TRT GatePOS Middleware

Contents

Terminology

Cluster	The collection of independent servers on which the middleware services are run (on AWS EC2) and managed by Kubernetes. There are five clusters, respectively for Dev, QA, Staging, UAT and Production, each of which has an individual configuration.
Instance	A single service running on a single machine at a given time in the cluster managed by Kubernetes. Multiple instances of a service may be run simultaneously.
Middleware Application	The entire middleware application, comprised of several services.
Node	A physical machine in AWS EC2.
Pod	A Kubernetes concept for an instance (a single service running on a single node). https://kubernetes.io/docs/concepts/workloads/pods/pod/
Service	A self-contained, configured component of the application that provides network connectivity to a Kubernetes pod. These are replicated so that multiple copies are running simultaneously. Middleware services: • api-gateway • api-mapping • data-composer — composer or composite • data-composer-scheduler • content server — server or service • offload server — server or service

Overview

The middleware is built using Maven, which fetches dependencies and ensures they are where they need to be in 3rd party libraries.

See more at: https://maven.apache.org/

Context

An iOS application is used on aircraft by flight crew as a Point Of Sale.

The iOS application communicates via the middleware with Gate's back-end server, TS5, which provides an inventory for a flight including menu options, stock availability, information regarding the specific flight, crew members and credit card blacklist.

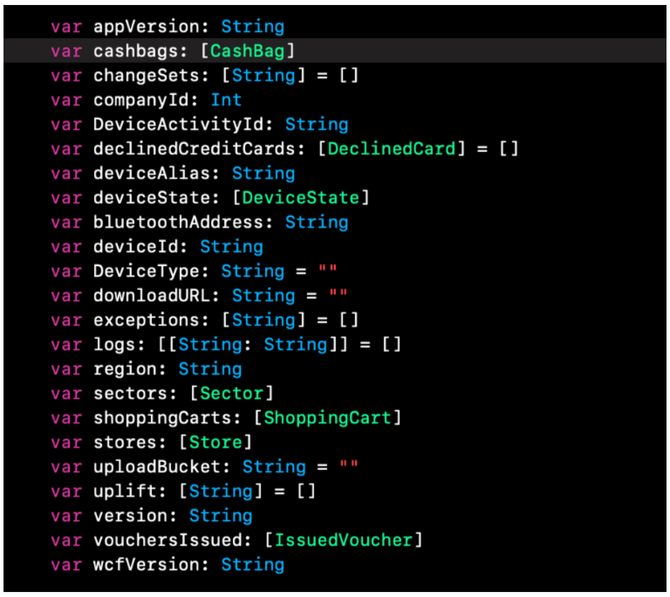
Middleware

- receives data from the iOS app and uploads it to an AWS S3 bucket for collection by TS5.
- receives data from TS5 and restructures it for the iOS app.

TS5 provides significantly more data than is required by the iOS app so, in order to limit the data download to the app, one of the middleware's functions is to reduce the data payload to transfer only that which the app needs.

Before a flight commences, a catering synchronisation is performed between the iOS app and the middleware. Once a flight is completed, the iOS app bundles flight data including all transactions and uploads to the middleware before deleting.   The middleware then uploads the post-flight data to S3, from where the TS5 server downloads it, on a schedule determined by TS5.

Example excerpt from iOS app post-flight data upload:



```
    var appVersion: String
    var cashbags: [CashBag]
    var changeSets: [String] = []
    var companyId: Int
    var DeviceActivityId: String
    var declinedCreditCards: [DeclinedCard] = []
    var deviceAlias: String
    var deviceState: [DeviceState]
    var bluetoothAddress: String
    var deviceId: String
    var DeviceType: String = ""
    var downloadURL: String = ""
    var exceptions: [String] = []
    var logs: [[String: String]] = []
    var region: String
    var sectors: [Sector]
    var shoppingCarts: [ShoppingCart]
    var stores: [Store]
    var uploadBucket: String = ""
    var uplift: [String] = []
    var version: String
    var vouchersIssued: [IssuedVoucher]
    var wcfVersion: String
```

The "shoppingCarts" array contains all the orders that were completed. The middleware does not save the offloaded data, or any other data handled by it.

Structure

The Middleware application is a microservice-based infrastructure, orchestrated by Kubernetes and deployed into any environment capable of hosting Docker containerised solutions: currently AWS, which allows for scalability of the system. The middleware is deployed using the Blue-Green technique, with multiple instances of every service running to make sure that each service has 99% up-time, thus avoiding downtime of the services during the infrastructure updates.

For more information on blue-green deployment, see: https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html .

The API gateway is the single point of entry for communication with the microservices. There are three types of micro-service:

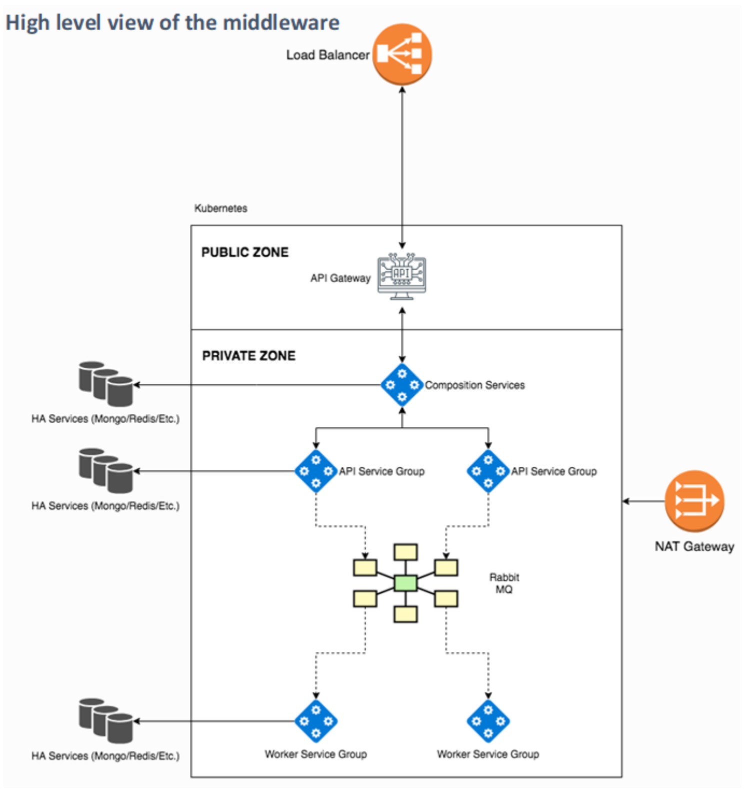
1. **Composite:** The composite is a jobless microservice that makes an initial request to the Gate Group backend to retrieve URLs, which can contain between 5 and 15 different types of data. The composite eliminates the need of having a separate microservice for each type of data. The composite stores Device ID data in a dedicated Redis cache. The composite communicates to each Web (API) service to

fetch data from the Gate Group backend and synchronise the database associated with this Web (API) service for its Device ID.

2. **Web (API) services:** Examples of which are
   - Content service which encapsulates the content result - the URL that contains models.
   - Images service, which encapsulates the images result.
3. **Workers:** The workers expose no APIs and trigger aggregation.

Elastic Load Balancer (ELB)

- distributes incoming application traffic and
- contains all certificates (TLS/HTTPS).



Implementation

The middleware is implemented using the Java programming language, utilising the Spring Boot framework.

Environments

An environment is a deployment of services that work together in order to facilitate communications between the iOS app and TS5 back end service. There are five environments: Dev, QA, Staging, UAT and Prod.

The middleware environments are independent clusters within AWS, managed by the Gate group, each individually configured to communicate with specific backend servers set up (e.g. for testing) to generate data without impacting actual operations.

Each environment is configured is to facilitate different types of testing, except for Production which is used with a live programme release.

The Middleware system is deployed in a QA (Quality Assurance) environment where new features are functionally tested, existing features are smoke tested and regression testing is performed.

When the system is deployed to the UAT environment, user requirement functionality is tested.

Components

Definition of component and microservice

The components of the middleware and the number of simultaneously running instances of each are as follows:

• External API gateway	x4
• API mapping service	x2
• Data composer	x2
• Content server	x2
• Composer scheduler (runs a CRON job every 5-10 minutes, ephemeral)	
• Offload server	x2

Requests can be routed to any of them from anywhere, then processed.

The middleware runs across a cluster of multiple computers within AWS, performing either several tasks independently or the same task, depending on load.

It is implemented as a number of microservices, some of which can be scaled horizontally to accommodate changes in load.

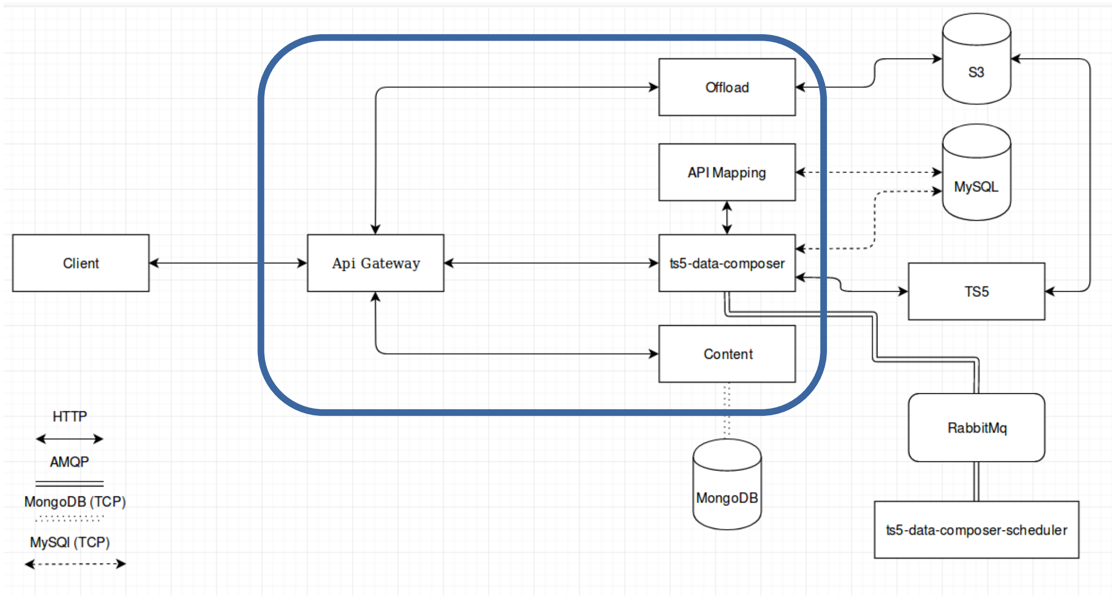
The API gateway service is horizontally scaled to handle required traffic. Four instances of the API gateway run discretely in the production environment, whereas only two instances are ever running in the development environment.

At present, the only service that is scaled is the Composer service that retrieves and compiles Catering Sync data for each mapped airline from its corresponding TS5 endpoint.

It is scaled according to how many airlines the system supports. There is one Composer service instance per airline.

The data-composer-scheduler is triggered periodically to fetch from TS5 all catering data, which the middleware processes and in some cases augments, before placing it in the content service.

Data exchange via middleware between iOS app and TS5



Hardware

Middleware is run purely on Amazon Elastic Compute Cloud (EC2), on a T2.medium instance, with 2 CPUs and 4GB RAM. For specific details of the AWS server, see https://aws.amazon.com/ec2/instance-types/ .

External services

Databases

Redis	In-memory data store used to cache database queries for faster access.
S3	Managed AWS service for flat file storage. Files and objects only - part of the data sent to the iOS app is in .png format, images come from S3. This is also used for data offload post-flight. On closing a sector, all transactions packaged into a response are sent by iOS to the middleware as a base64-encoded GZIP JSON file. The middleware decodes and decompresses the file, dynamically creating a file name from the following fields: companyID, deviceID, deviceActivityId, timestamp. It then uploads the decompressed file to S3, which TS5 retrieves from there according to a Gate-configured schedule.
MongoDB	Non-relational, document-based data storage contains data returned from TS5, unstructured.
MySQL	Relational datastore, for well-formed data (small amount) e.g. for every airline, access tokens, API URL.

Other

RabbitMQ	Asynchronous distributed message broker service.
InteliJ IDEA	IDE for Java development.

Database usage by component

api-gateway	does not use databases
api-mapping	**MySQL**: relates an airline to a backend TS5 API to get info for that airline, maps an ICAO code to a URL
data-composer	Currently using **MySQL** and **MongoDB** , but after refactoring will use neither. Data will be stored that was cached directly in the content service.
data-composer-scheduler	does not use databases
content server	**Redis**: in-memory database, mostly for caching content to be served (catering sync payloads)
offload server	**AWS S3** bucket.

Note:

Every environment has multiple S3 buckets. Documentation is also stored in S3.

Other data storage services

Nexus provides repos for shared Java library and Docker images (credentials required)

- https://nexus.bsgg.co.uk/#browse/browse:gate-snapshots:com%2Fbsd%2Fgate%2Fgrd-core-lib
- https://nexus.bsgg.co.uk/#browse/browse:gate-releases:com%2Fbsd%2Fgate%2Fgrd-core-lib
- https://nexus.bsgg.co.uk/#browse/browse:bsgg-docker:v2%2Fgate

One java library is currently stored in AWS. All docker images and libraries will be stored in AWS after MW refactor to v6.

Scaling

The middleware does not currently auto-scale (respond in real time by creating more virtual instances of a service) according to traffic fluctuations.

The API gateway however, which handles the iOS requests, is run in four simultaneous instances in order to have redundant capacity.

Real-time monitoring

In the production environment, Gate has implemented real-time monitoring using the Grafana dashboard to display their own specified metrics collected from all instances. TRT do not have access to this data.

Redundancy

Kubernetes runs our services in Docker containers and monitors them to ensure the containers are running. If Kubernetes determines that a pod has crashed, it schedules it for removal and replaces it.

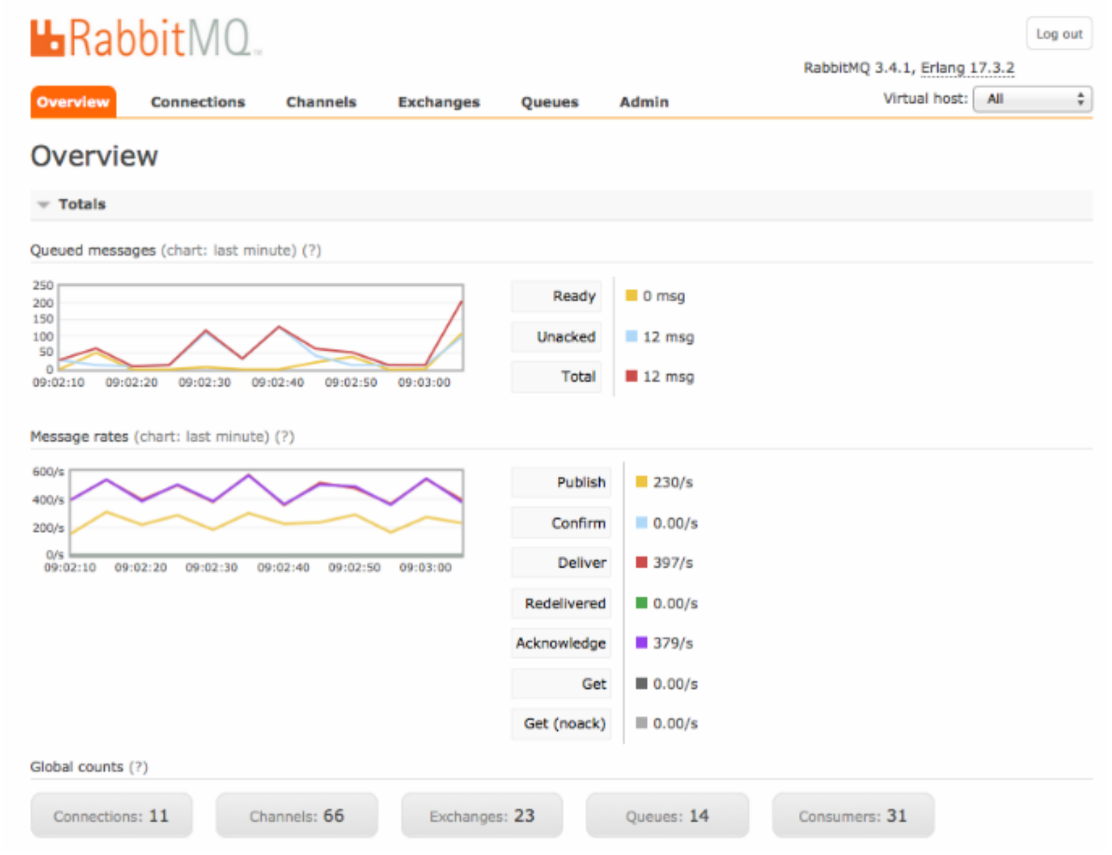
This can happen in two ways:

- When Kubernetes removes the crashed pod, it can start restart that pod on the same node. This results in a faster restart time, because the Docker image is already downloaded to that node.
- Alternatively, it will restart that pod on a different node, requiring the Docker image (500 MB) to be downloaded again from Nexus.

Once the Docker image downloads, the service takes around 20 seconds to start. In the worst case, it will take up to 90 seconds to download and start a new instance on a new node.

System alerts

Gate has a PCI-compliant dashboard for RabbitMQ, only available to them within their Amazon VPC. This measures the number of messages on queues, connections being handled, available queues and their properties.



RabbitMQ 3.4.1, Erlang 17.3.2

Virtual host: All

Overview

Totals

Queued messages (chart: last minute) (?)

Ready	🟨 0 msg
Unacked	🟦 12 msg
Total	🟥 12 msg

Message rates (chart: last minute) (?)

Publish	🟨 230/s
Confirm	🟦 0.00/s
Deliver	🟥 397/s
Redelivered	🟩 0.00/s
Acknowledge	🟪 379/s
Get	⬛ 0.00/s
Get (noack)	⬜ 0.00/s

Global counts (?)

Connections: 11 | Channels: 66 | Exchanges: 23 | Queues: 14 | Consumers: 31

Support set-up

Gate is responsible for monitoring the operation of the middleware. Marlabs provide 24/7 support.

Airline-specific technology

The iOS app only integrates with Gate technologies, via the middleware. It has no integration to airline-specific technologies.

Service inter-operation

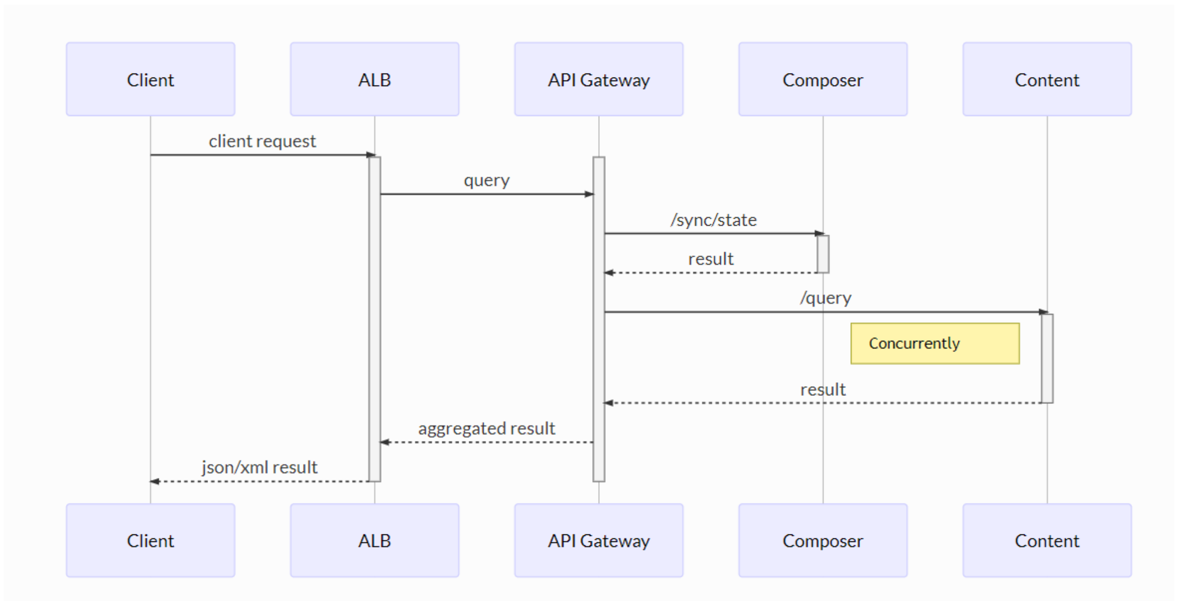
How do services influence each other? In two ways:

(1) Looking for a specific response over HTTP REST, inside the cluster

(2) Pushing messages onto a queue which are not looking for specific response, simply creating work on a queue. This uses a transport layer called Advanced Message Queuing Protocol (AMQP) for RabbitMQ to push a fire-and-forget message. RabbitMQ server ensures message is passed onto another component to execute a task.

Example of use:

- Data Composer Scheduler (CRON) starts, pushes message to queue scheduling background sync
- The message is placed on a queue from where it is picked up by one of the instances of the Composer microservice.
- The message is received by Composer microservice, which performs a background sync (to fetch latest data from TS5 and prepare it for the iOS app) and triggers notification of this action.



Back-up contingency

Data is persisted to MongoDB with 3 nodes for failover in the Redis cluster on AWS EC2 using Elastic Block Store. As the middleware retains no data and all data comes from TS5, it is within TS5 that redundancy is required.

Load balancing

The Elastic Application Load Balancer, which is part of the AWS EC2 Cloud Service, routes service requests for a microservice to one of the instances of that microservice.

Example of use:

- There are four API gateway instances to which 100 request messages are routed.
- The load server splits these into four packets of 25 messages and performs the SSL termination for the middleware.
- The messages are processed without encryption, within the cluster (HTTP).

Middleware payload

See `Payload documentation`

Middleware monitoring

See `MW Monitoring`

Further Reading

Internal documentation

Data uploaded by the iOS app post-flight	ePOS sync Payload
Architecture of the middleware	Architecture - Middleware

External documentation

Amazon Redis Cluster	https://aws.amazon.com/redis/Redis_Cluster_101/
Amazon Elastic Compute Cloud (EC2)	https://aws.amazon.com/ec2/
Application analytics platform: Grafana	https://grafana.com/docs/
Cloud repository manager: Nexus	https://nexus.bsgg.co.uk/#browse/welcome
Application packaging service: Docker	https://docs.docker.com/
Application container orchestration engine: Kubernetes	https://kubernetes.io/docs/home/