



Waterford Institute *of* Technology

Automation of Public House Back-end Business Processes

Dimitri Saridakis

Work submitted
within the
BSc in Applied Computing
Cloud & Networks

Supervisor: Dr Kieran Murphy
Second Reader: Clodagh Power

January 3, 2022

©2021 – DIMITRI SARIDAKIS
All rights reserved.

Contents

| | |
|---|------------|
| List of Figures | iii |
| Acknowledgements | 1 |
| Link to Demo Video | 2 |
| 1 Introduction and Project Goals | 3 |
| 1.1 Introduction | 3 |
| 1.1.1 Motivation | 3 |
| 1.1.2 Scope | 5 |
| 1.1.3 Side Benefits | 6 |
| 1.1.4 Planning and Strategy | 6 |
| 2 Architecture and Technologies | 7 |
| 2.1 Architecture | 7 |
| 2.1.1 Initial Architecture | 7 |
| 2.1.2 Current Architecture | 9 |
| 2.2 Technologies Used | 10 |
| 2.2.1 Kubernetes | 10 |
| 2.2.2 Transaction API for System Entry-Point | 11 |
| 2.2.3 Debezium | 12 |
| 2.2.4 Apache Kafka | 13 |
| 2.2.5 The Strimzi Operator | 15 |
| 2.2.6 Helm | 15 |
| 3 Implementation | 17 |
| 3.1 Kubernetes Ready | 17 |
| 3.1.1 API Containerization | 17 |
| 3.1.2 PostGres Containerization | 18 |
| 3.2 Deployment | 20 |
| 3.2.1 Automated Deployment | 20 |
| 3.2.2 Kubernetes Secret Object For PostGres and API | 21 |
| 3.2.3 PostGres Persistent Volume (PV) | 22 |
| 3.2.4 PostGres Persistent Volume Claim (PVC) | 22 |
| 3.2.5 PostGres Deployment | 23 |
| 3.2.6 PostGres Service | 24 |
| 3.2.7 API Deployment | 24 |
| 3.2.8 Test Deployment Thus Far | 25 |

Contents

| | | |
|----------|--|-----------|
| 3.2.9 | Strimzi Deployment | 25 |
| 3.2.10 | Kafka Connect for Debezium | 26 |
| 3.2.11 | Debezium PostGres Connector Deployment | 27 |
| 3.3 | Testing the System | 27 |
| 3.4 | Challenges | 31 |
| 4 | Semester One Conclusion | 33 |
| 4.1 | Semester One Technical and Personal Thoughts | 33 |
| 4.1.1 | Technical Thoughts | 33 |
| 4.1.2 | Personal Thoughts | 34 |
| 5 | Semester Two Plan | 36 |
| 5.1 | Work to be Completed | 36 |
| 5.1.1 | Kafka Consumers | 36 |
| 5.1.2 | Category Two Processes | 37 |
| 5.1.3 | Category Two Architecture | 39 |
| 5.1.4 | Future Work Plan | 40 |
| 6 | Supporting Material | 41 |
| 6.1 | Manifests | 41 |
| 6.1.1 | API Deployment Manifest | 41 |
| 6.1.2 | Docker-compose | 42 |
| | Bibliography | 43 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Initial Architecture | 8 |
| 2.2 | Current architecture of the system depicting the interaction between components. | 10 |
| 2.3 | The Kubernetes control loop, sourced from [44]. | 11 |
| 2.4 | Companies that use Apache Kafka in Production | 13 |
| 2.5 | Writing to Kafka Topic example | 14 |
| 2.6 | Operator control loop based on the Kubernetes <i>Observe, Act, Analyze</i> control loop. Sourced from [28]. | 15 |
| 2.7 | Helm showing the status of the Strimzi Operator deployment. | 16 |
| 3.1 | Transactions API Dockerfile. | 18 |
| 3.2 | Running the build script to generate the new Debezium friendly PostGres docker image. | 19 |
| 3.3 | The initial and re-tagged PostGres friendly image. | 19 |
| 3.4 | The deployment Makefile. | 20 |
| 3.5 | The Kubernetes Secret resource to be applied to the cluster with the real values altered. | 21 |
| 3.6 | Kubernetes Secrets in Cluster | 21 |
| 3.7 | Persistent Volume Manifest | 22 |
| 3.8 | Manifest binding the persistent volume to the PostGres database. | 22 |
| 3.9 | PostGres deployment manifest. | 23 |
| 3.10 | PostGres service manifest. | 24 |
| 3.11 | Logs for the API container displays migrations to the database | 25 |
| 3.12 | Helm successfully deploys the Strimzi Kafka Operator which in turn deploys the Kafka components. | 25 |
| 3.13 | Kafka Connect Custom Resource Definition manifest. | 26 |
| 3.14 | Debezium PostGres Connector definition manifest. | 27 |
| 3.15 | Kafka cluster information managed by the Kafka Entity Operator | 28 |
| 3.16 | The API's exposed URL via the service that was configured previously. | 29 |
| 3.17 | Testing the endpoint with Postman. The response can be observed. | 29 |
| 3.18 | Kafka Consumer Output | 30 |
| 3.19 | Proof of the newly created beer in the Kafka beers topic | 31 |
| 5.1 | Architecture of the Kafka Consumers | 37 |
| 5.2 | Proposed architecture of Category Two system and how it integrates with the Category One architecture. | 39 |
| 5.3 | Risk Register of components left to complete. | 40 |

List of Figures

| | | |
|-----|--|----|
| 5.4 | Gantt Chart for semester two. | 40 |
| 6.1 | Kafka Connect Custom Resource Definition manifest. | 41 |
| 6.2 | docker-compose used in the initial testing of the API and PostGres deployment. | 42 |

Acknowledgements

I would like to thank my supervisor, Dr Kieran Murphy, for his fantastic help and excellent guidance throughout this process. It's been a pleasure working with him.

Link to Demo Video

A 5-minute video outlining the system in operation, followed by a further deep dive into the technical side of the project is available here <https://youtu.be/x00TMmJFhmQ>.

1 Introduction and Project Goals

1.1 Introduction

This report's structure will follow this style:

- The report will outline the project's goals, the system architecture, the technologies used and the plan which was followed.
- It will detail the implementation of the various system components from the start to the working prototype.
- The report will outline the challenges faced in the implementation of the system.
- A conclusion will be given including both technical and personal reflection.
- Detailed analysis on proposed work to be completed in semester two.

1.1.1 Motivation

The motivation for this project comes from a number of major pain points of mine, from my previous / side career as a bar owner and manager.

This project focuses on building a system for use by bar / public houses (pubs) and as such a single bar entity will be referred to as a *user* of the system.

There is also scope for this system to be altered slightly and to be used with any business which operates with a similar back-end structure.

Some of the most time-consuming and least valuable, from a time - reward perspective, are the back-end processes of running the business. Reward is defined here as the actions which result in potential business growth. Time spent scanning invoices from suppliers, filling out income and expenditure spreadsheets and calculating gross and net figures (which will be referred to as '*group A*' activities). Whilst these processes are critical to a business' operation and regulatory compliance they do not do much for business growth. On the other hand, time spent on sourcing new products / inventory, finding new / novel forms of entertainment, business promotion and customer engagement (which will be referred to as '*group B*' activities) are the catalysts which drive sales and business growth.

The ultimate aim of this project is to provide more time for group B growth activities and processes by automating the group A processes.

1 Introduction and Project Goals

I hypothesize that this should lead to a healthier and more innovative industry by virtue of the extra amount of time spent on group B activities. With implementation of this system, barriers to entry should be broken down which should only help to increase innovation. This comes from the new entrants into the industry who may excel in group B processes but do not have the knowledge, cannot afford to pay accountants or have the confidence in their ability to perform the group A processes at a satisfactory standard. If these processes are automated then there should be fewer barriers of entry coupled with a reduction in accounting costs. Furthermore, the implementation of this system should increase the quality of life of business owners who no longer have to carry out menial, manual data entry and monotonous, simple and repetitive data manipulation.

I will use this project as a vehicle to explore and test both industry standard and brand-new technologies with an emphasis on open-source tech. This will stress test my knowledge of the technologies and both allow me to see what I can implement along with being a showcase of my skills in the field to potential employers. This system will be built with the primary target of having each component implemented in such a way as to ensure maximum efficiency is the front and center focus.

For this to be accomplished there is a list of core and essential processes that need to be tackled.

These core processes fall into two broad categories, defined as:

1. **Category One:** comes from the data collected from a sale of a user's product, i.e., a pub selling a beverage (Income).
2. **Category Two:** stems from the data collected from a user's purchases in relation to inventory and other purchases needed for the running of the business, i.e., a pub buying a crate of beer to be resold to the consumer or rent for premises (Expenditure).

Category One core processes include:

- Keeping a record of all sale transactions that enter the system, sorted by user, which will allow for the processing of sales.
- The subsequent saving and updating of the transactional sales figures, i.e., gross, net and tax figures.
- The updating of the inventory levels of products per sale.

Category Two core processes include:

- The scanning of supplier invoices and key information extraction from the invoices. This key information will be used to:
 - Update user's inventory levels as stock is invoiced / delivered.
 - Updating of cash flow levels to reflect the current available funds.

- Updating of tax collected and tax due figures.

This is quite a lengthy and complex list of processes to automate.

1.1.2 Scope

This project will focus on tackling all the Category, One and Two, processes. Given the time constraints and the complexity of the system in development there are some non-core components that will be omitted or generated / faked. These are documented here to allow the reader to know that these processes have been thought about thoroughly, before the decision was made to continue as detailed.

These include:

1. For the Category One processes, the users' sales transactions and details will be faked / generated. This is done through the main entry API which has a `.../transactions/fake/create` endpoint. There will be no Point of Sale (PoS) till software created.
 - If time constraints allow, there are plans to incorporate transactional data from a current free and popular PoS application. This will most likely be SquareUp PoS [47] by Square / Block as my business uses this PoS currently and there is a developer API which may potentially be leveraged to provide the transactional data into the system.
2. There will be no GUI to interact with or view data from a user standpoint. This would involve creating a web app which would be used to view data from the system about a user. This would include sales figures, inventory levels, cash flow levels and tax figures.
 - A sub goal is to implement a Grafana dashboard to at least view some monetary / inventory figures from the system.
3. All the Category Two processes will be automated. The only area to note here is a possible discrepancy between newly ordered inventory invoiced and the product actually delivered by a supplier to a business.

Sometimes products which have been ordered are not delivered, i.e., out of stock with supplier or damaged in transit.

 - For a solid fix for this situation the system should utilize a way of comparing delivery dockets with invoices. This would ensure only as accurate data as possible enter the system. However, these situations happen infrequently and for this project the potential discrepancy will be ignored with the invoice taken at face value of goods delivered.

1.1.3 Side Benefits

There are many useful features which become available as a result of having all of this information available in one system. These insights come in the form of individual data per business but also trends and such from the data aggregated from all users of the system. A brief example of some of these include:

- The ability to query the financial and inventory figures. With a simple GUI the user can be served up current sales vs other time periods and many other powerful ways to gain insight into the business with information already in the system.
- The ability to do some exploratory data analysis (EDA) and other data analytical activities which can provide the business owners with some new data driven insights about their business. These insights would usually only be available to larger businesses with IT teams or businesses with owners who are data science savvy. Businesses with these characteristics in the drinks' industry make up only a tiny fraction of the population based on my decade plus of experience within the industry in Ireland and around numerous European cities.
- An example of another insight that can be derived from the information in the system which is of value to external entities, given decent levels of adoption in the industry, are live sales per product. Having a multitude of different users in the system, the sale quantities of specific items can be accessed and / or extrapolated in real-time. This can provide some invaluable data to brewers, of sales which could be utilized to precisely schedule production times and production quantities.

1.1.4 Planning and Strategy

The strategy for the first semester was to tackle the Category One processes. ClickUp [3] is used as the project management solution (more details in Chapter 4: Conclusion). The aim was to have a prototype ready for the second semester. The second semester would see work continue on the system to implement some testing and adding to the overall robustness of the system along with automation of the Category Two processes. This plan has been nearly been achieved. The addition of custom consumers (will be explained in detail later) is the next step. Until otherwise specified, the focus will be on the architecture and implementation of the system — Category One specific.

2 Architecture and Technologies

2.1 Architecture

The architecture for this project has changed significantly from the initial design in the hunt for an ever optimal and efficient solution to the problem. Even though the diagram in Figure 2.1 is rough, it has been included for contrast and to depict the evolution to the current implementation.

2.1.1 Initial Architecture

The initial design centered around the deployment of a number of APIs and databases as micro-services. These microservices would be deployed as containers and orchestrated by Kubernetes [4]. There would be an initial entry-point API which would then call other API endpoints. Each API would be the primary vehicle for the flow of data through the system into the required databases / components. These APIs would both read and write to the databases.

This design, although sound and would provide the intended utility was far from an optimal solution. There would be a gargantuan number of expensive database calls with this design, violating a core target for the system to be as efficient as possible. The design is not very scalable either. With the addition of more users, the APIs and databases would need to be scaled up to accommodate the growth of the system. The components in this design are unnecessarily, tightly coupled.

2 Architecture and Technologies

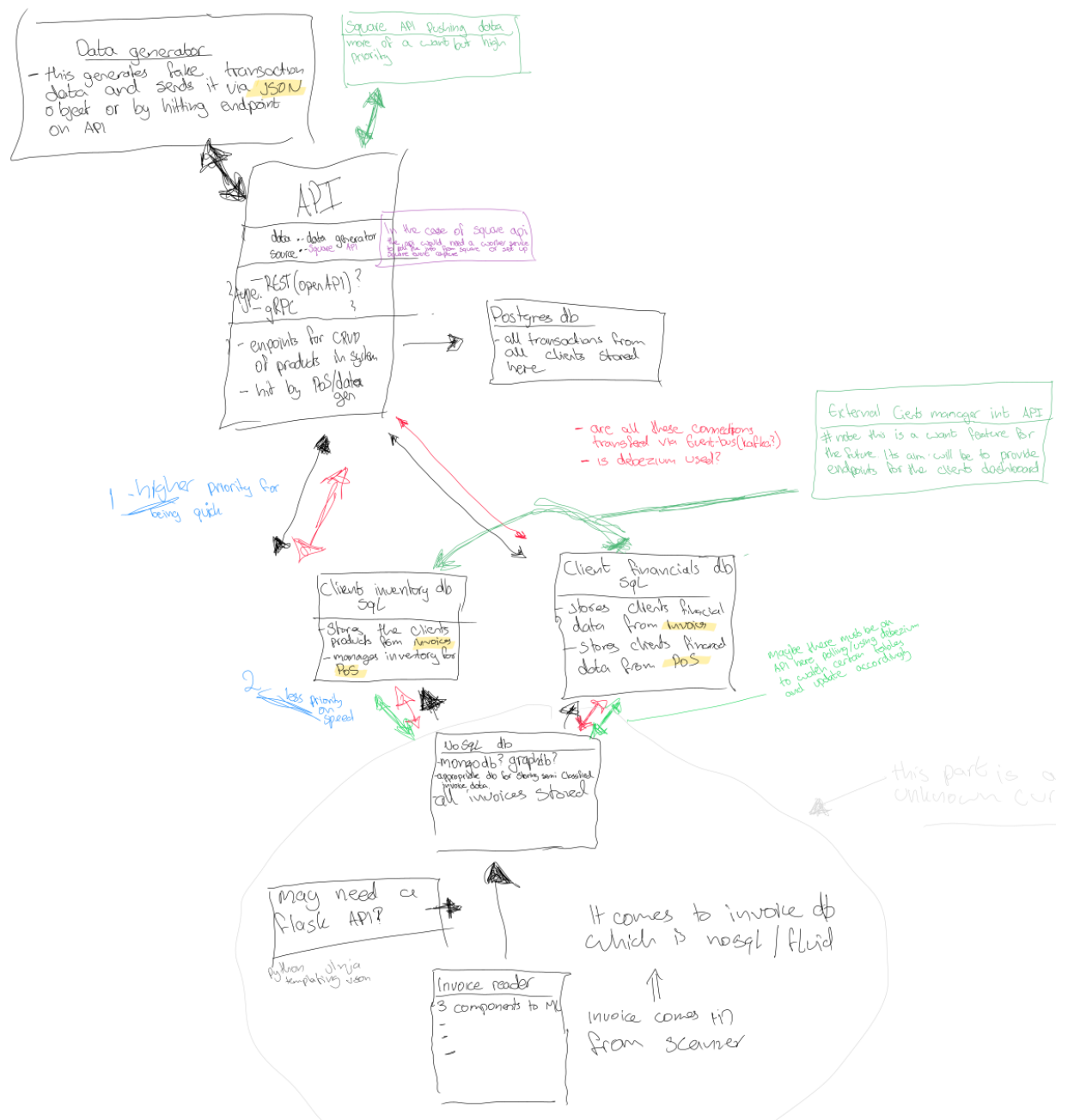


Figure 2.1: Initial architecture of the system depicting the interaction between components and data flow.

2.1.2 Current Architecture

The search for efficiency led to the morphing of the microservice architecture to an event-driven [59] microservice architecture. This means that components in the system no longer wait for requests, as is the traditional request-driven model, but the components react instead, in real-time, to changes of state in the system. This evolution was accomplished by the research and subsequent implementation of a new technology, *Change Data Capture* (CDC).

CDC is a software solution which identifies and tracks changes in data in a database [30]. The software then propagates these changes to the next steps in the data flow. Other microservices can then consume these changes and action upon the changed data. This allows for the further decoupling of the system as consumers / subscribers to the change stream can be added / upgraded or removed entirely without any alteration to, or downtime of, the system.

The chosen CDC technology for implementation is Debezium [8]. Debezium monitors a database and ingests all monitored changes to Apache Kafka [2] topics via Kafka Connect (more details to follow). The Kafka topics act as the stream of data to which consumers can subscribe and consume the data from.

An API is still needed as the data entry point into the system. This Transactions API is referred to as ‘*tapi*’ in the repo.

The steps needed for the implementation of the current architecture are:

1. The configuration of an API (*tapi*) which exposes an endpoint external to the Kubernetes cluster.
2. Once that endpoint is hit, *tapi* creates a sales transaction which is saved to the transactions PostGres database.
3. The Debezium PostGres connector is configured to listen for changes in the state of that PostGres database.
4. Upon a monitored state change Debezium will then utilize Kafka Connect to ingest the updates, those updates are subsequently pushed to Kafka Topics.
 - The Kafka components must be configured and are deployed via the Strimzi Operator.
 - The Strimzi Operator itself is deployed by Helm (Kubernetes package manager).
5. A consumer which is subscribed to a Kafka Topic to consume the changes and prove the system is operating as intended.

The deployment orchestration is managed by Kubernetes, in this case a single Kubernetes node is used with the utilization of Minikube[33]. The following is the system architecture diagram:

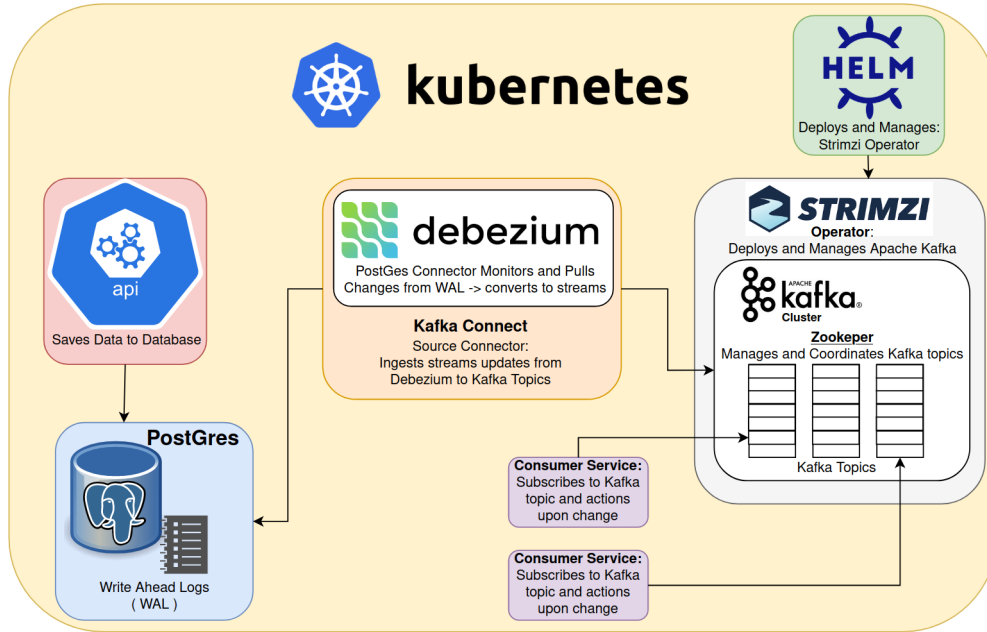


Figure 2.2: Current architecture of the system depicting the interaction between components.

Note: The system architecture has depicted a component that is not 100% accurate. Strimzi deploys and manages the Kafka Connect component, via a deployment manifest, and the initial diagram was implemented as such. However, since Debezium is implemented on top of Kafka Connect but not deployed by or managed by Strimzi, the decision was taken to remove the Kafka Connect component from the Strimzi component for better clarity, with this explanation in place to address questions.

All the black arrows in the diagram do not depict a flow of data through the system. They are a representation of the interactions between components. As an example: The arrow from **Helm** to **Strimzi** represents the deployment of the Strimzi operator component by the Helm package manager.

2.2 Technologies Used

2.2.1 Kubernetes

Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management. It was originally designed Google and is now maintained by the Cloud Native Computing Foundation (CNCF) [40]. Kubernetes has become the de facto industry standard for container orchestration. It is used in production by a host of the world's largest companies including Google,

Spotify, Adidas and Airbnb to name a few. These companies migrated to Kubernetes to increase speed and efficiency whilst decreasing down-time [19]. It is a technology whose adoption continues to grow at a rapid pace [27]. As such, it was one of the first technologies considered for this project.

Kubernetes can be a huge and complex system with multiple ways to implement it. For the sake of brevity this report will only explain Kubernetes components explicitly relevant to this project. This project utilizes Kubernetes for orchestration, deployment and fault-tolerance.

Fault-tolerance is taken care of by Kubernetes as Kubernetes routinely compares pods and services currently active / healthy and restarts the pods which have deviated from the desired configurations or found themselves in an ‘unhealthy’ state [46].

Kubernetes’ services are deployed in a cluster via a deployment manifest. The deployment manifest is a file in either JSON or a YAML file format which describes the desired state of the service. The Kubernetes control loop, as per Figure 2.3, is responsible for monitoring the state of the cluster and making changes as necessary. If a service is not running or has drifted from the desired state, the control loop restarts a fresh service from the deployment manifest.

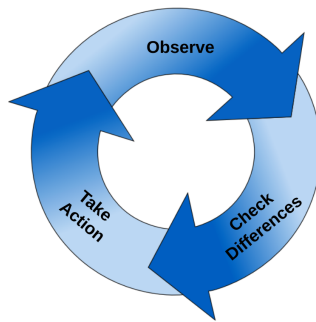


Figure 2.3: The Kubernetes control loop, sourced from [44].

Most other Kubernetes resources are created in the same manner and are *applied* to the cluster using `kubectl`. Kubectl is a command line tool which is used to interact with the Kubernetes cluster. Resources are applied using the `apply` command.

`kubectl apply -f resource-manifest-file-name` is used to apply a resource to the cluster.

2.2.2 Transaction API for System Entry-Point

A transaction API was developed for creation and saving of transactional data. The API is written in GoLang and uses Gorilla Mux (a high performance HTTP router

package). This API is operational but far from perfect in its current implementation. More testing and validation are required to ensure the API's robustness. As it stands it is merely a working prototype and is still currently in development.

The API exposes a `.../transactions/fake/create` endpoint external to the Kubernetes cluster which creates a fake transaction when hit. This transaction is a typical transaction that would be found in a pub / bar. It has a transaction owner, this is the entity for which the transaction has been possessed, in this case a pub / bar. The transaction contains a random number of beers with some randomly generated information for fields like name, ABV, price etc.

The API then attempts to save the transaction to the Postgres Transactions database container (details of the configuration to follow).

2.2.3 Debezium

Debezium is an open source Change Data Capture (CDC) technology which is configurable with a number of different connectors. There are specific connectors for each of the supported databases. Postgres, MySQL and MongoDB are some of the supported databases [5].

Debezium is implemented by the likes of Reddit, Shopify and Ubisoft. It is also integrated with and a driving force in a number of other technologies such as Apache Camel, Google Cloud DataFlow and IBM Event Streams [61].

Debezium is most commonly deployed via Apache Kafka's Kafka Connect framework [25]. Once a Debezium connector is applied to continuously monitor a database, it ingests the changes and then utilizes the Kafka Connect component to push the changes to Apache Kafka topics. It lets any of your applications / services stream every-row level change whilst preserving the order by which the changes were committed to the database [8]. In the case of Postgres, it does this by monitoring Postgres' *Write Ahead Logs*. These are binary logs of every event to the database. These include not only all CRUD updates to the database but schemas along with schema modifications also.

Debezium may also be deployed via the Debezium server [7]. This is a ready-to-use application which streams changes to a variety of different messaging systems including Redis, AWS Kinesis, Google Pub / Sub, and more.

There are a number of great reasons to use Debezium. It is an extremely efficient way to capture changes to a database because of the way it utilizes the WAL, Debezium can process changes in the database without having to interact with the database via the usual methodologies of expensive database calls, i.e., SELECT, INSERT etc. The conversion of these changes to streams can allow for multiple services to access the data changes relevant to them without having to interact with the database, so the database's primary job becomes servicing the incoming requests from the API.

This project will utilize Debezium's Kafka Connect implementation along with the aforementioned Postgres connector.

2.2.4 Apache Kafka

Apache Kafka is a high performance, distributed, fault-tolerant, event streaming broker / event bus. It was originally developed at LinkedIn but was open sourced in 2011 and is now maintained by the Apache Software Foundation. It is currently in use in more than 80% of all Fortune 100 companies.

APACHE KAFKA

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

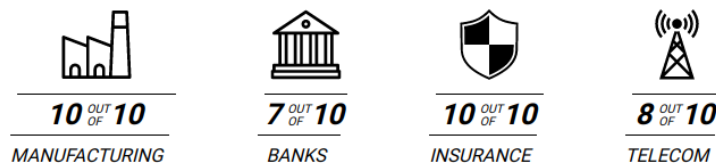


Figure 2.4: An example of the types of companies that use Apache Kafka in production, sourced from [2].

Event streaming is the process of capturing live events such as a CRUD operation to a database or other data from services. These services can be applications of any type. Mobile apps, web applications, microservices, IoT devices, etc.

According to Kafka documentation [2], it combines three key capabilities, namely:

- To allow publishers to write and subscribers to read streams of events, including continuous import / export of data from other systems.
- To store streams of events durably, reliably and in a fault-tolerant replicated manner.
- To process streams of events in real time or retrospectively as desired.

2 Architecture and Technologies

Kafka can be simply thought of as a distributed system which consists of servers and clients. Kafka can span multiple nodes and be run anywhere. Communication between the two happens via the TCP protocol.

- *Servers*: The servers that make up the storage layer of the system are called **brokers**. These brokers save events to topics. These topics are like logs, brokers append the events to the topic and consumers read from the topic via an offset. This ensures that the topics are read in order. Topics may be partitioned and replicated across multiple brokers. This ensures fault-tolerance and durability. Events can be read by a multitude of consumers and are kept for any configurable amount of time. Other servers run **Kafka Connect** which handles the continuous import and export of data as event streams to the topics.
- *Clients*: The clients that make up the application layer of the system are called consumers. These consumers read from topics and process the events. This can be done in parallel and as such can be used to process data at large scale.

Apache Zookeeper manages and coordinates the brokers. Kafka uses it to ensure data durability. If a leader node / broker fails, Zookeeper ensures a new leader is chosen without any consequence to data in the system via a leader election. It is a highly scalable, high-availability, distributed coordination system [60].

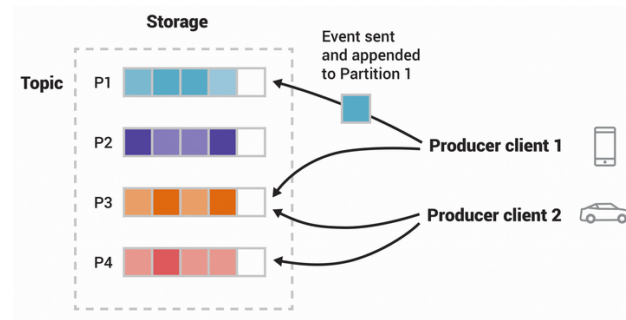


Figure 2.5: Simplified view of a partitioned topic being written to by multiple producers[2]. In this project there will be just one producer, the Debezium Kafka Connect instance.

2.2.5 The Strimzi Operator

Strimzi is an open-source Kubernetes operator that deploys Apache Kafka in a Kubernetes cluster using the operator pattern.

Operators are extensions to Kubernetes that are deployed using a *Custom Resource Definition* (CRD). A Kubernetes operator is a method of packaging, deploying and managing a Kubernetes application [57]. Operators are essentially just a non-Kubernetes-native piece of software that extends the functionality of Kubernetes. Operators follow the Kubernetes control loop and other Kubernetes principals. An operator can be thought of as a client of the Kubernetes API that acts as a controller for a Custom Resource [36]. Their goal is to bring the Kubernetes core concept of automation to non-Kubernetes components but with the added elements of domain / application-specific knowledge about the application it deploys via its own set of preconfigured and configurable CRDs that are specific to the application it deploys. Operators aim to automate the entire life cycle of the software under their control.

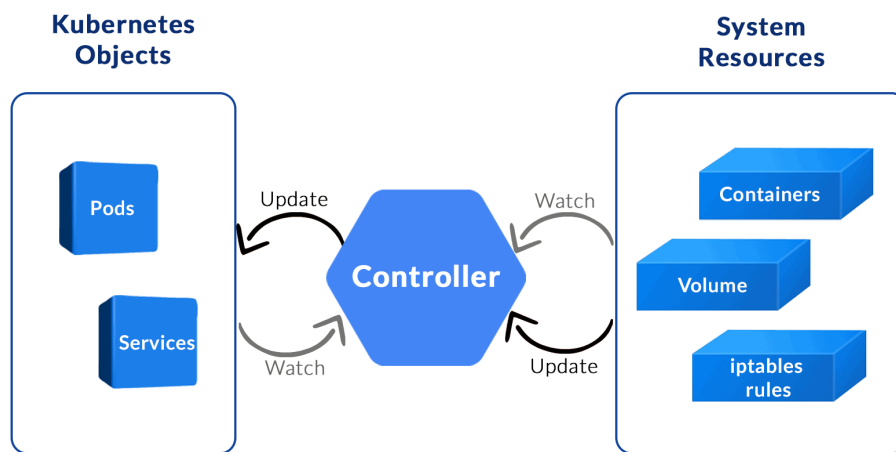


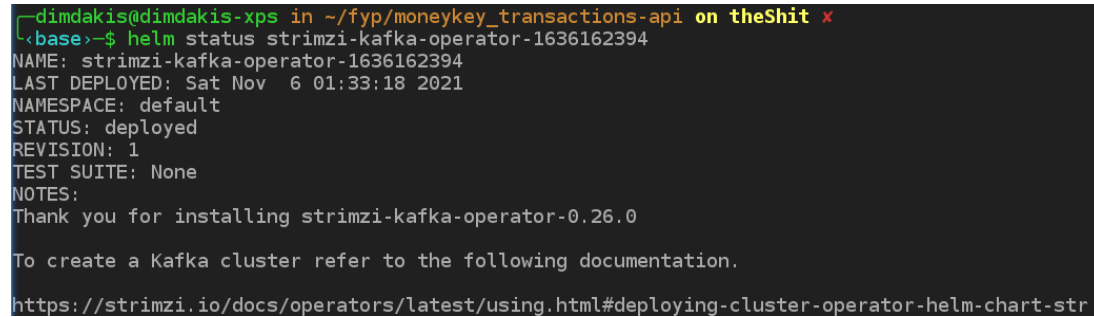
Figure 2.6: Operator control loop based on the Kubernetes *Observe, Act, Analyze* control loop. Sourced from [28].

2.2.6 Helm

This project also utilizes **Helm** which is a Kubernetes package and operations manager[53]. Helm is used to install and manage Kubernetes packages. These packages are usually referred to as **Charts**. Helm Charts are packaged in a **tar.gz** format and usually contain at least a Kubernetes Deployment object and a Kubernetes Service Object but can contain any number of other Kubernetes objects to satisfy the installation dependencies of the desired application [56].

2 Architecture and Technologies

This project utilizes Helm for the installation, deployment and management of the Strimzi Operator. After the installation of Helm, the Strimzi Chart repo is added to the Kubernetes cluster `helm repo add strimzi https://strimzi.io/charts/`. The Strimzi Operator is installed by running the `helm install strimzi/strimzi-kafka-operator` command.



```
dimdakis@dimdakis-xps in ~/fyp/moneykey_transactions-api on theShit x
(base)-$ helm status strimzi-kafka-operator-1636162394
NAME: strimzi-kafka-operator-1636162394
LAST DEPLOYED: Sat Nov 6 01:33:18 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thank you for installing strimzi-kafka-operator-0.26.0

To create a Kafka cluster refer to the following documentation.
https://strimzi.io/docs/operators/latest/using.html#deploying-cluster-operator-helm-chart-str
```

Figure 2.7: Helm showing the status of the Strimzi Operator deployment.

3 Implementation

Note: Some screenshots used in this section are re-used screenshots from a research paper that I completed for Cloud Computing 1. I leveraged that research paper to explore Debezium in great detail and to ascertain if its implementation was feasible and optimal for this project.

3.1 Kubernetes Ready

Minikube is started by running `$ minikube start`.

3.1.1 API Containerization

As Kubernetes deploys containers as services, every service (microservice) needs to be containerized. The Transactions API is the first service containerized.

The Transaction API container is built based on a Dockerfile found here [\[10\]](#) with some modifications. The reason for the two-step build is to keep the container size as small as possible. The API is built then the necessary files needed to run it are used in the final step.

3 Implementation

```
1 # Start from golang base image
2 FROM golang:alpine as builder
3 # Add Maintainer info
4 LABEL maintainer="Dimitri Saridakis <dimitri.saridakis@gmail.com>"
5
6 # Install git.
7 # Git is required for fetching the dependencies.
8 RUN apk update && apk add --no-cache git
9
10 # Set the current working directory inside the container
11 WORKDIR /app
12
13 # Copy go mod and sum files
14 COPY go.mod go.sum ./
15
16 # Download all dependencies. Dependencies will be cached if the go.mod and the go.sum files are not changed
17 RUN go mod download
18
19 # Copy the source from the current directory to the working Directory inside the container
20 COPY . .
21
22 # Build the Go app
23 RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
24
25 # Start a new stage from scratch
26 FROM alpine:latest
27 RUN apk --no-cache add ca-certificates
28
29 WORKDIR /root/
30
31 # Copy the Pre-built binary file from the previous stage. Observe we also copied the .env file
32 COPY --from=builder /app/main .
33 COPY --from=builder /app/.env .
34
35 # Expose port 8080 to the outside world
36 EXPOSE 8080
37
38 #Command to run the executable
39 # CMD ["./main"]
40 ENTRYPOINT [ "./main", "serve" ]
```

Figure 3.1: Transactions API Dockerfile.

There was a lot of testing carried out at this stage in perfecting the image and deploying both a PostGres container along with the API container. Most of the testing here was done using `docker-compose`.

In the API design, a custom logger was implemented, and its role was invaluable in debugging these steps from code to container to deployment. Every warning / error being documented from the exact line of code were the event occurred along with the event message ensured that debugging was swift. Something else to note here is that the choice of GoLang for API design was a good one. In GoLang every error is treated as an error object and must be handled graciously to follow the Golang convention. This is not strictly enforced in the language as it is possible to just ignore the error object, but that has obviously negative effects on the application.

3.1.2 PostGres Containerization

This is not trivial. By default, Debezium streams events in ‘pgoutput’ format. A JSON format is the desired format so for this a ‘wal2json’ plugin was downloaded. A new custom docker image had to be created which would specify the wal2json plugin to create a PostGres container which is capable of writing data to JSON format.

The Debezium connector specific components must also be included. As per the official documentation [32] There are two ways to do this.

- One is to install the components on a running PostGres database then containerize that database.

3 Implementation

- Luckily, the second method is far more straight-forward. The Debezium project has a repo on their GitHub [12] that makes this process easier with a dedicated build script. The repo was forked and git cloned locally. Then the build script was run as per Figure 3.2.

```
dimdakis@dimdakis-xps in ~/fyp/cluster/docker-images on main ✓ (origin/main)
(base)-$ build-postgres.sh 13

*****
** Building debezium/postgres:13
*****
Sending build context to Docker daemon 11.78kB
Step 1/16 : FROM postgres:13 AS build
13: Pulling from library/postgres
7d63c13d9b9b: Already exists
cad0f9d5f5fe: Already exists
ff74a7a559cb: Already exists
c43dfd845683: Already exists
e554331369f5: Already exists
d25d54a3ac3a: Already exists
bbc6df00588c: Already exists
d4deb2e86480: Already exists
d4132927c0d9: Pull complete
3d03efa70ed1: Pull complete
645312b7d892: Pull complete
3cc7074f2000: Pull complete
4e6d0469c332: Pull complete
Digest: sha256:1adb50e5c24f550a9e68457a2ce60e9e4103dfc43c3b36e98310168165b443a1
Status: Downloaded newer image for postgres:13
--> 113197da0347
```

Figure 3.2: Running the build script to generate the new Debezium friendly PostGres docker image.

Whilst the build script did the job it was intended to do it saved the image as “debezium/postgres:tag-name”. This is somewhat problematic as there is already an image with that name on docker hub. To overcome this obstacle the image had to be re-tagged as per Figure 3.3. Not a lot of work, but the process could just be a tad bit more user-friendly.

```
dimdakis@dimdakis-xps in ~/fyp/cluster/docker-images on main ✓ (origin/main)
(base)-$ docker image ls
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------------|--------|--------------|--------------|-------|
| debezium/postgres | 13 | 4d5e4b180e1f | 40 hours ago | 624MB |
| dimakis/debezigres | latest | 4d5e4b180e1f | 40 hours ago | 624MB |

Figure 3.3: The initial and re-tagged PostGres friendly image.

3.2 Deployment

Deployments in Kubernetes are made up of one or more identical replicas of a containerized application, as specified in the deployment manifest. Once a deployment is applied to the cluster. The Kubernetes Deployment Controller will automatically create the required number of replicas along with making sure that if any pods go down or become unresponsive new instances will be spun up in their place.

A Kubernetes Pod is simply a collection of containers that are run on a single node. Each pod can be thought of as a running process in the Kubernetes environment [38]. Each container in a pod can share networking and storage resources with other containers in the same pod.

Deployment of the system to Kubernetes must be done in a non-arbitrary order as some services depend on others. If the steps are done in the wrong order then the pods may get stuck in a `CrashLoopBackOff` state.

3.2.1 Automated Deployment

To simplify the deployment process, to ensure deployment in the correct order and to aid in the automation of the project, the initial section of deployment is farmed out to a Makefile. The Makefile is a simple shell script that runs the necessary commands to deploy the PostGres database container and then the API along with the required Kubernetes resources. It is simply run using `$ make deploy`:

```
10  # creates tapi, its dp and service
11  .PHONY: deploy
12  deploy:
13      kubectl apply -f postgres-secret.yaml
14      kubectl apply -f postgres-db-pv.yaml
15      kubectl apply -f postgres-db-pvc.yaml
16      kubectl apply -f postgres-db-deployment.yaml
17      kubectl apply -f postgres-db-service.yaml
18      kubectl apply -f tapi-deployment.yaml
19      kubectl apply -f tapi-service.yaml
20
```

Figure 3.4: The deployment Makefile.

3.2.2 Kubernetes Secret Object For PostGres and API

The first resource that is created is a Kubernetes Secret object. A Kubernetes Secret object is a file that contains sensitive information such as password, tokens, private IP addresses / URLs, keys or any other data that the engineer chooses to hide from the application [45] as per Figure 3.5. This is the industry standard method for dealing with sensitive data with Kubernetes.

```

You, seconds ago | 1 author (You) | io.k8s.api.core.v1.Secret (v1@secret.json)
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: postgres-secret
5  type: Opaque
6  stringData:
7    POSTGRES_USER: dimakis
8    POSTGRES_PASSWORD: password
9    POSTGRES_DB: moneykey-t-api-postgres
10
11   DB_HOST: moneykey-t-api-postgres
12   DB_DRIVER: postgres
13   API_SECRET: 98hbun98h
14   DB_USER: dimakis
15   DB_PASSWORD: password
16   DB_NAME: moneykey-t-api-postgres
17   DB_PORT: '5432'
18

```

Figure 3.5: The Kubernetes Secret resource to be applied to the cluster with the real values altered.

```

dimakis@dimakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud_Computing_1 - Term Paper
(base)-$ kube get secrets
NAME                                TYPE                                DATA  AGE
default-token-qmwv5                 kubernetes.io/service-account-token 3      22d
kafka-cluster-clients-ca            Opaque                              1      22d
kafka-cluster-clients-ca-cert       Opaque                              3      22d
kafka-cluster-cluster-ca            Opaque                              1      22d
kafka-cluster-cluster-ca-cert       Opaque                              3      22d
kafka-cluster-cluster-operator-certs Opaque                              4      22d
kafka-cluster-entity-operator-certs Opaque                              4      22d
kafka-cluster-entity-operator-token-8zrdq kubernetes.io/service-account-token 3      22d
kafka-cluster-kafka-brokers          Opaque                              4      22d
kafka-cluster-kafka-token-xhnnpn    kubernetes.io/service-account-token 3      22d
kafka-cluster-zookeeper-nodes        Opaque                              4      22d
kafka-cluster-zookeeper-token-6vp2v  kubernetes.io/service-account-token 3      22d
my-connect-cluster-connect-token-lmhgb kubernetes.io/service-account-token 3      19d
postgres-secret                     Opaque                              10     22d
sh.helm.release.v1.strimzi-kafka-operator-1636162394.v1 helm.sh/release.v1                  1      22d
strimzi-cluster-operator-token-x46wd  kubernetes.io/service-account-token 3      22d
dimakis@dimakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud_Computing_1 - Term Paper

```

Figure 3.6: Kubectl showing the secrets available in the cluster.

Note: The secrets of type ‘Opaque’ are user defined secrets. Also note that on this system `kubectl` has long been aliased to `kube`.

The secret values are then used to fill the environmental variables (ENV vars) in the API container.

3.2.3 PostGres Persistent Volume (PV)

The next resource object that is created is a Kubernetes Persistent Volume. A Kubernetes Persistent Volume is a storage resource that is backed by a file system. It allows for the application to store data in a persistent manner even though each instance of the application is ephemeral.

```
1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: postgres-pv-volume
5    labels:
6      type: local
7      app: moneykey-t-api-postgres
8  spec:
9    storageClassName: manual
10   capacity:
11     storage: 5Gi
12   accessModes:
13     - ReadWriteMany
14   hostPath:
15     path: "/mnt/data"
16   persistentVolumeReclaimPolicy: Retain
```

Figure 3.7: Manifest provisioning a persistent volume of size 5 GB which will be used for the PostGres database.

3.2.4 PostGres Persistent Volume Claim (PVC)

The next resource that is created is a Kubernetes Persistent Volume Claim. A Kubernetes Persistent Volume Claim is a request for an existing Persistent Volume to be used with an application. The previously provisioned Persistent Volume is then bound to the Persistent Volume Claim for use with the PostGres database.

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: postgres-pv-claim
5    labels:
6      app: moneykey-t-api-postgres
7  spec:
8    storageClassName: manual
9    accessModes:
10     - ReadWriteMany
11    resources:
12      requests:
13        storage: 5Gi
```

Figure 3.8: Manifest binding the persistent volume to the PostGres database.

3.2.5 PostGres Deployment

The next resource to be deployed is the PostGres deployment. In the deployment manifest the custom container image along with other required resources such as the secret from which the ENV vars should be populated, the persistent volume to be used and location to mount to along with the ports to be exposed are specified.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: moneykey-t-api-postgres
5    labels:
6      app: moneykey-t-api-postgres
7  spec:
8    selector:
9      matchLabels:
10       app: moneykey-t-api-postgres
11       tier: postgres
12    replicas: 1
13    template:
14      metadata:
15        labels:
16          app: moneykey-t-api-postgres
17          tier: postgres
18      spec:
19        containers:
20          - image: debezium/postgres:latest
21            name: postgres
22            imagePullPolicy: "IfNotPresent"
23            envFrom:
24              - secretRef:
25                name: postgres-secret
26            ports:
27              - containerPort: 5432
28                name: postgres
29            volumeMounts:
30              - mountPath: /var/lib/postgresql/data
31                name: postgres-persistent-storage
32          volumes:
33            - name: postgres-persistent-storage
34              persistentVolumeClaim:
35                claimName: postgres-pv-claim
36

```

Figure 3.9: PostGres deployment manifest. The yellow ‘warning’ lines courtesy of VS Code, come from the Kubernetes extension. The warning is a result of not setting a resource limit on the deployment. As it is not yet known what kind of resources the deployment will need at this stage, this warning will be ignored.

3.2.6 PostGres Service

The last step to enable the database to run as intended is a Kubernetes service object. A 'service' object is the medium through which pods communicate with other pods in the system. Without exposing a service the database will not be accessible.

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: moneykey-t-api-postgres
5    labels:
6      app: moneykey-t-api-postgres
7  spec:
8    type: NodePort
9    ports:
10     - port: 5432
11     selector:
12       app: moneykey-t-api-postgres
13       tier: postgres
```

Figure 3.10: PostGres service manifest.

3.2.7 API Deployment

The API deployment is a simpler process. It only needs a deployment manifest along with a service manifest. These are very similar to the PostGres versions as detailed above, save for application specific differences and as a result they will not be documented, however it is available in the 'Supporting Material' section.

3.2.8 Test Deployment Thus Far

A simple test to determine if communication is possible between the API and the database, is to get the logs from the API container. As migrations are handled in the API, if these migrations appear in the logs then the system is nominal thus far:

```

dimdakis@dimdakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing 1 - Term Paper
(base)-$ kube logs tapi-85846fd4db-f9rxz
Nov 28 18:44:35 [INFO] logger/logger.go:26 replaced zap's global loggers
Nov 28 18:44:35 [INFO] app/main.go:33 logger initialized
Nov 28 18:44:35 [INFO] cmd/serve.go:36 serve called
Nov 28 18:44:35 [INFO] api/app.go:28 We are getting the env values
Nov 28 18:44:40 [INFO] controllers/app.go:56 created singleton postgres db connection
we are connected to the postgres database
2021/11/28 18:44:40 /app/pkg/controllers/app.go:62
[warn] Model github.com/dimakis/moneykey_transactions-api/pkg/models.User don't match BeforeSaveInterface,
e https://gorm.io/docs/hooks.html

2021/11/28 18:44:40 /go/pkg/mod/gorm.io/driver/postgres@v1.1.2/migrator.go:194
[110.722ms] [rows:1] SELECT count(*) FROM information_schema.tables WHERE table_schema = CURRENT_SCHEMA() AND
table_name = 'users'

2021/11/28 18:44:40 /go/pkg/mod/gorm.io/driver/postgres@v1.1.2/migrator.go:74
[85.151ms] [rows:1] SELECT CURRENT_DATABASE()

2021/11/28 18:44:40 /go/pkg/mod/gorm.io/driver/postgres@v1.1.2/migrator.go:310
[199.961ms] [rows:-] SELECT column_name, is_nullable, udt_name, character_maximum_length, numeric_precision,
e_precision, 8 * typelen FROM information_schema.columns AS cols JOIN pg_type AS pgt ON cols.udt_name = pgt.
name AND table_schema = CURRENT_SCHEMA() AND table_name = 'users'

```

Figure 3.11: Logs for the API container displays migrations to the database. This shows the two containers are communicating and behaving as expected.

3.2.9 Strimzi Deployment

Kubernetes package manager Helm is used to deploy the Strimzi operator. Once Strimzi is deployed it will deploy a Kafka Broker, the Kafka Cluster Entity Operator and Apache Zookeeper. Once the Strimzi Helm Chart URL is added, as described in the *Kubernetes Ready* section, it is deployed using this command

`helm install strimzi/strimzi-kafka-operator -n default` where the `-n` flag specifies the namespace to deploy to.

```

dimdakis@dimdakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing 1 - Term Paper
(base)-$ kube get pods

```

| NAME | READY | STATUS | RESTARTS |
|--|-------|---------|---------------|
| kafka-cluster-entity-operator-5fb4956bcd-m2zw1 | 3/3 | Running | 41 (157m ago) |
| kafka-cluster-kafka-0 | 1/1 | Running | 3 (157m ago) |
| kafka-cluster-zookeeper-0 | 1/1 | Running | 15 (158m ago) |

Figure 3.12: Helm successfully deploys the Strimzi Kafka Operator which in turn deploys the Kafka components.

3.2.10 Kafka Connect for Debezium

The final installation and configurations are for Kafka Connect and the Debezium Postgres Connector. The first step is to create a Kafka Connect *Custom Resource* (CR) Kubernetes object. This is done by applying the Kafka Connect *Custom Resource Definition* (CRD) to the cluster.

```

io.strimzi.kafka.v1beta2.KafkaConnect (v1beta2@kafkaconnect.json)
1  apiVersion: kafka.strimzi.io/v1beta2
2  kind: KafkaConnect
3  metadata:
4    name: my-connect-cluster
5    annotations:
6      strimzi.io/use-connector-resources: "true"
7  spec:
8    image: dimakis/strimzi-kafka-connect:latest
9    replicas: 1
10   bootstrapServers: kafka-cluster-kafka-bootstrap:9092
11   # tls:
12   #   trustedCertificates:
13   #     - secretName: my-cluster-cluster-ca-cert
14   #       certificate: ca.crt
15   config:
16     config.storage.replication.factor: 1
17     offset.storage.replication.factor: 1
18     status.storage.replication.factor: 1
19     config.providers: file
20     config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
21   externalConfiguration:
22     volumes:
23       - name: connector-config
24         secret:
25           secretName: postgres-secret
26

```

Figure 3.13: Kafka Connect Custom Resource Definition manifest.

Note: In the metadata the:

```

annotations:
  strimzi.io/use-connector-resources: "true"

```

This allows for the Strimzi operator to automatically create the Kafka Connect resources based on this CRD.

Furthermore, a custom image is used. There were only minor changes from the default to suit this use case.

The traffic throughout the system is unencrypted. This would be a security concern in a production environment, however, for the sake of the time needed to add TLS, the benefit doesn't out-weigh the time outlay. In particular with the time spent on the configuration, testing and debugging of the system.

The `postgres-secret` is used here for the Kafka Connect cluster to connect to the

PostGres database as without credentials it will not be authorised to connect.

3.2.11 Debezium PostGres Connector Deployment

The last step is to create the Debezium Connector. This is done by applying the Debezium Connector *Custom Resource Definition* (CRD) to the cluster.

```
io.strimzi.kafka.v1beta2.KafkaConnector (v1beta2@kafkaconnector.json)
1  apiVersion: kafka.strimzi.io/v1beta2
2  kind: KafkaConnector
3  metadata:
4    name: transactions-connector
5    labels:
6      strimzi.io/cluster: my-connect-cluster
7  spec:
8    class: io.debezium.connector.postgresql.PostgresConnector
9    tasksMax: 1
10   config:
11     database.hostname: moneykey-t-api-postgres
12     database.port: 5432
13     database.user: dimakis
14     database.password: password
15     database.server.id: 18405454
16     database.server.name: moneykey-t-api-postgres
17     database.dbname: moneykey-t-api-postgres
18     database.whitelist: moneykey-t-api-postgres
19     database.history.kafka.bootstrap.servers: kafka-cluster-kafka-bootstrap:9092
20     database.history.kafka.topic: schema-changes.moneykey-t-api-postgres
21     key.converter: "org.apache.kafka.connect.json.JsonConverter"
22     include.schema.changes: "true"
23     plugin.name: wal2json
24     key.converter.schemas.enable: "true"
25     value.converter.schemas.enable: "true"
```

Figure 3.14: Debezium PostGres Connector definition manifest.

In this manifest the `labels` and `spec` are particularly important.

The `my-connect-cluster` label refers to the name given to the Kafka Connect cluster. This is what binds the Debezium connector to the Kafka Connect cluster.

The `config` fields must match both the PostGres database details along with the appropriate Kafka configurations.

3.3 Testing the System

There are a number of ways to test the setup to see if everything is operational as intended. The first and quickest is to use `$ kube get strimzi`. This should return all Kafka topics and other data about offsets etc.

3 Implementation

```

dindakis@dindakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing 1 - Term Paper
base-~$ kube get strimzi

```

| NAME | CLUSTER | CONNECTOR CLASS | MAX TASKS | READY |
|--|--------------------|--|-----------|-------|
| kafkaconnector.kafka.strimzi.io/transactions-connector | my-connect-cluster | io.debezium.connector.postgresql.PostgresConnector | 1 | True |

| NAME | DESIRED KAFKA REPLICAS | DESIRED ZK REPLICAS | READY | WARNINGS |
|--------------------------------------|------------------------|---------------------|-------|----------|
| kafka.kafka.strimzi.io/kafka-cluster | 1 | 1 | True | |

| NAME | REPLICATION FACTOR | READY | CLUSTER | PARTITIONS |
|---|--------------------|-------|---------------|------------|
| kafkatopic.kafka.strimzi.io/beers | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/connect-cluster-configs | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/connect-cluster-offsets | 1 | True | kafka-cluster | 25 |
| kafkatopic.kafka.strimzi.io/connect-cluster-status | 1 | True | kafka-cluster | 5 |
| kafkatopic.kafka.strimzi.io/consumer-offsets--84e7a678d08f4bd226872e5cdd4eb527fad1c6a | 1 | True | kafka-cluster | 50 |
| kafkatopic.kafka.strimzi.io/moneykey-t-api-postgres.beers | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/moneykey-t-api-postgres.public.beers | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/moneykey-t-api-postgres.public.transactions | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/moneykey-t-api-postgres.public.users | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/strimzi-store-topic--effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55 | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/strimzi-topic-operator-kstreams-topic-store-changelog--b75e702040b99be8a9263134de3507fc0cc4017b | 1 | True | kafka-cluster | 1 |
| kafkatopic.kafka.strimzi.io/transactions | 1 | True | kafka-cluster | 1 |

| NAME | DESIRED REPLICAS | READY |
|--|------------------|-------|
| kafkaconnect.kafka.strimzi.io/my-connect-cluster | 1 | True |

Figure 3.15: Kafka cluster information managed by the Kafka Entity Operator. This is the operator that manages both topics and users and the operator itself is deployed and managed by the Strimzi operator.

As can be seen from Figure 3.15, topics have been created with

`..some code.. pulic.beers`, `..some code.. pulic.transactions` and `..some code.. pulic.users`. These are the table names from the PostGres database. It seems at very least the schemas have been created properly.

To test the system in a live way, a simple Kafka consumer is created subscribing to one of these topics. The API endpoint will be exposed to external traffic (external to the Kubernetes cluster). Once this endpoint is hit, the data should be saved to the database where it will progress through the system and be observable via the consumer. The consumer is based on test image made available for this exact purpose via the Strimzi documentation hosted on quay.io. This is the command to create the consumer:

```

kubectrl run kafka-consumer -ti
--image=quay.io/strimzi/kafka:0.26.0-kafka-3.0.0 --rm=true
--restart=Never -- bin/kafka-console-consumer.sh
--bootstrap-server kafka-cluster-kafka-bootstrap:9092
--topic moneykey-t-api-postgres.public.beers

```

The image is pulled, Kubectrl starts a new container from the image, once it is quit it will be removed. It connects to the Kafka Broker and the topic is the beers topic which comes from the `beers` table in the PostGres database.

Minikube allows for an easy way to find the exposed API endpoint via the following command:

3 Implementation

```
dimdakisd@dmdakis-xps in ~/year4/semester
<base>-$ minikube service tapi --url
http://192.168.49.2:30392
```

Figure 3.16: The API's exposed URL via the service that was configured previously.

Postman is used to make a GET request to the service address and hit the `transactions/fake/create` endpoint. This will create a new transaction which we should be able to observe in both the response to postman and the terminal where the consumer is running.

Keep an eye out for the `beerName` value that is returned via the API response. It is a randomly generated beer name "Ruinatation IPA".

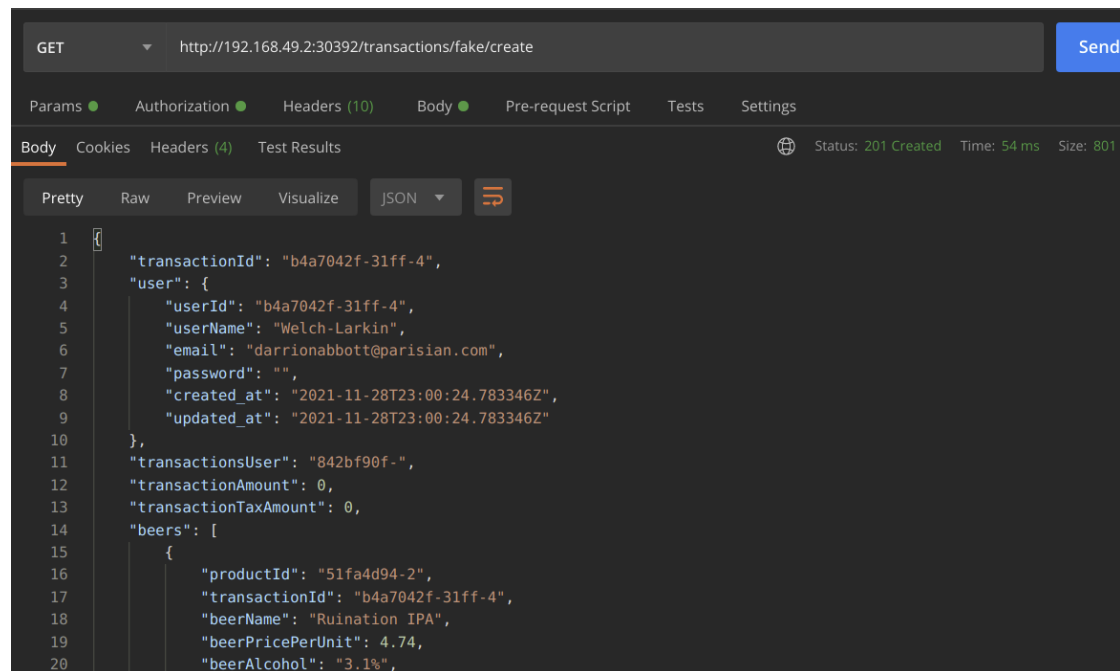


Figure 3.17: Testing the endpoint with Postman. The response can be observed.

3 Implementation

As the response shows, a fake transaction is created. This proves that the API is accessible and that it does its job of creating some fake transaction data.

The final step is to check the output from the Kafka consumer:

```
dimdakisd@dimdakisd-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing I - Term Paper
l.base>~$ kubectl run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.26.0-kafka-3.0.0 --rm=true --rest
art=Never -- bin/kafka-console-consumer.sh --bootstrap-server kafka-cluster-kafka-bootstrap:9092 --topic m
oneykey-t-api-postgres.public.beers
If you don't see a command prompt, try pressing enter.
{"schema":{"type":"struct","fields":[{"type":"struct","fields":[{"type":"string","optional":false,"field":
"product_id"}, {"type":"string","optional":true,"field":"transaction_id"}, {"type":"string","optional":true,
"field":"beer_name"}, {"type":"struct","fields":[{"type":"int32","optional":false,"field":"scale"}, {"type":
"bytes","optional":false,"field":"value"}], "optional":true, "name":"io.debezium.data.VariableScaleDecimal",
"version":1, "doc":"Variable scaled decimal", "field":"beer_price_per_unit"}, {"type":"string","optional":tru
e, "field":"beer_alcohol"}, {"type":"struct","fields":[{"type":"int32","optional":false,"field":"scale"}, {"t
ype":"bytes","optional":false,"field":"value"}], "optional":true, "name":"io.debezium.data.VariableScaleDeci
mal", "version":1, "doc":"Variable scaled decimal", "field":"beer_tax_rate"}, {"type":"int64","optional":true,
"field":"millilitre"}, {"type":"struct","fields":[{"type":"int32","optional":false,"field":"scale"}, {"type":
"bytes","optional":false,"field":"value"}], "optional":true, "name":"io.debezium.data.VariableScaleDecimal",
"version":1, "doc":"Variable scaled decimal", "field":"price_per_ml"}], "optional":true, "name":"moneykey_t_ap
i_postgres.public.beers.Value", "field":"before"}, {"type":"struct","fields":[{"type":"string","optional":fa
lse, "field":"product_id"}, {"type":"string","optional":true, "field":"transaction_id"}, {"type":"string","opt
ional":true, "field":"beer_name"}, {"type":"struct","fields":[{"type":"int32","optional":false, "field":"scal
e"}, {"type":"bytes","optional":false, "field":"value"}], "optional":true, "name":"io.debezium.data.VariableSc
aleDecimal", "version":1, "doc":"Variable scaled decimal", "field":"beer_tax_rate"}, {"type":"int64","optio
nal":true, "field":"millilitre"}, {"type":"struct","fields":[{"type":"int32","optional":false, "field":"scal
e"}, {"type":"bytes","optional":false, "field":"value"}], "optional":true, "name":"io.debezium.data.VariableSc
aleDecimal", "version":1, "doc":"Variable scaled decimal", "field":"price_per_ml"}], "optional":true, "name":"m
oneykey_t_api_postgres.public.beers.Value", "field":"after"}, {"type":"struct","fields":[{"type":"string","o
ptional":false, "field":"version"}, {"type":"string","optional":false, "field":"connector"}, {"type":"string",
"optional":false, "field":"name"}, {"type":"int64","optional":false, "field":"ts_ms"}, {"type":"string","optio
nal":true, "name":"io.debezium.data.Enum", "version":1, "parameters":{"allowed":["true","last","false"],"default":
"false", "field":"snapshot"}, {"type":"string","optional":false, "field":"db"}, {"type":"string","optional":fa
lse, "field":"schema"}, {"type":"string","optional":false, "field":"table"}, {"type":"int64","optional":true,
"field":"txId"}, {"type":"int64","optional":true, "field":"lsn"}, {"type":"int64","optional":true, "field":"xmi
n"}, {"type":"string","optional":false, "name":"io.debezium.connector.postgresql.Source", "field":"source"}, {"type":"string",
"optional":false, "field":"op"}, {"type":"int64","optional":true, "field":"ts_ms"}], "optional":false, "name":"
moneykey_t_api_postgres.public.beers.Envelope"}, {"payload":{"before":{"product_id":"51fa4d94-2
", "transaction_id":"b4a7042f-31ff-4", "beer_name":"Ruinatation IPA", "beer_price_per_unit":{"scale":2, "value":
"Ado="}, "beer_alcohol":"3.1%", "beer_tax_rate":{"scale":2, "value":"Fw=="}, "millilitre":0, "price_per_ml":{"sc
ale":0, "value":"AA=="}, "source":{"version":"0.10.0.Final", "connector":"postgresql", "name":"moneykey-t-api
-postgres", "ts_ms":1638140424797, "snapshot":"false", "db":"moneykey-t-api-postgres", "schema":"public", "tabl
e":"beers", "txId":607, "lsn":24038952, "xmin":null}, "op":"c", "ts_ms":1638140424835}}
```

Figure 3.18: The data is streamed to the consumer via Debezium and the Kafka components.

Whilst there is a lot of output there, all schema and file type (GoLang structs that create the beer object) changes are being observed and recorded. This is configurable. If the same beer is in the output then we can be sure the system is working as intended.

3 Implementation

```
field": "txId"}, {"type": "int64", "optional": true, "field": "lsn"}, {"type": "int64", "optional": true, "field": "xmin"}], "optional": false, "name": "io.debezium.connector.postgresql.Source", "field": "source"}, {"type": "string", "optional": false, "field": "op"}, {"type": "int64", "optional": true, "field": "ts_ms"}], "optional": false, "name": "moneykey_t_api_postgres.public.beers.Envelope", "payload": {"before": null, "after": {"product_id": "51fa4d94-2", "transaction_id": "b4a7042f-31ff-4", "beer_name": "Ruination IPA", "beer_price_per_unit": {"scale": 2, "value": "Ado="}, "beer_alcohol": "3.1%", "beer_tax_rate": {"scale": 2, "value": "Fw=="}, "mililitre": 0, "price_per_ml": {"scale": 0, "value": "AA=="}, "source": {"version": "0.10.0.Final", "connector": "postgresql", "name": "moneykey-t-api-postgres", "ts_ms": 1638140424797, "snapshot": "false", "db": "moneykey-t-api-postgres", "schema": "public", "table": "beers", "txId": 607, "lsn": 24038952, "xmin": null}, "op": "c", "ts_ms": 1638140424835}}}
```

Figure 3.19: Proof of the newly created beer in the Kafka `beers` topic

Success! The system is working nominally, and that the data produced by the consumer is printed to the terminal instantaneously as the GET request is made via Postman.

3.4 Challenges

There were very many challenges in the configuration of the system. At essentially every point along the way configuration changes had to be implemented.

- The development of the Transaction API and subsequent testing was a challenge. Initially the API ENV vars just exposed a port on the host machine. Once that was operational the switch to containerization meant that most of the ENV vars needed to be altered to suit a containerized environment. As initial testing between containerized API and database container was carried out using Docker-Compose the ENV vars once again needed to be altered to suit a Kubernetes environment.
- The creation of the PostGres container that is configured for use with Debezium went through many iterations. Debezium requires specific configuration to function. Replication permissions are required for a user, in this case a default user 1001, to be enabled. Replication logic itself must be set so that postgresql writes to the WAL in with a `logical_decoding` setting [32].
- Issues were had with docker-compose during the testing phase that required a purge of the docker-compose network. This was a trivial fix but a lack of error reporting from Docker meant that this took much longer than it should have.
- The installation of Kafka via the Strimzi operator was an unfortunately necessary pivot. The initial plan was to deploy Kafka manually via a custom set of CRDs. However, there were very many connection issues and `CrashLoopBackOff` errors. The use of an operator was initially thought to be too much to learn on top of the knowledge needed for the rest of the system. However, the operator made it incredibly easy (comparatively) to deploy the Kafka cluster.

3 Implementation

- The deployment of Strimzi followed the same pattern as the Kafka deployment. Initially set up was using CRDs and the use of Helm only stemmed from the goal of trying to automate the process. In doing so the installation of Strimzi became also much easier and more automated.
- The configuration of the Kafka Connect cluster and the addition of the Debezium Kafka Connector resource were very time-consuming and needed a lot of research into what flags / configurations needed to be passed to the Strimzi Operator. The `kube describe` command was used to check the status of the Kafka Connect clusters along with CRDs whose deployment are automated via operators. For quite a while a mere one field was left out of the configuration manifest. The seemingly obvious `database.server.name: moneykey-t-api-postgres` was omitted for far too long and resulted in a non-functional system.

4 Semester One Conclusion

4.1 Semester One Technical and Personal Thoughts

In the following some technical thoughts around the improvement of the system are discussed. Subsequently, some personal reflection is included.

4.1.1 Technical Thoughts

- The system can easily be set up to be much more fault-tolerant by increasing the amount of replicas of each service in the system. Most replica values used throughout this are only values of 1, but there are quite a few services that comprise the system. So for the benefit of the machine on which the testing was carried out the values were set to 1.
- More system metrics would be the logical next step in the current system. The use of Prometheus [\[41\]](#) would be a great way to monitor the system. Whilst the incorporation of Grafana [\[17\]](#) would simplify the monitoring of metrics by providing incredibly useful visualizations. These technologies, fast becoming industry standard [\[9\]](#), would enable the system to be monitored in real time to ensure no issues and help flag any potential downtime.
With metrics such as these the bottlenecks of the system may be identified and the system's performance can be improved.
- The Makefile should be extended and added to. The aim should be to try and create the entire system in a single command (once Kubernetes is running). This does seem feasible, however, plenty of time would be taken testing how long different components take to start up and if retries / back-offs are required as a result.
- The security of the data at both rest and in transit now needs to be revisited. The addition of mTLS between each component is another sub-goal for the system. The initial design for the system was to implement Kafka without the Strimzi Operator. Strimzi has both positives and negatives. Its positives include ease of deployment and automation of the deployment process in particular with regard to the upgrading of the Kafka component versions. The main negative is the Strimzi Operators incompatibility with the Istio Service Mesh.
The incorporation of a Service Mesh into the architecture, this would sit in Kubernetes and all other components would be situated in the mesh, was the first choice design choice. This was actually the reason for not using Strimzi from

the start. The addition of Istio (a leader in the service mesh tech world and the service mesh with which I am most familiar with) would bring with it a host of security and other benefits from mTLS to the automation of Prometheus, Grafana and a whole host of other metric gathering tools which come natively with an Istio deployment. Trade-offs had to be made in order to achieve a working prototype.

Mutual TLS can still be configured for the transit of data in the Kafka components as is, but the inclusion of a service mesh (which is not easily implemented with Kafka) would be a better option.

- The API, as mentioned previously, needs a lot more work. Even the current faked endpoint could do with some improvement in regards a bug where the total transaction figure is not calculated.
- The current consumer of the data from the Kafka topics merely print out the data to the console. The goal is to have a consumer which saves each user's financial data to a dedicated database, whilst a second consumer does the same with inventory to an inventory database (more detail in following section).

4.1.2 Personal Thoughts

- The use of ClickUp [3] for the management of the project has been invaluable in the development of the system. Keeping a record of the project's progress and the progress of the individual components has been a great help in the development of the system. I've used an agile methodology here with a Scrum-Ban approach. This is not quite either Scrum or Kan-Ban but a relaxed and more flexible mixture of the two. This was the same approach that was used in the early stage development of the project I was working on whilst at my internship at Red Hat. I didn't stick stringently to the ceremonies that make up either approach, as they don't necessarily make sense when working solo. Blocks of work (epics) were first laid out and followed. These blocks of work and the sprints (very loosely defined) were completed in the order that the Implementation section (Chapter 3) was laid out.
- The current architecture is an excellent fit for this project. It allows for scalability, fault-tolerance and additional security whilst being incredibly efficient. I am much happier with the architecture as it stands now as opposed to the initial ideas. This event-driven, microservice architecture is fast becoming the desired architecture of projects in the industry [14].
- The technologies used are also a great fit. They allow for a very loosely coupled system and combine both industry standard tools along with some brand new emerging technologies. This means I have hit my goals from the start of the project. These have also given me some great exposure into the tools that make up the backend of some of the most successful, highly trafficked and popular

4 Semester One Conclusion

applications in the world. The successful migration to a more efficient event-driven architecture through the use of CDC should prove to future employers that I am not only capable of slotting in and using industry standard tools but that I am also capable of researching and using new and emerging technologies which lead to an improved system.

- My current understanding around how the Kubernetes Operator pattern is being developed and deployed makes me believe it is a game changer. The ability to automate / abstract layers of components in the system is a very powerful one. It is one I only began to realise when I noticed that the Strimzi operator deploys and manages other operators (Kafka Entity Operator). The ease at which the system can be scaled up and down is frighteningly straightforward. The learning curve is a little steep, but once one is familiar enough with the Kubernetes basics the architecture becomes a thing of beauty.
- I am also quite happy that every goal as set out in the introduction was achieved (save the consumer improvements) and that all components used are open-source. A fascinating insight in to the open-source world came when I was rather disillusioned about the Kafka Connect configurations. I reached out to the Debezium community and was answered to by none other than the lead-maintainer and co-founder of the Debezium project, Gunnar Morling. I asked what I thought was maybe a stupid simple question, but it turns out it was an issue that had caught others before. Gunnar pointed me to the Debezium community group chat where I found an article which helped me overcome the issue.

The knowledge that one can build a system that is capable of slotting into the back-end of Fortune 100 companies, from completely free and open sourced software is truly amazing. The whole area of systems architecture was one that I was intrigued about before this project. But now it excites and intrigues me even more.

- The usefulness of the project has been, anecdotally, further corroborated from answers to questions which I have posed to other publicans / business owners and management about their largest pain points and access to statistical insights from their current back-end system. The goal of these questions was to ascertain if a solution for these pain points exists from technologies already in existence. Whilst some PoS software has inventory management as an option, there is no software stack that allows the processing of invoices (more details in next section) and the complete automation of the backend system. Further questions were asked about the wants or needs for such a system and feedback has been taken into consideration during the entire thought process for the design of the system.

5 Semester Two Plan

5.1 Work to be Completed

The plan for semester two is to finish off the Category One processes and complete and implement the Category Two processes. Some preliminary research has already been done into what is needed to complete and integrate both categories of the project.

5.1.1 Kafka Consumers

Consumers need to be developed. These consumers will finish off the Category One processes needed to complete the system. Each consumer is an individual microservice and has a different responsibility:

1. The first consumer will consume the transaction data with a focus on the product (s) in the transaction. It will then save / update the inventory for the user as indicated in the transaction.
2. The second consumer will consume the transaction data with a focus on the financial figures and save / update the inventory for the user as indicated in the transaction.
3. The third consumer which is non-essential to the core system is the consumer which will focus on counting particular product consumption and alert the necessary external entities when the threshold is reached.

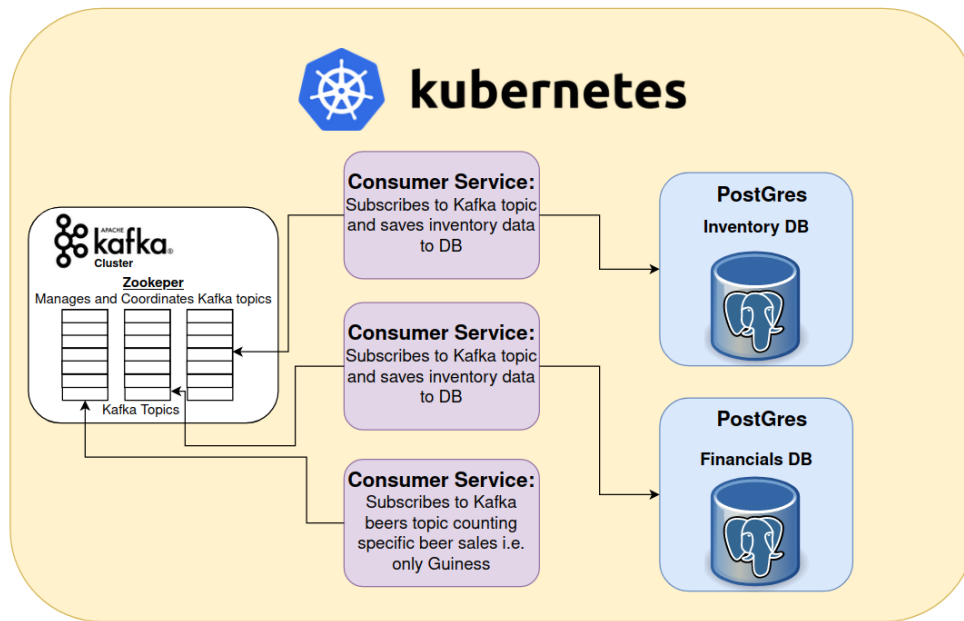


Figure 5.1: Architecture of the Kafka Consumers, saving inventory and financial data to their respective databases.

5.1.2 Category Two Processes

This sees the focus move away from systems architecture to a focus on data science and machine learning (ML). It is the section which has the most uncertainty in regards implementation. The Category Two processes, expenditure from scanned invoices / receipts, can be broken down into three distinct problems. Each problem should be accomplishable with the use of a different ML model:

1. **Text Localisation:** There is no standard when it comes to invoice templates. Each company has their own and as such a deep learning model is needed to localise and apply bounding boxes to the text in the invoice. This is a problem that is solved by using a pre-trained model and some open-source models already exist which may be used to solve this problem. The issue here is that whilst the implementation of a simple open-source model would return a decent result, this is only step one of a three-step process. The subsequent steps rely on the results of the previous steps in the pipeline, so utmost care must be taken to ensure that the results are accurate as possible.
2. **Scanned Invoice Optical Character Recognition (OCR):** This stop can be thought of as a solved problem, for the most part. Many open-source OCR technologies already exist, and one of these shall be implemented to solve this step.

3. **Key Information Extraction (KIE):** This is where, by an order of magnitude, the most uncertainty lies. This is **not** a solved problem in the area of computer science. To such an extent that there are regular competitions [43] to try and improve industry standard solutions. These competitions include teams from some top ML universities globally along with some expert companies such as Samsung and Tencent. Some open-source solutions for similar problems (simpler receipt KIE) exist, but not all of them are suitable for the problem at hand nor are feasible as some solutions require an enormous amount of compute power to implement. Initial estimates of some of the more promising solutions cost upwards of €5,000 to re-train the model. There are some simpler (less expensive) open-source models available too and these may be used as a starting point to solve this problem.

There is a further problem identified here which is the feedback loop to improve the performance of the ML model. This can be achieved currently by “boosting”, which is the application of weights to the models parameters after the model has been trained. This is quite a crude and simplistic approach which requires a lot of computation. It is also an area of computer science that is an active area of research and development where there is no clear de facto standard. This must be achieved to some level to improve the model. It is also necessary that the information is correctly categorized and that the correct fields are populated / updated in the appropriate databases.

The results from the aforementioned 2019 ICDAR Robust Reading competition [43], which is still ongoing (the newest results were published toward the end of November 2021) will be the main inspiration for the models used in this project. I am very excited about the areas of research needed to understand the implementation of this part of the project.

5.1.3 Category Two Architecture

The current architecture of this section is to deploy the ML models to AWS SageMaker or Google Cloud Platform (GCP) AI Platform. A lambda function will be utilized to send the new invoices, ideally obtained from a specific email inbox, to the appropriate ML model. The results of the model will be sent back into the system to populate / update the appropriate databases via an API. This serverless architecture will mean that scaling up the system or increasing the underlying hardware to enable quicker processing should be possible with minimal effort.

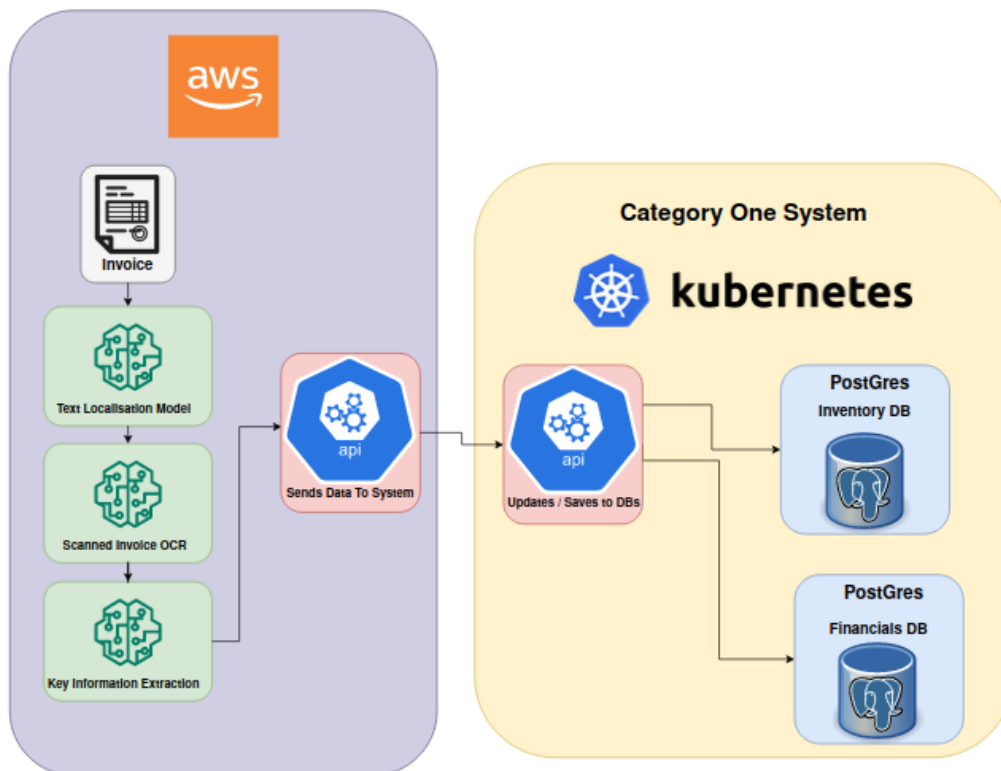


Figure 5.2: Proposed architecture of Category Two system and how it integrates with the Category One architecture.

5 Semester Two Plan

There are some further unknowns in regards the need for two APIs as is detailed by the architectural diagram. I have not deployed an ML pipeline yet and as such the exact methodology that will be used to integrate the two systems is liable to change. A further point of interest is the potential need for a lock on the inventory and financial databases once they are being updated by the consumers or by the ML pipeline APIs to ensure data consistency.

| Risk Register | |
|----------------------------------|------------|
| Component | Risk 1 - 5 |
| Text Localisation Model | 2 |
| Scanned Invoice OCR Model | 1 |
| Key Information Extraction Model | 5 |
| Consumers | 2 |
| APIs for System integration | 3 |

Figure 5.3: Risk Register of components left to complete.

5.1.4 Future Work Plan

Once again ClickUp will be the Primary tool for tracking progress. Included, even though clarity is not optimal, is the Gantt Chart for semester two as provided by ClickUp. The chart starts at January 17th and continues until the end of the semester.

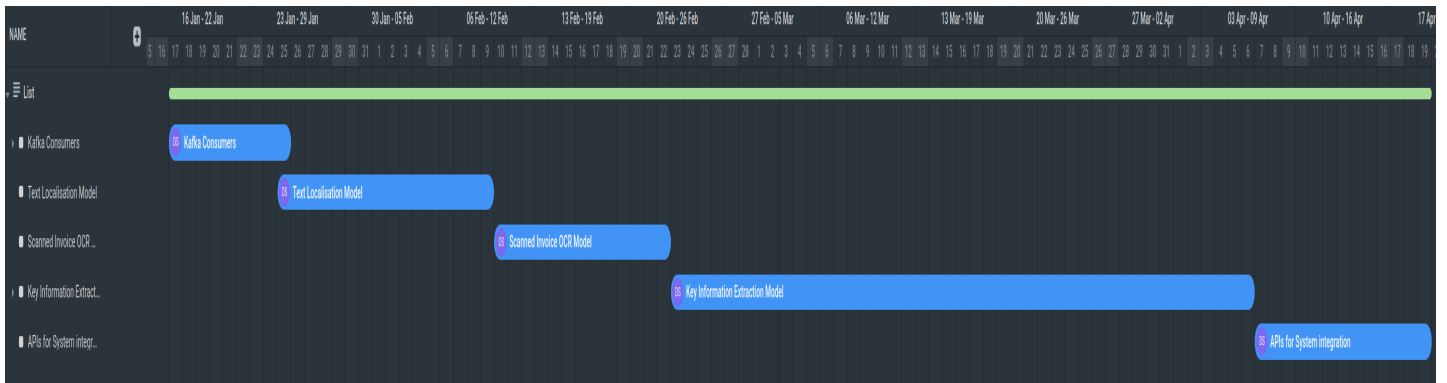


Figure 5.4: Gantt Chart for semester two.

6 Supporting Material

6.1 Manifests

6.1.1 API Deployment Manifest

```
io.strimzi.kafka.v1beta2.KafkaConnect (v1beta2@kafkaconnect.json)
1  apiVersion: kafka.strimzi.io/v1beta2
2  kind: KafkaConnect
3  metadata:
4    name: my-connect-cluster
5    annotations:
6      strimzi.io/use-connector-resources: "true"
7  spec:
8    image: dimakis/strimzi-kafka-connect:latest
9    replicas: 1
10   bootstrapServers: kafka-cluster-kafka-bootstrap:9092
11   # tls:
12   #   trustedCertificates:
13   #     - secretName: my-cluster-cluster-ca-cert
14   #       certificate: ca.crt
15   config:
16     config.storage.replication.factor: 1
17     offset.storage.replication.factor: 1
18     status.storage.replication.factor: 1
19     config.providers: file
20     config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
21   externalConfiguration:
22     volumes:
23     - name: connector-config
24       secret:
25         secretName: postgres-secret
26
```

Figure 6.1: Kafka Connect Custom Resource Definition manifest.

6.1.2 Docker-compose

```

1 version: '3'
  You, 3 weeks ago | 1 author (You)
2 services:
  You, a month ago | 1 author (You)
3   app:
4     container_name: full_app
5     build: .
6     ports:
7       - 8080:8080
8     restart: on-failure
9     volumes:
10      - api:/usr/src/app/
11     depends_on:
12      - moneykey-t-api-postgres      # Uncomment this when using postgres.
13     networks:
14      - fullstack
15
16   moneykey-t-api-postgres:
17     image: dimakis/debezigres:latest
18     container_name: full_db_postgres
19     environment:
20       - POSTGRES_USER=${DB_USER}
21       - POSTGRES_PASSWORD=${DB_PASSWORD}
22       - POSTGRES_DB=${DB_NAME}
23       - DATABASE_HOST=${DB_HOST}
24     ports:
25       - '5432:5432'
26     volumes:
27       - database_postgres:/var/lib/postgresql/data
28     networks:
29       - fullstack
30
31   pgadmin:
32     image: dpape/pgadmin4
33     container_name: pgadmin_container
34     You, a month ago | 1 author (You)
35     environment:
36       PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL}
37       PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD}
38     depends_on:
39       - moneykey-t-api-postgres
40     ports:
41       - "5050:80"
42     networks:
43       - fullstack
44     restart: unless-stopped

```

Figure 6.2: docker-compose used in the initial testing of the API and PostGres deployment.

Bibliography

- [1] *A Gentle Introduction to Batch Normalization for Deep Neural Networks*.
<https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/>.
- [2] *Apache Kafka*. <https://kafka.apache.org/documentation/>.
- [3] *ClickUp™ — One App to Replace Them All*. <https://clickup.com>.
- [4] *Concepts*. <https://kubernetes.io/docs/concepts/>.
- [5] *Connectors :: Debezium Documentation*.
<https://debezium.io/documentation/reference/stable/connectors/index.html>.
- [6] *DD5D5V08elb*. <https://link.medium.com/DD5D5V08elb>.
- [7] *Debezium Architecture :: Debezium Documentation*.
<https://debezium.io/documentation/reference/architecture.html>.
- [8] Debezium Community. *Debezium*. <https://debezium.io/>.
- [9] Loris Degioanni. *3 Phases of Prometheus Adoption*.
<https://www.infoworld.com/article/3275887/3-phases-of-prometheus-adoption.html>. May 2018.
- [10] *Deploying a Containerized API on Kubernetes — by Victor Steven — Level Up Coding*. <https://levelup.gitconnected.com/deploying-dockerized-golang-api-on-kubernetes-with-postgresql-mysql-d190e27ac09f>.
- [11] *Docker-Compose and Create Db in Postgres on Init*.
<https://stackoverflow.com/questions/59715622/docker-compose-and-create-db-in-postgres-on-init>.
- [12] *Docker-Images/Postgres/13 at Main · Debezium/Docker-Images*.
<https://github.com/debezium/docker-images>.
- [13] *Docker-Images/Postgres/9.6 at Main · Debezium/Docker-Images*.
<https://github.com/debezium/docker-images>.
- [14] *Event-Driven Architecture and Microservices Combination Concerns*. July 2020.
- [15] *GOCDC and Postgres*. <https://dev.to/thiagosilvaf/gocdc-and-postgres-1m4m>.
- [16] *Gorilla/Mux*. Gorilla Web Toolkit. Oct. 2021.
- [17] *Grafana: The Open Observability Platform*. <https://grafana.com/>.
- [18] *Helm*. <https://helm.sh/>.

Bibliography

- [19] *How 8 Giant Companies Use Kubernetes & 60 Others That Use It.*
<https://www.containiq.com/post/companies-using-kubernetes>.
- [20] *How to Integrate Kafka with Istio on OpenShift.*
https://labs.consol.de/development/2021/02/02/istio_and_kafka_on_openshift.html.
- [21] *How to Plot Train and Validation Accuracy Graph?*
<https://discuss.pytorch.org/t/how-to-plot-train-and-validation-accuracy-graph/105524>.
- [22] *How to Use Change Data Capture (CDC) with Postgres.*
<https://dev.to/thiagosilva/how-to-use-change-database-capture-cdc-in-postgres-37b8>.
- [23] *Intelligent Document Processing with AI.*
<https://nanonets.com/blog/receipt-ocr/%23receipt-digitization-using-tesseract>.
- [24] Sergei Issaev. *Beginner's Guide to Loading Image Data with PyTorch.*
<https://towardsdatascience.com/beginners-guide-to-loading-image-data-with-pytorch-289c60b7afec>. Dec. 2020.
- [25] *Kafka Connect — Confluent Documentation.*
<https://docs.confluent.io/platform/current/connect/index.html>.
- [26] Soham Kamani. *Implementing a Kafka Producer and Consumer In Golang (With Full Examples) For Production.*
<https://www.sohamkamani.com/golang/working-with-kafka/>.
- [27] *Kubernetes Adoption Trends Report.*
<https://www.cockroachlabs.com/guides/kubernetes-trends/>.
- [28] *Kubernetes Operator — Stateful Kubernetes Application.*
<https://k21academy.com/docker-kubernetes/kubernetes-operator/>. May 2021.
- [29] *Kubernetes Security Best Practices: 10 Steps to Securing K8s.*
<https://www.aquasec.com/cloud-native-academy/kubernetes-in-production/kubernetes-security-best-practices-10-steps-to-securing-k8s/>.
- [30] John Kutay. *Change Data Capture (CDC): What It Is and How It Works.* May 2021.
- [31] *Lessons Learned from Running Debezium with PostgreSQL on Amazon RDS.*
<https://debezium.io/blog/2020/02/25/lessons-learned-running-debezium-with-postgresql-on-rds/>.
- [32] *Logical Decoding Output Plug-in Installation for PostgreSQL :: Debezium Documentation.*
<https://debezium.io/documentation/reference/postgres-plugins.html>.
- [33] *Minikube Start.* <https://minikube.sigs.k8s.io/docs/start/>.
- [34] Naincyjain. *Effect of Batch Size on Training Process and Results by Gradient Accumulation.* <https://medium.com/analytics-vidhya/effect-of-batch-size-on-training-process-and-results-by-gradient-accumulation-e7252ee2cb3f>.

Bibliography

- [35] Pinku Deb Nath. *Graceful Shutdown of Golang Servers Using Context and OS Signals*. May 2019.
- [36] *Operator Pattern*. <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>.
- [37] *Part 5: Handling Migrations With Gorm in Go — Code Sahara*. <https://codesahara.com/blog/making-migrations-with-gorm-in-go/>.
- [38] *Pod — Kubernetes Engine Documentation — Google Cloud*. <https://cloud.google.com/kubernetes-engine/docs/concepts/pod>.
- [39] *PostgreSQL: Linux Downloads (Ubuntu)*. <https://www.postgresql.org/download/linux/ubuntu/>.
- [40] *Production-Grade Container Orchestration*. <https://kubernetes.io/>.
- [41] Prometheus. *Prometheus - Monitoring System & Time Series Database*. <https://prometheus.io/>.
- [42] *Releases · Strimzi/Strimzi-Kafka-Operator*. <https://github.com/strimzi/strimzi-kafka-operator/releases>.
- [43] *Results - ICDAR 2019 Robust Reading Challenge on Scanned Receipts OCR and Information Extraction - Robust Reading Competition*. <https://rrc.cvc.uab.es/?ch=13&com=evaluation&task=1>.
- [44] Sub says. *Kubernetes - Desired State and Control Loops*. Sept. 2019.
- [45] *Secrets*. <https://kubernetes.io/docs/concepts/configuration/secret/>.
- [46] Christos Sotiriou. *Fault Tolerance in Kubernetes Clusters*. Apr. 2020.
- [47] *Square: Solutions & Tools to Grow Your Business*. <https://squareup.com/ie/en>.
- [48] *Strimzi Documentation (0.16.2)*. <https://strimzi.io/docs/0.16.2/>.
- [49] *Strimzi Quick Start Guide (0.26.0)*. <https://strimzi.io/docs/operators/latest/quickstart.html>.
- [50] *The 7 Best CDC Tools (Change Data Capture) - Learn — Hevo*. <https://hevo.com/learn/7-best-cdc-tools/>.
- [51] *Tips & Tricks for Running Strimzi with Kubectl*. <https://strimzi.io/blog/2020/07/22/tips-and-tricks-for-running-strimzi-with-kubectl/>.
- [52] *Understanding Backpropagation in a Neural Network - 1*. <https://www.linkedin.com/pulse/understanding-backpropagation-neural-network-1-srikanth-machiraju/>.
- [53] *Using Helm*. https://helm.sh/docs/intro/using_helm/.
- [54] *Using Strimzi*. <https://strimzi.io/docs/operators/latest/full/using.html>.
- [55] Ivan Velichko. *How to Grasp Containers - Efficient Learning Path - Ivan Velichko*. <https://iximiuz.com/en/posts/container-learning-path/>.

Bibliography

- [56] *What Is A Helm Chart? – A Beginner’s Guide.* Aug. 2019.
- [57] *What Is a Kubernetes Operator?*
<https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator>.
- [58] *What Is a Makefile and How Does It Work?*
<https://opensource.com/article/18/8/what-how-makefile>.
- [59] *What Is Event-Driven Architecture?*
<https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture>.
- [60] *What Is Zookeeper and Why Is It Needed for Apache Kafka? - CloudKarafka, Apache Kafka Message Streaming as a Service.*
<https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html>.
- [61] *Who’s Using Debezium?* <https://debezium.io/community/users/>.