Flight Display

Uses cases

1. Big ticker board will fetch upcoming depating flights from the web API
2. Airline admin makes updates to the airline's departure schedule
3. Airline staff makes updates to change the status of a particular flight

Traffic

Inbound

1. From big ticker boards
2. From schedule updater app
3. From flight status updater app
4. Customer website where they can check flight statuses (and subscribe to a particular flight)
5. Must handle spikey traffic from point 3

Outbound

1. Push notifications for customer flight subscriptions

Data Modelling

flight_details

- Stores the details of the flight such as flight numbers and destination
- Has a unique index on flight_number and airline_id columns. Since it does not make sense to have duplicate flight_numbers within the same airline.

airlines

- Stores informaiton about the airlines.

employees

- Stores information about the airlines.
- Most important relationship is the airline_id. It tells us which airline this employee works for

- Has a unique index on email. This implies that an employee can only work for one airline at a time

schedules

- Stores the schedule of the flight_detail
- We use boolean columns to store the days of the week the flight is scheduled for. This makes it easier to accomodate query the data in the future since each day of the week is treated as a column. Let say, we stored this as a string of serialized JSON, e.g. ["mon", "fri"], it makes it harder to perform filters on these columns
- a unique index on flight_detail_id since we want to ensure a flight_detail only ever have one schedule. This also speeds up lookups. When an employee wants to update a schedule, they first must select the flight number first.

flights

- Stores information about the flight. Such as departure time, gate, and status.
- This will be by far the fastest growing table in terms of size and the most read-heavy. Hence, we want to speed up the reading of this table by denormalizing it to avoid table joins. As you can see, flight_number and destination_name columns are all denormalized.
- We want an index on actual_departure column. This ensure we retrieve upcoming flights quickly

flight_subscriptions

- Stores user subscriptions to flight updates for push notifications
- Has a unique constraint on user_id and flight_id. Since a user can only subscribe to a flight once
- Once again, this will be a read-heavy table, so we try to denormalize fields to avoid table joins

users

- Stores users, or travellers, of the system
- Has a unique index on email
- We want to make the sign up process as seamless as possible. Hence, we only ask the user to enter an email (for identification purposes mostly). A password is not needed. Upon sign up, the API server will generate a

JSON Web Token (without expiration) which will be stored securely on the mobile device. Keychain in iOS and KeyStore in android

devices

- Stores information about each of the user's devices
- This means our system supports users logging in from multiple mobile devices
- We want a unique index on the user_id and app_identifier to ensure a device is only associate with the same user once. This does still allow different users to use the same mobile device

System Design

Components

Load Balancer

- Distributes web traffic to each of the API servers
- Does SSL Termination
- HAProxy
- Uses round-robin
- Has a static public IP address

API

- Serves public API requests. Flight schedules and subscribing to push notifications
- RESTful
- API versioning done via uri path e.g. host/api/v1/...
- Use JWT for authentication. No expiration.
- JSON request/response format
- NodeJS
- is within an autoscaling group - scale based on average CPU utilization rate
- Session-less

DB Router

- Load balances and distribute db queries to master and slave databases
- MySQL Router
- Directs writes to master
- Directs reads to slave

Master/Slave databases

- Databases in a master-slave configuration
- MySQL

FlightUpdaterAPI

- Handles private requests to update flight schedules and statuses
- Same api design and technology as the API servers
- Puts 6 events onto a queue after a business operation has successfully computed

1. flight_status_updated - whenever the status of a flight has been update
2. flight_departed - whenever a flight has departed
3. flight_cancelled - whenever a flight has been cancelled
4. schedule_created - whenever a schedule for a flight has been created
5. schedule_updated - whenever a schedule for a flight has been updated
6. schedule_deleted - whenever a schedule for a flight has been deleted

Message Queue

- A message broker for storing business events
- ActiveMQ

NotificationPusher

- A worker responsible for sending push notifications to mobile devices
- Is configured with the Apple Push Notification private key and the Google Cloud Messaging credentials
- Listens for flight_status_updated events from the message queue
- Upon such an event, it will look into the flight_subscriptions table and find all the users subscribed to that flight. It will then go through and load all the devices for all the users, and send a push notifcation to all the devices
- NodeJS

FlightScheduler

- A worker responsible for populating the flights table
- It listens for events from the message queue
- schedule_created - worker will insert a new entries into the flights table.

- NodeJS

For example, 1. a schedule was created with departure_time="14:00" and active_on_mon=true, active_on_wed=true, and active_on_fri=true 2. a schedule_created event is emitted 3. Upon receiving the event and its payload (which is the schedule json object) - the FlightScheduler will check the current day of the week. Let's say its current tuesday. It will inspect the schedule and find the next scheduled flight is wednesday at 14:00. It will then create an entry in the flights table with those fields. The status column's default value is OnTime

- schedule_updated - will delete the most recent entry for the schedule_id in the flights table and insert a new one based with the new schedule information
- schedule_deleted - will delete the most recent entry for the schedule_id in the flights table
- flight_departed - generate the next scheduled flight and insert into the flights table
- flight_cancelled - generate the next scheduled flight and insert into the flights table

Private VNET

- A private virtual network
- Can only be accessed via the Load Balancer or the VPN
- The VPN is configured to only allow traffic to the FlightUpdaterAPI
- Includes a jumphost where administrators can log in to access machines residing in the private vnet. We use a jumphost to increase security. We only need to harden one machine, the jumphost itself, and none of the other machines because they are not public available. The jumphost will use IP whitelisting to ensure its only accessible from a certain IP range.

API Endpoints

Public API

- GET /api/v1/flights - returns all upcoming flights where actual_departure is NULL or greater than one hour ago
- POST /api/v1/users - creates a new user. Will return http status 400. Returns a JWT token
- POST /api/v1/flight_subscriptions - subscribes to a flight for push notifications
- POST /api/v1/users/id/devices - registers the mobile device for push notifications
- GET /api/health - used by the load balancer

FlightUpdaterAPI

- POST /api/v1/employees/:id/session - logs into the employee account. Returns a JWT token with 8 hour expiration
- GET api/v1/schedules - views all the schedules for the employee's airline
- POST/PUT/DELETE api/v1/schedule/:id - creating, modifying, and removing schedules
- GET api/v1/flights - list all the scheduled flights (where departure_time is NULL or greater than 1 hour ago) for the employee's airline
- PUT api/v1/flights/:id - update the flight. One of the validations is if the flight's status to be updated is BOARDING, the depature_gate needs to be present.

Requirements meet, tradeoffs, and assumptions

Requirements Met

1. Only departing flights are accounted for in the system
2. Airline employees can use the "Update Flight Schedule App" to create weekly schedules for flights
3. Airline employees can also use the "Update Flight Schedule App" to modify/delete schedules
4. The API allows any system to retrieve a list of upcoming flights
5. All the attributes of a flight is persisted in the database
6. We have an API that the big ticker board can use
7. There is a "Flight Status App" that travellers can use to check the status of a flight
8. The API servers will be a configured with a autoscaling group to handle any traffic spikes. Also, there is a db router to load balance read requests to database slaves
9. Travellers can use the "Flight Status App" to subscribe to push notifications for specific flights
10. The FlightUpdaterApi authenticates the id of the employee to make sure they can only view and edit the flight information for the airline they belong to
11. The FlightUpdaterAPI is not exposed to the internet. It's hidden inside the private virtual network. Only VPN access to it is allowed.

Tradeoffs

- Assuming this system will be read-heavy instead of write-heavy. Thus, have gone with a master-slave configuration for the database. This was a conscious decision over using an in-memory cache, such as memcached.

The caching strategy (when to warm the cache, when to invalidate etc) seemed more complicated to manage than database replication at this stage. The trade-off is we have increased storage size because now we have extra database servers to manage.

- We want to abstract the application code from knowing about the master and slaves, and just believe there is only one database at play. This is why we chose to use a db router, such as MySQL Router to route read requests to the slave and write requests to the master. This increases the architectural complexity but reduces the application code complexity. This also makes adding new database instances easier in the future because we just need to make a configuration change in the db router, and not in each of the app servers themselves. We have also introduced a single-point-of-failure. If the router dies, none of the services can access the databases. An alternative is to run the router on each of the API servers. It's a pretty light-weight process and we get the benefit of redundancy.
- When the status of the flight changes, we update the column *flight_status* of the corresponding row in the *flights* table. It's important to note that we do not track when the status was changed. If the airline requests for information such as, "tell me how long it took for flight XXX to change from BOARDING to DEPARTED", we would not be able to do so. This might be a viable feature in the future, where the airline might want to view such information in their business intelligence platforms.
- We use JWT tokens to secure our API. To ease development, we chose not to use refresh tokens. This means after our JWT expires, users are forced to re-login. This user experience can be improved iteratively with time
- Because we are using a VPN connection to access our FlightUpdaterAPI, this increases the operational overhead of rolling out the software to the airlines. We have to make sure their devices are pre-configured with a VPN client and have to provision credentials.
- We have chosen asynchronous communication via message queues to propagate business events from services. This increases the scalability and robustness of our system. However, this comes at a cost of architectural complexity and operational overhead. We have to configre and monitor uptimes of our message queue. Furthermore, this makes debugging the system a little harder because we have to trace events.
- We have chosen to refactor FlightScheduler and NotificationPusher into their own processes. We can actually run these two processes on the same VM, and move them out into separate VMs when load demands it. Regardless, running these processes as separate workers means we have more opational overhead because we have to monitor the running and uptime of these workers. However, the advantage it gives us is the ability to scale these processes independently from each other and the APIs, and it also gives a bounded-context to make scheduling-related changes to our system without affecting the API logic.
- We have chosen to build the pushing of notifications in-house, through the

NotificationPusher. This may be re-inventing the wheel since there are pretty good services out there who do such things. An example, Notification Hub on the Azure platform is designed for this exact purpose. So, depending on the hosting platform, we can move the pushing of notifications to some of their third party servies if required.

Assumptions

- To update the status of a flight, it is done via a mobile application (e.g. running on an ipad device). An airline personnel managing that particular flight will log into the system, find the flight they are managing, and manually update the flight status at the appropriate times.
- There is no distinction of employee roles, so any employee in the system could update the flight schedules and flight statuses
- The same flight can depart from the same airport once per day. This is a big assumption. The data database table, schedule, is currently modelled in such a way that it does not accomodate same flight departing more than once from the same airport. If this assumption is wrong, we would need to refactor the table layout. Not much data migration is needed because in this system, we almost never care about older flights i.e. once the flight is departed, it is never shown on the big ticker board.
- The system is not required to generate reports. Hence, we do not keep track of audit information such as when a status of a flight was changed.
- A employee can only belong to one airline. That translates to a business constraint, a employee can only work for one airline at a time.

Development effort

A bit confused about this question. How long it takes depends on team size and experience. And even so, we estimate the development effort in terms of points rather than duration. Furthermore, the architectural diagram does not tell the entire picture. For instance, there will most likely be tech related stories such as setting up a CI pipeline, automating deployments, performing tech spikes etc

However, for this question, I will just assume I am the sole developer of this system and the estimate is only for what you see in the diagram excluding all the devops. In this case, it work take me: 8 months...