter pack for us to reference when creating our services and tests (special shoutout to the provided *wait-for-it.sh* script). Our team was also very collaborative in general, with each member readily giving and receiving help and feedback, thus elevating our combined productivity as a team.

The team also made good progress with our preliminary exploration into the Saga Engine. This was made possible by our team culture of fast and constant feedback. We continually discovered new limitations with the Saga Engine and quickly worked to solve them.

The loose coupling between microservices enabled parallel development of services and granted flexibility to adjust implementation details without affecting other services. For example, our notifications service went through three phases: we initially planned to use Firebase, subsequently considered using Gmail, and eventually implemented email notifications via SendGrid. Despite these changes to our notification system, the development of other services could continue without a hitch.

B. What We Would Change

a) Standardised Interface: The development of the auction composite service would have been easier if there was a standard interface across each atomic service, such as following a standard template for the "message" field in the JSON response.

b) Service Discovery: Instead of hardcoding URLs, we could use service discovery solutions (e.g. Netflix Eureka, AWS Route 53) to dynamically discover other microservices on the network.

c) Bidding History on Listing Service: Currently, we do not persist bidding information of past bidders in our listing service, only the current highest bidder and the previous highest bidder. If we had more time, we would have changed how this worked by introducing another "bidding" table (or "bidding" service) to store each bid transaction.

d) Staging Area: We could add an intermediary staging environment in between development and production – the staging area will allow for manual QA testing in a near-production environment. To grant more control over production, staging would have continuous deployment, while production would have manual deployment.

e) EC2 to Fargate Instances: Our ECS task definitions and services are currently configured to use EC2 instances, meaning they are all being run on a single ECS cluster instance. Changing EC2 to Fargate would have allowed our services to run with their own elastic IP addresses and would have given us better access to service discovery.

C. What We Could Have Explored

a) Infrastructure as Code (IaC): We would like to explore IaC tools such as Terraform or CloudFormation to declaratively define and deploy infrastructure in a consistent and centralised manner. By automating this process, the infrastructure supporting Kopi Time can be provisioned much quicker in a less error-prone manner.

b) End-to-end/UI testing: Given time constraints and the requirements of our project, we only implemented integration and component tests on the various services in our application. However, given more time, we would have liked to explore additional testing methods such as end-to-end and UI testing (e.g., Cypress, Selenium) to ensure the complete correctness and health of our application. Since the application would be tested thoroughly, the chances of frequent breakdowns and repetitive testing efforts would be greatly reduced. Adding these automated tests and increasing our overall test coverage would also reduce the chances of failure and thus further increase our confidence in the function and performance of the application.