Waterford Institute *of* Technology

Automation of Bar/ Pub Back-end Business Processes including Inventory Management, Accounting and Tax Compliance

Dimitri Saridakis

Work submitted
within the
BSc in Applied Computing
Cloud & Networks

Supervisor: Dr. Kieran Murphy
Second Reader: Clodagh Power

December 29, 2021

Contents

List of Figures

1 Project Goals and Technologies

1.1 Introduction

This report's structure will follow this style:

- First: The report will outline the project's goals, the system architecture and the technologies used.
- Second: The report will detail the implementation of the varies system components from the start to the working prototype.
- Third: The report will outline the challenges faced in the implementation of the system.
- Fourth: A conclusion will be given including both technical and personal reflection along with detailed analysis on proposed work to be completed in semester two.

1.1.1 Motivation

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.
The motivation for this project comes from the major pain points of mine, from my previous 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 follows 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. I'm defining reward 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 I will refer to as '*group A*' activities), whilst these processes are critical to a business' operation and regulatory compliance - 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 I will refer to as '*group B*' activities) are the catalysts which drive sales and business growth.

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 or 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. Furthermore, the implementation of this system should increase the quality of life of business owners who no longer have to carry out manual data entry and monotonous The aim is to provide more time for group B growth activities and processes by automating the group A processes in this project. For this to be accomplished there is a list of processes that need to be tackled.
These processes fall into two broad categories, namely:

- The first category comes from the data collected from a sale of a users' product(a pub selling a beverage)
- The second stems from the data collected from a users purchases in relation to inventory.

These include:

- Keeping a record of all sale transactions that enter the system which will allow for the processing of sales, this is done by saving each transaction's data to it's own database
- The subsequent saving and updating of the transaction sales figures i.e. gross, net figures to their own database
- The updating of the inventory levels of products per sale
- The scanning of supplier invoices and key information extraction from the invoices
  - Updating of inventory levels as stock is delivered/ invoiced
  - Updating of cash flow levels

This is quite a lengthy list of processes to implement. It can be broken into two main components.
For this FYP I will be simulating the Point of Sale(PoS) till software with fake data as I do not have time to build a user facing PoS system. If time constraints allow, I would like to incorporate transactional data from a current free and popular PoS application. This will most likely be SquareUp PoS by Square as my business uses this PoS currently and there is a developer API which can potentially be leveraged to provide the transactional data into the system. This is more of a "want" currently.
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 users of the system. Some of these include:
The ability to do some exploratory data analysis and other ML activities which can provide the businesses that use the system with some new data driven insights about the 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 and questions I have posed to other business owners/ management.
An example of some more insights that can be derived from the information in the system, given decent levels of adoption in the industry, are live sales per product. Having a multitude of different businesses using the system, the sales of specific items can be accessed in real time. This provides some invaluable data to brewers of sales which can be utilised to precisely schedule production times and quantities.
Further motivation The system I am building is an amalgamation of numerous different concepts and modules that I have studied over the past 4 years. I am using this project as a way to try and implement some of the architectures, processes and technologies that are industry standard and at the cutting edge of innovation in the tech industry. These include: (maybe go into detail on each one here) Event-driven Micro-service architecture Kubernetes Apache Kafka Debezium Change data capture Deep learning Neural Networks

The motivation for this research paper is to explore some technologies that would build a cloud native build a *cloud native* system that is highly scalable and as efficient as possible with as much of the processes automated as can be done.
A system that is capable of reacting - in real time - to changes i.e. any of the CRUD operations in a database.
Upon these change events occurring, the changes to the data in the system which triggered the events should be made available to other services/ components of the system to allow for the business logic, processing of data and subsequently business goals of the system to be met. To aid with accomplishing these requirements the system design is of an event-driven, microservice architectured nature.

1.1.2 High Level Architecture

Debezium will be looking after the change events from a PostGres database, interacting with and utilizing Apache Kafka (*Apache Kafka* n.d.) for the event streams. The deployment orchestration is managed by Kubernetes, in this case I'll be using a single Kubernetes node on my machine by utilizing Minikube(*Minikube Start* n.d.). These are the primary technologies around which this term paper will focus.

1.1.3 System Goals

The goals for this system are to:

- Utilize an API which exposes an endpoint external to the Kubernetes cluster.
- Once that endpoint is hit, the API creates some data which is saved to a database.
- Configure the Debezium PostGres connector to listen for changes in the state of the database. Debezium will then utilize Kafka Connect to ingest the stream updates, those updates are then pushed to Kafka Topics.
- Create a consumer which is subscribed to a Kafka Topic to consume the changes and prove the system is operating nominally.
- Automate as much of the processes as possible.
- Focus on the use of open source technologies.

1.1.4 System Architecture

The following is the system architecture diagram:

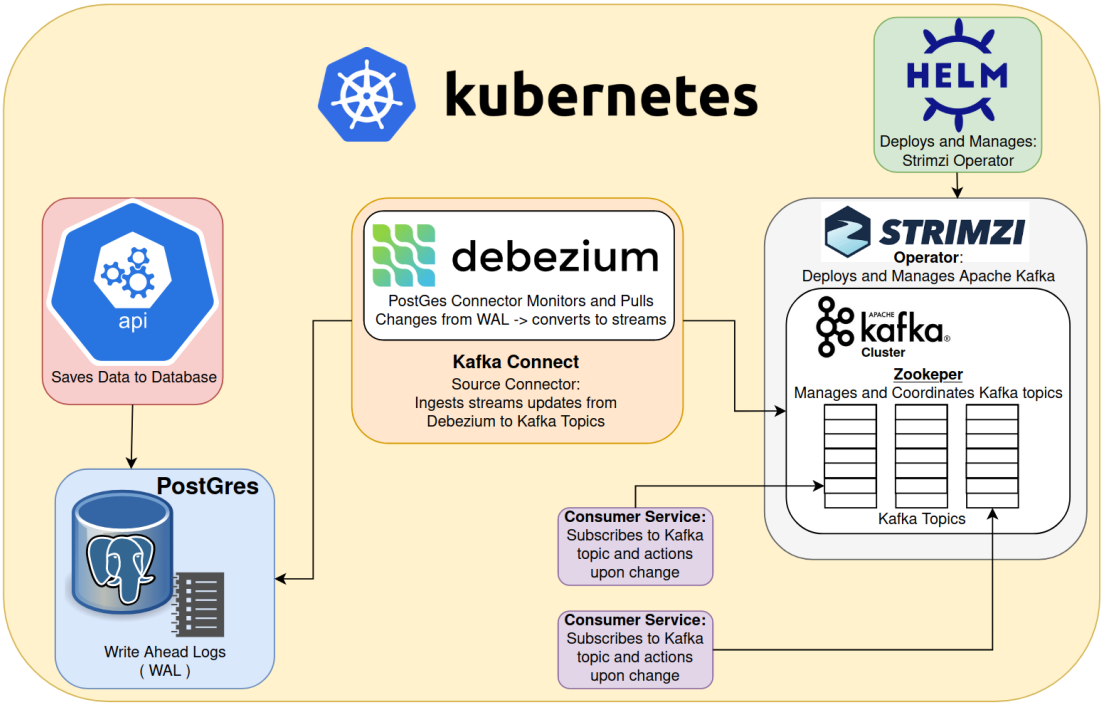

Figure 1.1: Architecture of the system depicting the interaction between components.

Note: The system architecture has a section 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.

1.2 Technologies Used

1.2.1 Kubernetes

Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling, and management. It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation(CNCF) (*Production-Grade Container Orchestration* n.d.). A whole paper could be written about Kubernetes, but for the sake of brevity this report will only explain Kubernetes components explicitly relevant to this project. This report 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.
Kubernetes' services are deployed in a cluster via a deployment manifest. The deployment manifest is either a JSON file or a YAML file which describes the desired state of the service. The Kubernetes control loop 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 restart the service from the deployment manifest.

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.
This paper will also use **Helm** which is Kubernetes' package manager(*Using Helm* n.d.).

1.2.2 API for System Data

Since this report will focus on capturing changes from a database, there will need to be a methodology for creation and saving the data. This is handled by an API in current

development. It is essentially a Transaction API, it exposes an endpoint and which creates a fake transaction. 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 randomized information for things like name, ABV, price etc. The API is written in GoLang and uses Gorilla Mux(a high performance HTTP router package).

1.2.3 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 (*Connectors :: Debezium Documentation* n.d.).
Debezium is most commonly deployed via Apache Kafka's Kafka Connect framework (*Kafka Connect — Confluent Documentation* n.d.). 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 (Debezium Community n.d.). 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 (*Debezium Architecture :: Debezium Documentation* n.d.). 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 some 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' primary job becomes servicing the incoming requests from the API.

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

1.2.4 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.
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 (*Apache Kafka* n.d.), 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.

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 (*What Is Zookeeper and Why Is It Needed for Apache Kafka?* N.d.).

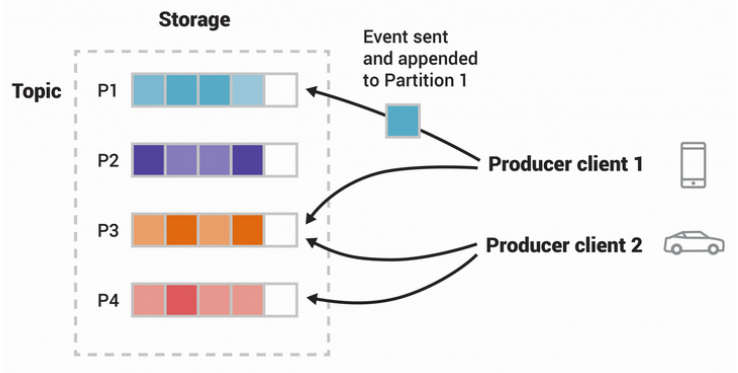

Figure 1.2: Simplified view of a partitioned topic being written to by multiple producers(*Apache Kafka* n.d.). In this report there will be just one producer, the Debezium Kafka Connect instance.

1.2.5 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 (*What Is a Kubernetes Operator?* N.d.). 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 (*Operator Pattern* n.d.). 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.

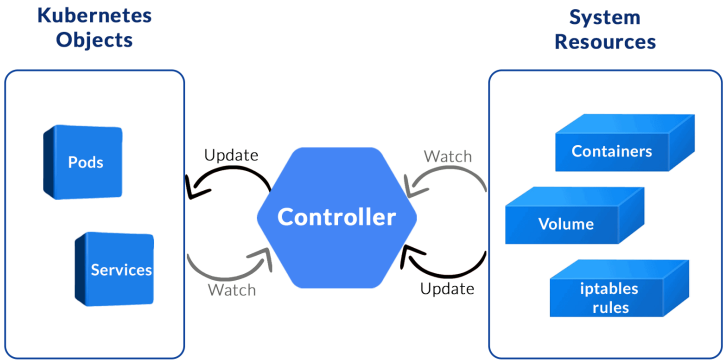

Figure 1.3: Operator control loop based on the Kubernetes *Observe, Act, Analyze* control loop. Sourced from (*Kubernetes Operator — Stateful Kubernetes Application* 2021).

2 Implementation

2.1 Kubernetes Ready

Minikube is started by running `$ minikube start`.

2.1.1 API Containerization

As Kubernetes deploys containers as services, the Transactions API is first containerized.
The Transaction API container is built based on a Dockerfile found here (**stevenDeployingDockerizedGolang2019**) 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.

```
# Start from golang base image
FROM golang:alpine as builder
# Add Maintainer info
LABEL maintainer="Dimitri Saridaksi <dimitri.saridakis@gmail.com>"

# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache git

# Set the current working directory inside the container
WORKDIR /app

# Copy go mod and sum files
COPY go.mod go.sum ./

# Download all dependencies. Dependencies will be cached if the go.mod and the go.sum files are not changed
RUN go mod download

# Copy the source from the current directory to the working Directory inside the container
COPY . .

# Build the Go app
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

# Start a new stage from scratch
FROM alpine:latest
RUN apk --no-cache add ca-certificates

WORKDIR /root/

# Copy the Pre-built binary file from the previous stage. Observe we also copied the .env file
COPY --from=builder /app/main .
COPY --from=builder /app/.env .

# Expose port 8080 to the outside world
EXPOSE 8080

#Command to run the executable
# CMD ["./main"]
ENTRYPOINT [ "./main", "serve" ]
```

Figure 2.1: 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 error/ important detail being documented from the exact line of code were the error/ event occurred along with the error/ event message/ details 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.

2.1.2 PostGres Containerization

This section was 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 (*Logical Decoding Output Plug-in Installation for PostgreSQL :: Debezium Documentation* n.d.) There are two ways to do this.

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

Luckily, there is a far more straight forward method. The Debezium project has a repo on their GitHub (*Docker-Images/Postgres/13 at Main · Debezium/Docker-Images* n.d.) 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 *Fig. 2.2*.

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 *Fig. 2.3*. 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›─$ 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 2.2: Running the build script to generate the new Debezium friendly PostGres docker image

```
┌─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 2.3: The initial and re-tagged PostGres friendly image.

2.2 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 (*Pod — Kubernetes Engine Documentation — Google Cloud* n.d.). 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.

2.2.1 Automated Deployment

In keeping with one of the reports goals, 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 2.4: The deployment Makefile

2.2.2 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 (*Secrets* n.d.) as per *Fig. 2.5*.

```
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 2.5: The Kubernetes Secret resource to be applied to the cluster with the real values altered.

```
┌─dimdakis@dimdakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing 1 - Term Paper
└‹base›─$ kube get secrets
NAME                                                  TYPE                                  DATA   AGE
default-token-qmvw5                                   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-xhnpn                       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
┌─dimdakis@dimdakis-xps in ~/year4/semesterOne/Cloud_Computing_I/term-paper/Cloud Computing 1 - Term Paper
```

Figure 2.6: 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.

2.2.3 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.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv-volume
  labels:
    type: local
    app: moneykey-t-api-postgres
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
  persistentVolumeReclaimPolicy: Retain
```

Figure 2.7: Manifest provisioning a persistent volume of size 5 GB which will be used for the PostGres database.

2.2.4 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.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pv-claim
  labels:
    app: moneykey-t-api-postgres
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Figure 2.8: Manifest binding the persistent volume to the PostGres database.

2.2.5 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.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: moneykey-t-api-postgres
  labels:
    app: moneykey-t-api-postgres
spec:
  selector:
    matchLabels:
      app: moneykey-t-api-postgres
      tier: postgres
  replicas: 1
  template:
    metadata:
      labels:
        app: moneykey-t-api-postgres
        tier: postgres
    spec:
      containers:
        - image: debezium/postgres:latest
          name: postgres
          imagePullPolicy: "IfNotPresent"
          envFrom:
            - secretRef:
                name: postgres-secret
          ports:
            - containerPort: 5432
              name: postgres
          volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-persistent-storage
      volumes:
        - name: postgres-persistent-storage
          persistentVolumeClaim:
            claimName: postgres-pv-claim
```

Figure 2.9: 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 require this warning will be ignored.

2.2.6 PostGres Service

The last step to enable the database to run nominally is the service. 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.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: moneykey-t-api-postgres
  labels:
    app: moneykey-t-api-postgres
spec:
  type: NodePort
  ports:
    - port: 5432
  selector:
    app: moneykey-t-api-postgres
    tier: postgres
```

Figure 2.10: PostGres service manifest.

2.2.7 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.

2.2.8 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:

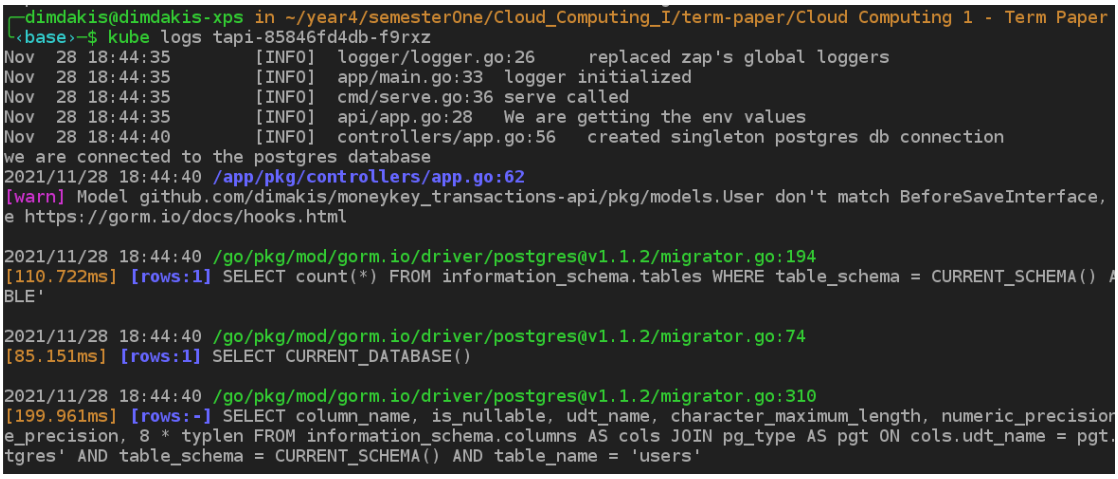

Figure 2.11: Logs for the API container displays migrations to the database. This shows the two containers are communicating and behaving nominally.

2.2.9 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. It is simply deployed using this command `helm install strimzi/strimzi-kafka-operator -n default` where the `-n` flag specifies the namespace to deploy to.

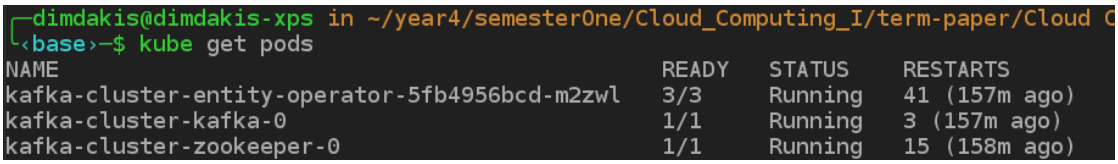

Figure 2.12: Helm successfully deploys the Strimzi Kafka Operator which in turn deploys the Kafka components.

2.2.10 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)
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name:  my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  image: dimakis/strimzi-kafka-connect:latest
  replicas: 1
  bootstrapServers: kafka-cluster-kafka-bootstrap:9092
  # tls:
  #   trustedCertificates:
  #     - secretName: my-cluster-cluster-ca-cert
  #       certificate: ca.crt
  config:
    config.storage.replication.factor: 1
    offset.storage.replication.factor: 1
    status.storage.replication.factor: 1
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
  externalConfiguration:
    volumes:
      - name: connector-config
        secret:
          secretName: postgres-secret
```

Figure 2.13: 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 didn't out weight 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.

2.2.11 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)

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: transactions-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.postgresql.PostgresConnector
  tasksMax: 1
  config:
    database.hostname: moneykey-t-api-postgres
    database.port: 5432
    database.user: dimakis
    database.password: password
    database.server.id: 18405454
    database.server.name: moneykey-t-api-postgres
    database.dbname: moneykey-t-api-postgres
    database.whitelist: moneykey-t-api-postgres
    database.history.kafka.bootstrap.servers: kafka-cluster-kafka-bootstrap:9092
    database.history.kafka.topic: schema-changes.moneykey-t-api-postgres
    key.converter: "org.apache.kafka.connect.json.JsonConverter"
    include.schema.changes: "true"
    plugin.name: wal2json
    key.converter.schemas.enable: "true"
    value.converter.schemas.enable: "true"
```

Figure 2.14: 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.

2.3 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.

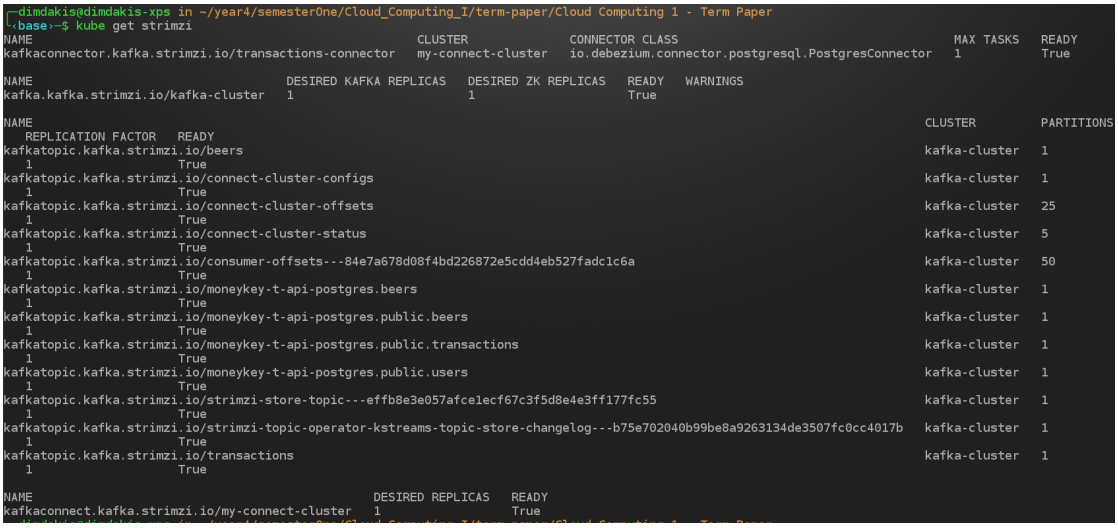

Figure 2.15: Kafka cluster information managed by the Kafka Entity Operator. This is the operator that manages both topics and users and the operator itself is deployed and managed by the Strimzi operator.

As can be seen from *Fig. 2.15* above, 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:

```
kubectl 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, Kubectl 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:

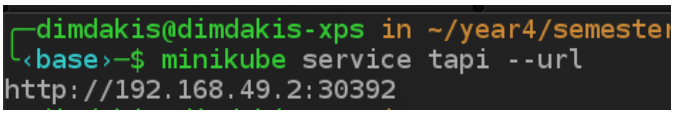

Figure 2.16: The API's exposed URL via the service that was configured previously.

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 "Ruination IPA".

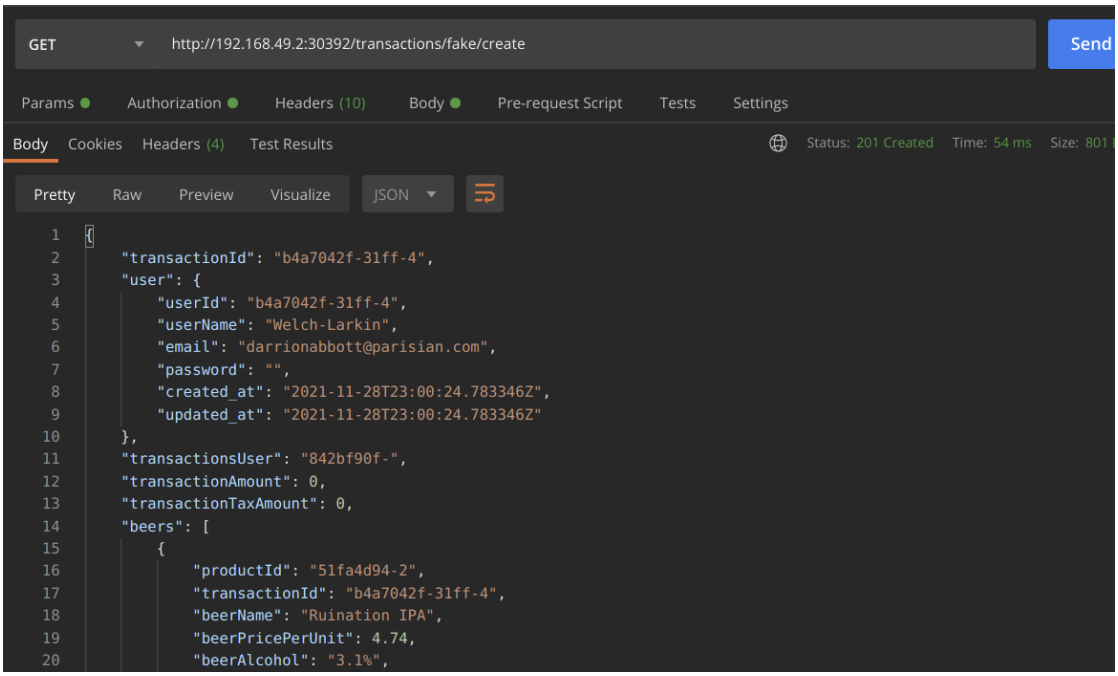

Figure 2.17: Testing the endpoint with Postman. The response can be observed.

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:

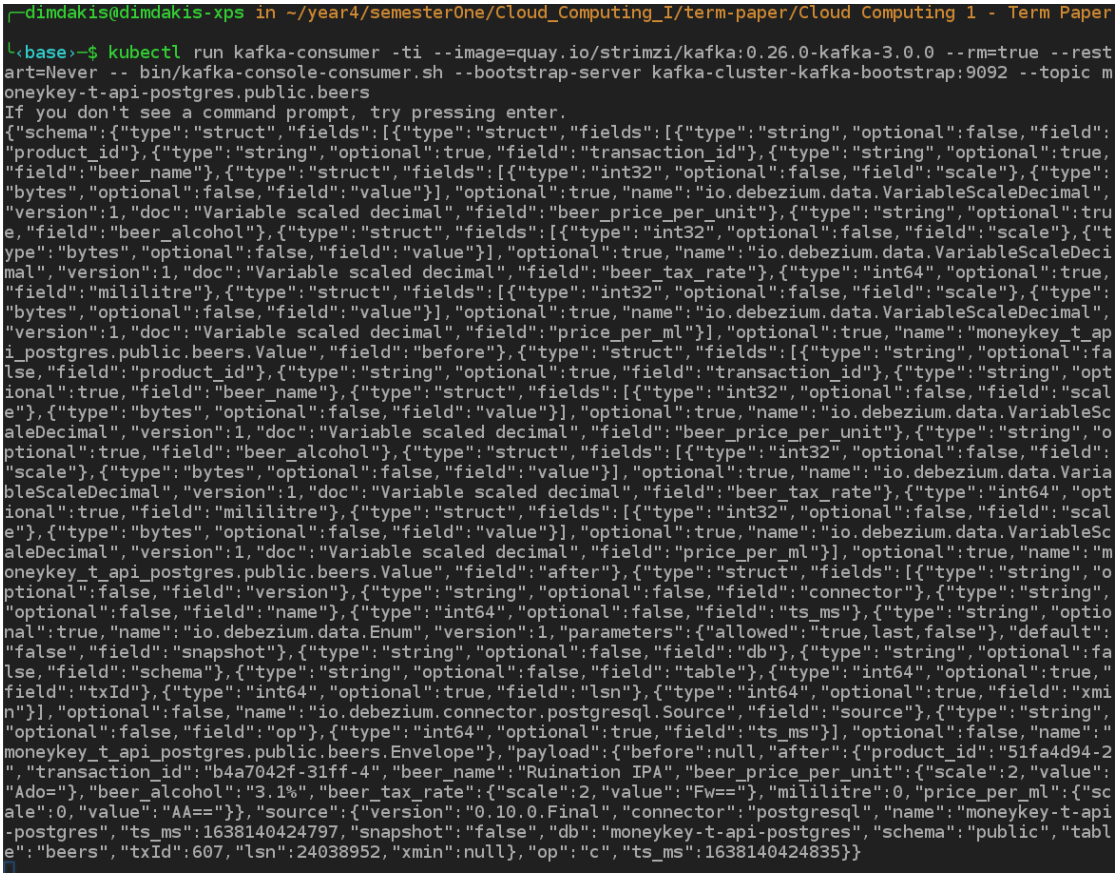

Figure 2.18: The data is streamed to the consumer via Debezium and the Kafka components.

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.

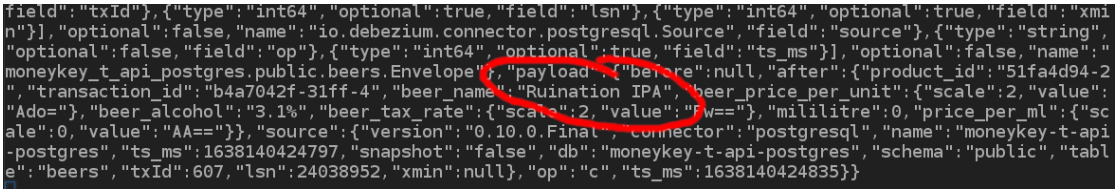

Figure 2.19: Proof of the newly created beer in the Kafka `beers` topic

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.

2.4 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 creation of the PostGres container that is configured went through many iterations. Debezium requires certain 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 (*Logical Decoding Output Plug-in Installation for PostgreSQL :: Debezium Documentation* n.d.).

- Issues were had with docker-compose during the testing phase that required a purge of the docker-compose network. This wasn't a trivial fix either.

- The installation of Kafka via the Strimzi operator was a desperate last chance pivot. Initially Kafka was deployed 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.

- 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 largest chunk of time was probably spent on the configuration of the Kafka Connect cluster and the addition of the Debezium Kafka Connector resource. 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 1 value was left out of the configuration manifest. The seemingly obvious `database.server.name: moneykey-t-api-postgres` was omitted for far too long.

3 Conclusion and Outlook

3.1 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 system. The use of Prometheus and Grafana would be a good way to monitor and find ways to improve performance.

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 retires/ back-offs are required as a result.

A consumer with some basic functionality in processing the stream data would be a great addition here.

As to would the addition of TLS between each component. Maybe this could be with the incorporation of a Service Mesh into the architecture. This was actually looked into and one of the reasons for not using Strimzi from the start was its seemingly incompatibility with Istio, the service mesh with which I have the most experience. If Istio was to be incorporated that would enable the easy automation of Prometheus, Grafana and a whole host of other metric gathering tools which come natively with an Istio deployment.

3.2 Personal Thoughts

I could have written so much more. I knew that the system is a big one to implement and given less time and word constraints I would have liked to explore this area much more.
I am actually quite proud of the system that was created. These components(scaled up versions) are used in industry for the deployment of some of the largest and highly trafficked systems in the world. It was a sensational learning experience putting it all together.

The amount of knowledge I have gained about Kubernetes and microservice architectures is much, much more than I thought I would learn at the outset of this report, and it will surely stand by me as I, hopefully, make my way into industry.
My current understanding around the operator pattern is making 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 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 I was intrigued about before this report. But now I'm even more intrigued now.

4 Supporting Material

4.1 Manifests

4.1.1 API Deployment Manifest

io.strimzi.kafka.v1beta2.KafkaConnect (v1beta2@kafkaconnect.json)

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name:  my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  image: dimakis/strimzi-kafka-connect:latest
  replicas: 1
  bootstrapServers: kafka-cluster-kafka-bootstrap:9092
  # tls:
  #   trustedCertificates:
  #     - secretName: my-cluster-cluster-ca-cert
  #       certificate: ca.crt
  config:
    config.storage.replication.factor: 1
    offset.storage.replication.factor: 1
    status.storage.replication.factor: 1
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
  externalConfiguration:
    volumes:
      - name: connector-config
        secret:
          secretName: postgres-secret
```

Figure 4.1: Kafka Connect Custom Resource Definition manifest.

4.1.2 Docker-compose

```yaml
version: '3'
You, 3 weeks ago | 1 author (You)
services:
  You, a month ago | 1 author (You)
  app:
    container_name: full_app
    build: .
    ports:
      - 8080:8080
    restart: on-failure
    volumes:
      - api:/usr/src/app/
    depends_on:
      - moneykey-t-api-postgres          # Uncomment this when using postgres.
    networks:
      - fullstack

  You, 3 weeks ago | 1 author (You)
  moneykey-t-api-postgres:
    image: dimakis/debezigres:latest
    container_name: full_db_postgres
    environment:
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASSWORD=${DB_PASSWORD}
      - POSTGRES_DB=${DB_NAME}
      - DATABASE_HOST=${DB_HOST}
    ports:
      - '5432:5432'
    volumes:
      - database_postgres:/var/lib/postgresql/data
    networks:
      - fullstack

  You, a month ago | 1 author (You)
  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin_container
    You, a month ago | 1 author (You)
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD}
    depends_on:
      - moneykey-t-api-postgres
    ports:
      - "5050:80"
    networks:
      - fullstack
    restart: unless-stopped
```

Figure 4.2: docker-compose used in the initial testing of the API and PostGres deployment.

Bibliography

[1] *Apache Kafka*. https://kafka.apache.org/documentation/.

[2] *Apache Kafka*. https://kafka.apache.org/.

[3] *Connectors :: Debezium Documentation*. https://debezium.io/documentation/reference/stable/connectors/index.html.

[4] *DD5D5V08elb*. https://link.medium.com/DD5D5V08elb.

[5] *Debezium Architecture :: Debezium Documentation*. https://debezium.io/documentation/reference/architecture.html.

[6] Debezium Community. *Debezium*. https://debezium.io/.

[7] *Docker-Compose and Create Db in Postgres on Init*. https://stackoverflow.com/questions/59715622/docker-compose-and-create-db-in-postgres-on-init.

[8] *Docker-Images/Postgres/13 at Main · Debezium/Docker-Images*. https://github.com/debezium/docker-images.

[9] *Docker-Images/Postgres/9.6 at Main · Debezium/Docker-Images*. https://github.com/debezium/docker-images.

[10] *GOCDC and Postgres*. https://dev.to/thiagosilvaf/gocdc-and-postgres-1m4m.

[11] *Gorilla/Mux*. Gorilla Web Toolkit. Oct. 2021.

[12] *Helm*. https://helm.sh/.

[13] *How to Integrate Kafka with Istio on OpenShift*. https://labs.consol.de/development/2021/02/02/istio_and_kafka_on_openshift.html.

[14] *How to Use Change Data Capture (CDC) with Postgres*. https://dev.to/thiagosilvaf/how-to-use-change-database-capture-cdc-in-postgres-37b8.

[15] *Kafka Connect — Confluent Documentation*. https://docs.confluent.io/platform/current/connect/index.html.

[16] Soham Kamani. *Implementing a Kafka Producer and Consumer In Golang (With Full Examples) For Production*. https://www.sohamkamani.com/golang/working-with-kafka/.

[17] *Kubernetes Operator — Stateful Kubernetes Application*. https://k21academy.com/docker-kubernetes/kubernetes-operator/. May 2021.

[18] *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/.

[19] *Logical Decoding Output Plug-in Installation for PostgreSQL :: Debezium Documentation.* https://debezium.io/documentation/reference/postgres-plugins.html.

[20] *Minikube Start.* https://minikube.sigs.k8s.io/docs/start/.

[21] *Operator Pattern.* https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[22] *Pod — Kubernetes Engine Documentation — Google Cloud.* https://cloud.google.com/kubernetes-engine/docs/concepts/pod.

[23] *PostgreSQL: Linux Downloads (Ubuntu).* https://www.postgresql.org/download/linux/ubuntu/.

[24] *Production-Grade Container Orchestration.* https://kubernetes.io/.

[25] *Secrets.* https://kubernetes.io/docs/concepts/configuration/secret/.

[26] *Strimzi Documentation (0.16.2).* https://strimzi.io/docs/0.16.2/.

[27] *Strimzi Quick Start Guide (0.26.0).* https://strimzi.io/docs/operators/latest/quickstart.html.

[28] *The 7 Best CDC Tools (Change Data Capture) - Learn — Hevo.* https://hevodata.com/learn/7-best-cdc-tools/.

[29] *Tips & Tricks for Running Strimzi with Kubectl.* https://strimzi.io/blog/2020/07/22/tips-and-tricks-for-running-strimzi-with-kubectl/.

[30] *Using Helm.* https://helm.sh/docs/intro/using_helm/.

[31] *Using Strimzi.* https://strimzi.io/docs/operators/latest/full/using.html.

[32] Ivan Velichko. *How to Grasp Containers - Efficient Learning Path - Ivan Velichko.* https://iximiuz.com/en/posts/container-learning-path/.

[33] *What Is a Kubernetes Operator?* https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator.

[34] *What Is a Makefile and How Does It Work?* https://opensource.com/article/18/8/what-how-makefile.

[35] *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.