Published in A peek into my mind…

Partha Pratim Sanyal Follow
Jul 11, 2020 · 17 min read · Listen

System Design: DoorDash — a prepared food delivery service

Introduction

System Design interviews have gained a lot of steam over the last couple of years and are considered as an important ***event*** on the day of onsite interviews with the majority of product development companies. These interviews tend to be very open-ended and conversational. The interviewers are assessing the candidate based on their technical skills involving thinking about the big picture, articulation, friendliness (Will I be able to work with this person?), curiosity, and logical understanding of the presented problem domain. The candidate is given an example of a ***popular*** system and asked to brainstorm a design for the same. During the process, candidates are expected to ask clarifying questions and think through every component, including the data model, keeping scalability, redundancy, fault tolerance, etc in mind.

The intended audience for this article is software engineers who are interested in designing systems.

This article discusses a popular food delivery service named **DoorDash** that has saved the lives of millions of hungry people around the world, who either find cooking a sort of abomination, or are just not *in the mood* to cook, but are starving and can't wait :). Of course, it is a detailed analysis and not all of it can ever be discussed in a traditional 45 minute/1 hour interview. I have tried to not dive deep into the individual components as the idea is to just have a high-level overview of various important pieces of the puzzle. Hopefully, this can give you an idea of the intricate relations between several components within an ecosystem.

How to start?

src: http://clipart-library.com/clipart/1402216.htm

gain!

2. ***Don't jump into the design*** right away.
3. Start thinking through the problem and ***ask a significant number of clarifying questions***. Again, don't spend 20 minutes asking questions, that won't work. Don't avoid the problem and waste time with useless chatter!
4. Discuss the ***scope of the problem*** with your interviewer. Don't try to design each and everything. Remember you have 45 minutes only! You got to identify and focus only on the most important use cases.
5. ***Identify the users*** (actors) of your application. This is important! Companies like Amazon obsess on the customer; you will earn brownie points if you specify that.
6. ***Talk about the scale, talk about growth***. Remember considering scalability from the go isn't always necessary but good to keep in mind and not get stumped when the interviewer asks you "*How will you scale this system for 1 Billion users*?"
7. ***Think out loud***! Unless the interviewer is capable of penetrating mental discernment, there's no way for that person to peek into your mind.

What is DoorDash?

> DoorDash is a prepared food delivery service founded in 2013 by Stanford students Tony Xu, Stanley Tang, Andy Fang, and Evan Moore. A Y combinator–backed company, DoorDash is one of several technology companies that use logistics services to offer food delivery from restaurants on-demand. DoorDash launched in Palo Alto and, as of May 2019, had expanded to more than 4,000 cities and offers a selection of 340,000 stores across the U.S. and Canada. The company is currently worth more than $13 billion and is the largest third-party delivery service in the USA, surpassing Grubhub in 2019.

Requirements and Goals of the System

Main actors in our system

- Customers/consumers
- Restaurant (Merchant)
- Doordashers (Drivers)
- DoorDash Admin (from the company)

Assumptions

- Unlimited food is available between the times a restaurant opens up and closes. So, no need to check for the quantity of available food. No inventory management required.
- Restaurants don't have their own online ordering system/infrastructure, they will solely rely on us for all orders.
- Customers will be shown restaurants within a particular radius, say 10 miles.
- Customers are allowed to order food from only one restaurant at a time in an online order. Menu items from multiple restaurants can't be combined in an order.

Customers should be able to:

- Search for restaurants based on the cuisines type, menu items, etc.
- Create a cart, add menu items to the cart and order.
- Receive notifications about the status of an order once placed.
- Track the status of the order on the app.
- Cancel the order.
- Pay for the order.
- Create/Update their account and contact information.

Restaurants should be able to:

- Create their profile (onboarding) and create/refresh/add new menu items, pictures.
- Receive notifications about orders placed, assigned doordasher, and update the status of order etc.
- Offboarding: If the restaurant goes out of business, or decides to discontinue taking online orders.

Doordashers should be able to:

- Receive notifications about the available orders in the area, from which they can choose.
- Know when the order is available for pickup.
- Inform the customer/restaurant of any problems they encounter while executing the order pickup/delivery.
- De-register in case they don't want to continue with the job anymore.

Non Functional Requirements

- ***Latency****: Hungry* users will not like to wait to see the menu/ restaurant details, so the search functionality should be very fast. The ordering experience should also not have high latency and must be seamless and fast. Also, since the Restaurant/menu related data will be entered through a different service altogether, the lag between the data being entered and the result showing up in the search should be acceptable but shouldn't be too much.
- ***Consistency****:* When a new restaurant is onboarded, or a new menu is added, the information needn't be available immediately. Eventual consistency is desirable. However, when an order is placed, the customer, the restaurant, and the door dasher should see the same order without any issues. Hence, in this scenario consistency is imperative.
- ***Availability****:* High availability is desirable for the best experience of a customer and also the restaurants that are processing the order, as well as the doordasher. No restaurant would like their business to go down just because the system has crashed, that's a serious loss of $.
- ***High Throughput***: The system should be able to handle high peak load without problems or failures.

Out of scope

- Customers being able to rate a restaurant or a specific doordasher.
- Sales reporting, dashboarding capabilities.
- Doordasher onboarding & payout.

- OAuth 2.0 authentication via Facebook/Google/Apple, etc.
- Recommendations based on customer's order history/profile and other preferences.
- Mapping capabilities (in the form of Google Maps), to show the location of the doordasher, or location of the restaurant.
- ETA calculation using Isochrones.
- DoorDash admin functionalities.
- Supply/demand logistics & decision making.

Capacity estimation & Constraints

Let's assume we are catering to users in 10M area codes. This number will grow.

On average, each area code can have 100 restaurants.

Each restaurant can have 15 dishes that can be served.

Total records of dishes = 10M * 100 * 15 = 15B

Number of customers: 20M

If each customer on an average places 2 orders every day, number of orders in a day = 40M

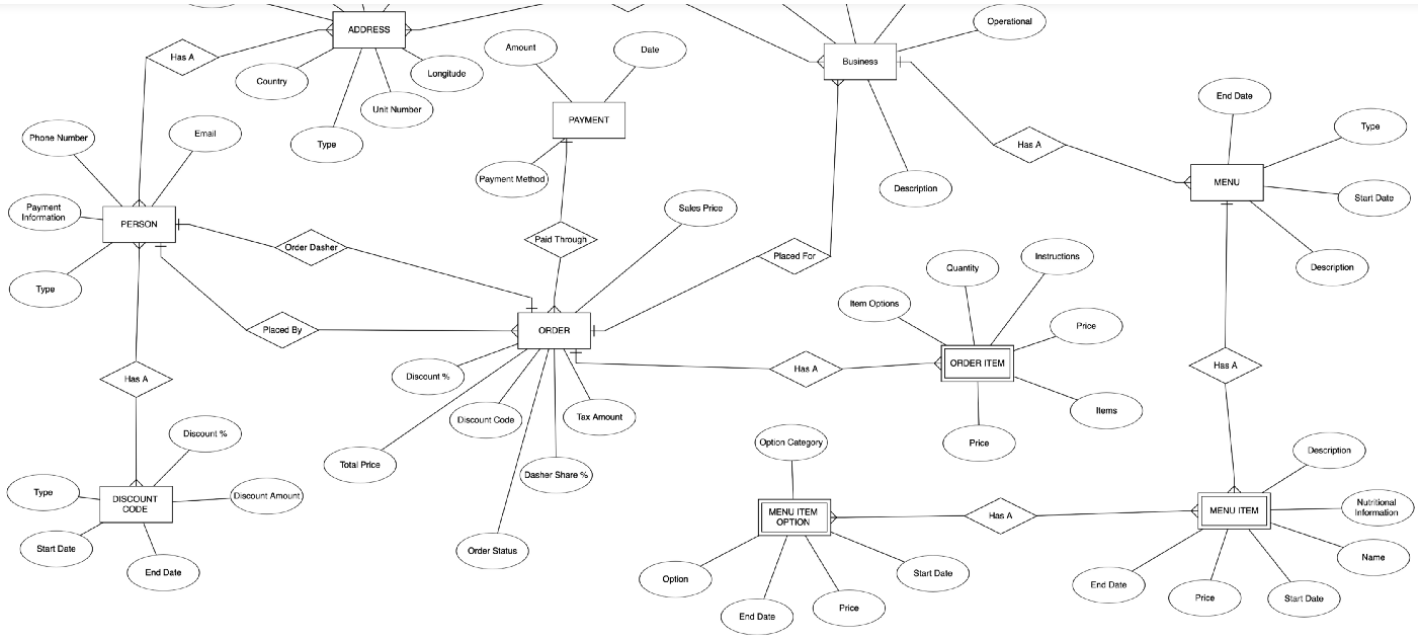
The peak time of orders will vary usually based on the day of the week. For example, weekends might have more orders than on weekdays. Peak times could be somewhere around noon or at dinner time in each region.

In general, the searching of menus/restaurants will be ***read-heavy*** and the ordering functionality will be ***write-heavy***. The potential of customers looking up past orders once they have been delivered and the food has been consumed is very less.

Data Model

It's not necessary to do an extensive analysis of the data model during the interview, merely listing out the main entities and attributes should be sufficient in the context. Also, it's not required to come up with a diagram like below, this is only for demonstration purpose.

A sample ***Entity Relationship Diagram*** for the application:



DoorDash Sample ER Diagram

Data Storage

The choice of the database usually depends on the amount of data that is being stored, the ease of scaling, partitioning, replication, and several other factors. Application owners may choose to use a mixture of different databases to accomplish certain use cases. For ***ACID*** (Atomicity, Consistency, Isolation, Durability) requirements, a relational database is always preferred over a NoSQL counterpart. NoSQL and relational databases have their own prospects and constraints and the decision to use either should be clearly thought through based on functionalities.

As evident from the capacity estimation, the amount of data of restaurants, menu descriptions, user data, dasher data, etc is going to be huge, and hence, NoSQL/ Columnar database like ***Cassandra*** could be used. The structure of data, especially attributes might also vary between restaurants and it could be difficult to fit in the data into a relational schema.

Pictures (Restaurants, menu items) can be stored in an object storage service like ***Amazon S3***.

Ordering is a transactional process and can be stored in ***Oracle/MySQL/Postgres***.

Component Design & Architecture

The overall architecture could be ***microservices*-based** with heavy usage of the ***publisher-subscriber*** pattern, involving a queuing technology like ***Kafka***, ***RabbitMQ***, ***ActiveMQ***, ***Amazon SNS***, or ***Amazon MQ***. Each microservice can talk with another one using this model of publishing a message and subscribing to channels or topics. This makes the services de-coupled from each other in the best possible way. Consequently, microservice A doesn't need to know the endpoint of microservice B when it publishes a message to the queue. So, the publisher doesn't need to know the consumers (subscribers) of its published messages. Similarly, the subscriber doesn't know about the source of the message, i.e the publisher. The Pub/Sub system becomes a broker and serves as a contract between all the involved parties.

Also, it's imperative that each microservice interacts with their own database and doesn't share with anyone else. This approach is motivated by the **database-per-service** paradigm. Microservice A doesn't have direct access to microservice B's database as they are separated and individually owned.

As described earlier, the data model proposes the idea of just one big fat schema containing every possible table. Microservices architecture is opposed to this concept. ***Functional partitioning*** is required in order to grant the ownership of each significant table or

Various components of the system

UI Client

The application will be accessible via mobile, web, tablets, etc. Based on the actor, the interfaces presented will be different, or a combination of many. So, individual interfaces/pages will talk to the respective service for parts of the functionality. For example, the search can be performed by ***Restaurant Search Service,*** orders can be handled by ***Ordering Service***, and so on.

Primarily, there will be four versions of the interface for the four actors, viz customer, restaurant, doordasher & admin.

Search Ecosystem

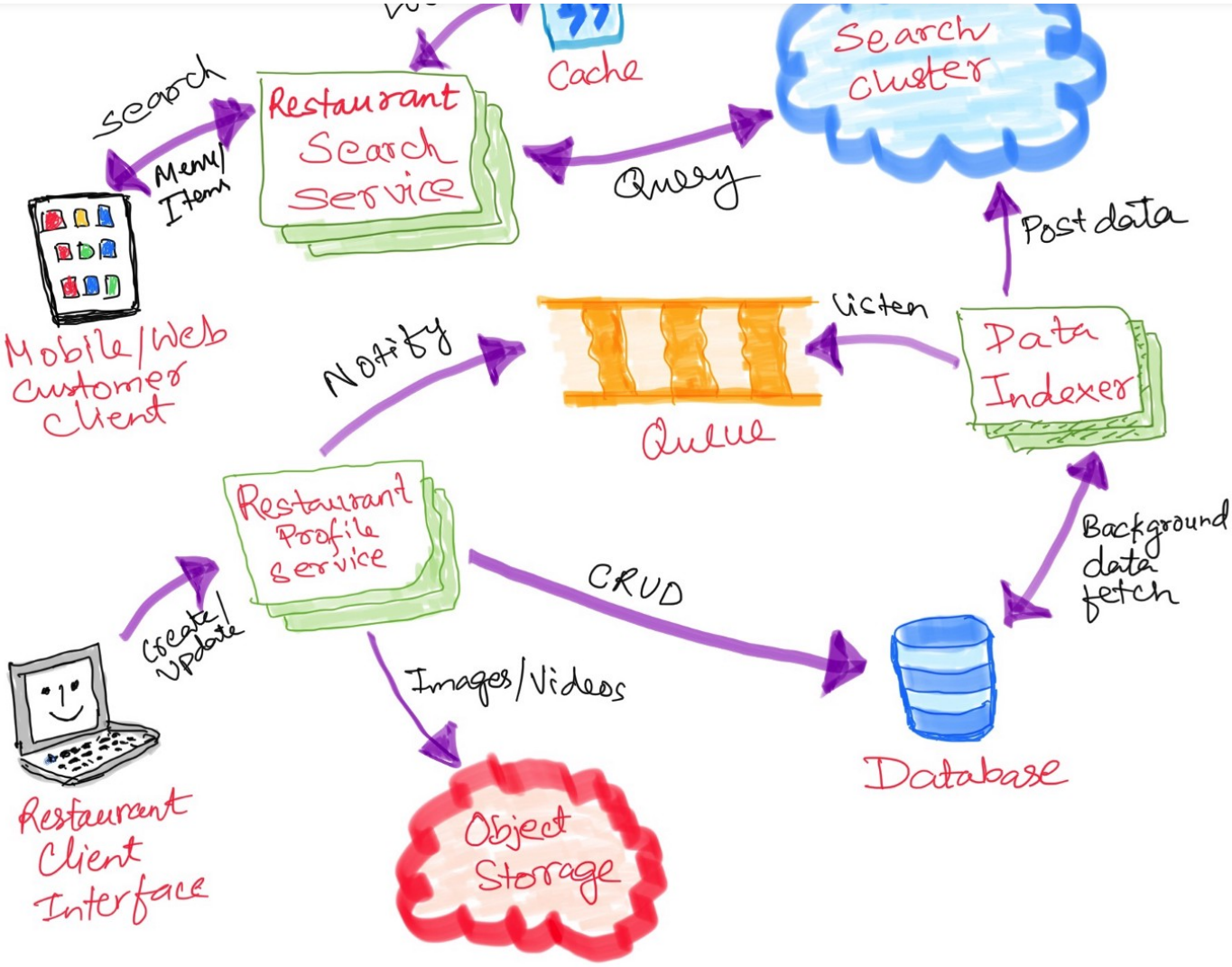
The most important and coveted functionality that has to be provided by the system is the capability of searching on menu items, cuisines, restaurants, among other things. In the food ordering journey, this functionality will be the point of entry for all the customers, unless they already have a favorite restaurant in their preferences from which they can select dishes. Thus a personalized discovery & search experience based on a customer's past search and order history has to be provided. As is apparent, this particular part of the entire system will be ***read-heavy***.

We can think of leveraging popular off-the-shelf search offerings from the market such as ***Elasticsearch*** or ***Apache Solr*** for quick lookup as per user search parameters. Both of these technologies are open-source, distributed, and based on ***Apache Lucene*** and have their own strengths and weaknesses.

We will need to have a ***queue*** in place to process ***asynchronous*** updates to the search cluster. When the ***Restaurant Profile Service*** (see below) creates/updates a restaurant/menu data by performing CRUD operations on the database, it can also post an ***event*** to the queue. This event could be any of the CRUD. We need a ***data indexer*** that listens to the queue for such events. The data indexer then picks up the event, runs a query against the database to formulate a document as per the correct format, and posts the data into the search cluster. We also need to have a ***Restaurant Search Service*** that executes queries on the search cluster based on user inputs and returns the result to be displayed on the user interface.

Elasticsearch has a ***Geo-distance query***, which can be leveraged to return all the restaurant/menus that the user is searching for based on a defined radius from the location of the user. What this essentially means is, users will be shown only those restaurants that are ***reachable*** from the users' addresses. Similarly, Apache Solr also has ***Spatial Search*** which caters to the use case of geographic search. As such, Elasticsearch is very fast in retrieving results and can be directly queried. In order to further reduce latency and improve user experience, we should use a cache, working alongside the ***Restaurant Search Service.*** More on caching later.

The search ecosystem will look like the below :



Search Ecosystem

Ordering Service

This service will manage the menu selection, shopping cart management, and placement of food orders. It will process the payment using an ***External Payment Gateway*** and persist the result into an ***Orders*** database. Because order placement is transactional in nature, the best idea is to use a relational database.

Customers will be able to get the full receipt of their order using this service along with other details about the order. They can also cancel the order if they changed their mind for some reason. Customers can also view their past order histories.

Order Fulfillment Service

Few high-level functionalities that this service will handle are as below:

- The restaurant accepts the order (using the client app)
- Customers can check the status of their orders.
- Doordashers can check if the order is ready to be picked up. They can also view the details of the current order they are picking up/delivering.

The actors in the system, namely, customers, restaurant staff, doordashers, and system admin will need a way to create their profile with personal information, contact, address, and will be assigned a role based on their profile. Individual actors will also have preferences based on their role. For example, customers may have set preferences for selecting from a fixed set of restaurants or zip codes or cuisines. Doordashers might have a preference for delivering only within their specific area codes, or choice of restaurants, etc. Similarly, actors will also have their own method of ordering or getting paid as appropriate. Notification preferences of actors will also vary. This service will manage the profiles and preferences of all such actors across the board.

Doordasher Dispatch Service

This service will be used to accomplish use-cases relating to a doordasher. A doordasher will be able to:

- View pick-up orders from a list, that they can accept.
- Accept an order.
- View past orders that they have accepted.
- View the customer information from the order in case they need to communicate with them about delivering, or inform the restaurant about any problem that stops them from picking up or delivering the food.

Restaurant Profile Service

This service will be managing the data related to restaurants, menus, offerings, etc. A restaurant or business can:

- Onboard
- Update/Delete their profile.
- Create/Update/See/Delete menus, dishes etc.
- Upload images of their restaurants or dishes/menus into an object storage service.
- View their financial details based on past orders etc.
- Setup a payment method for money transfers.

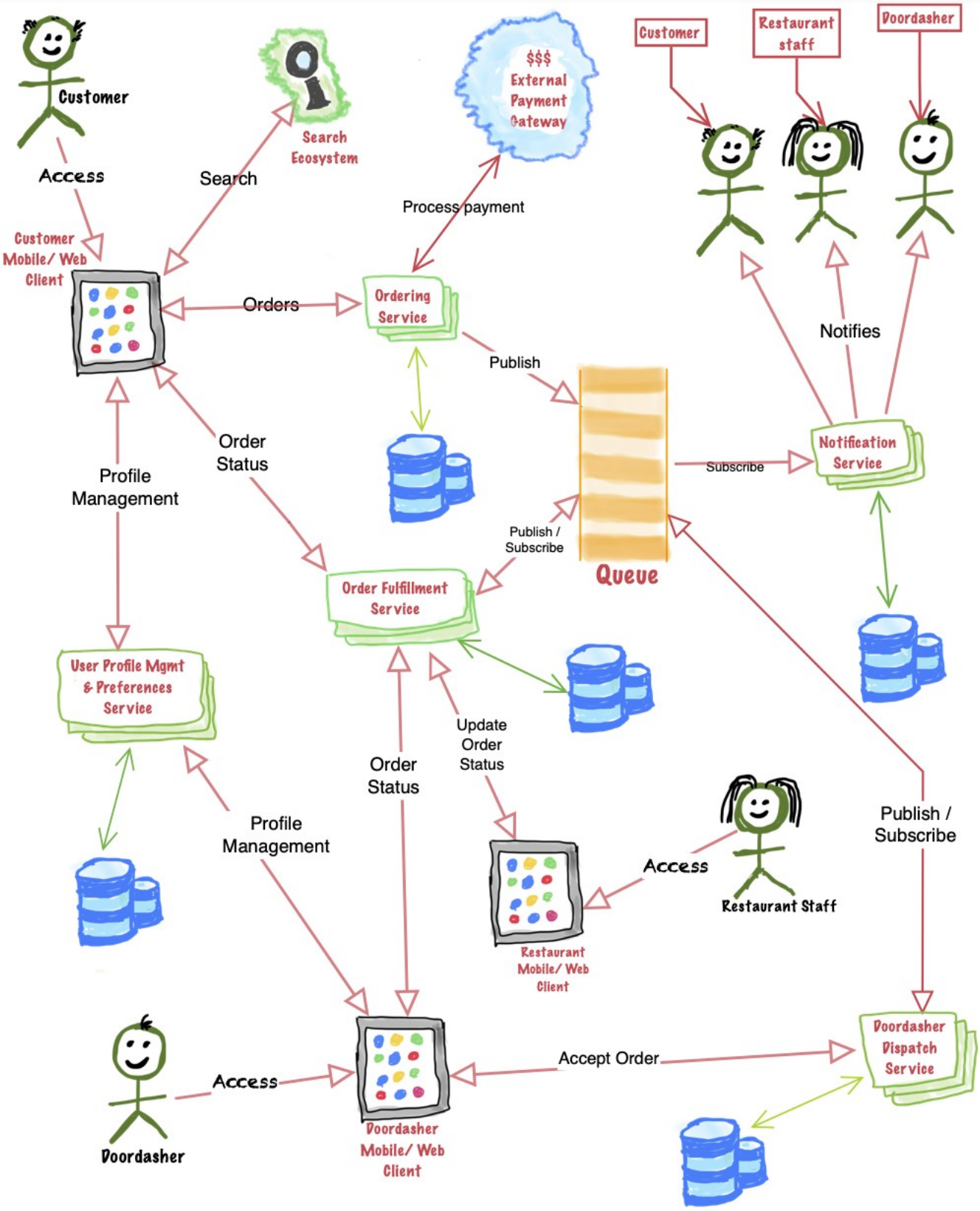
External Payment Gateway

This component can interface with popular payment gateways like Amazon Payments, We Pay, PayPal, ApplePay, or individual Credit Card providers like Amex, Visa, Mastercard, etc. The Order Service will interact with this component to ensure the payment is done at the time of confirming the order. The interaction should be synchronous in nature.

Notification Service

This service is responsible for delivering notifications to every actor with the system. The notifications could be sent out to the individual actors in the preferences they have set for receiving them. Some actors might prefer push notifications, some might receive text or emails. This service is supposed to abstract out the medium in which notifications are being sent. This means the underlying interactions with the mobile carrier, email service providers, etc will be abstracted. Actors could also receive ***In-App*** notifications. The responsibilities are notifying:

- ***The customer***: about various stages of the order, for example, the order was placed successfully, or, the restaurant has accepted the order, or, doordasher has picked up the order and is on his/her way to deliver it.
- ***The restaurant:*** that an order was placed, or a doordasher is assigned to the order, or the doordasher is on his/her way to pick up the order.



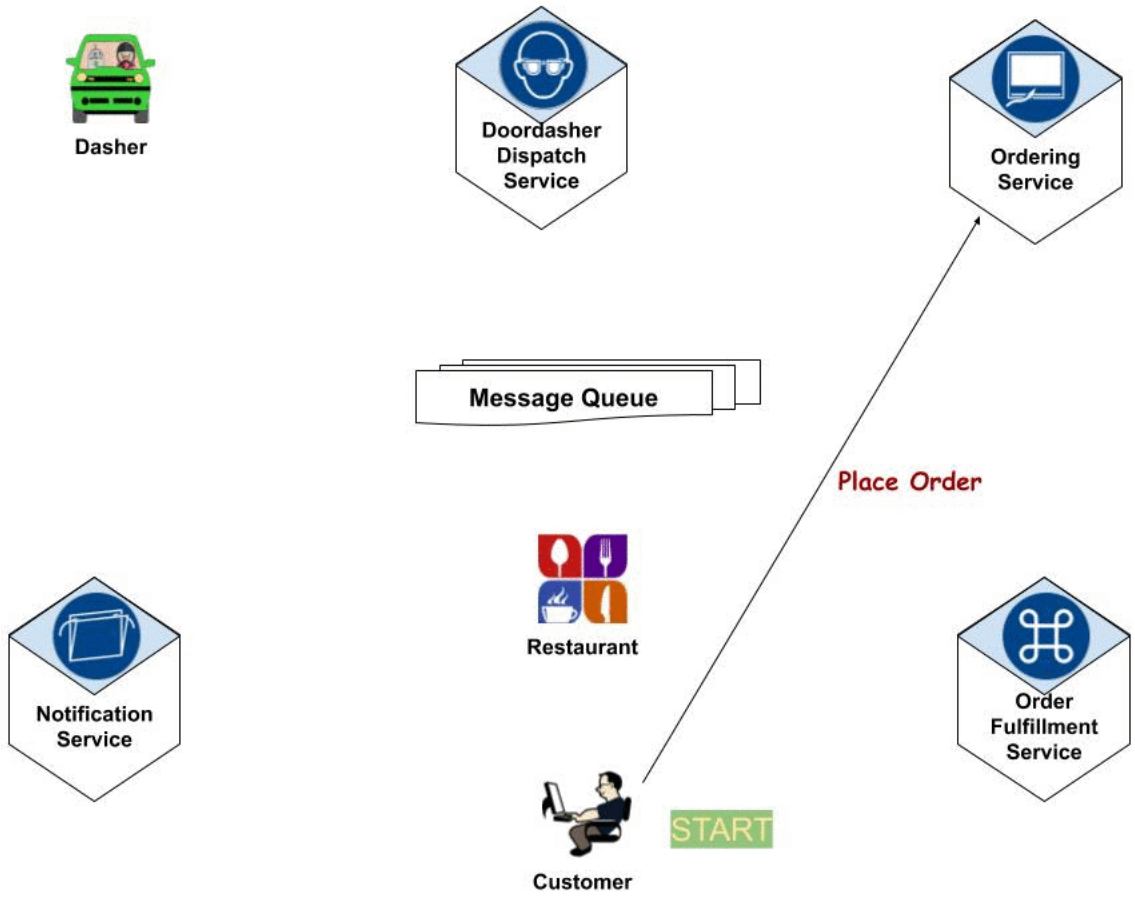
DoorDash System Architecture

Order fulfillment Workflow

Let's define some acronyms:

- ***Notification Service***: NS
- ***Doordasher Dispatch Service***: DDS
- ***Message***: MSG

Once the customer places an order using the mobile/web client using the OS, the order processing workflow could be as follows:



Order fulfillment Workflow

1. **OS** publishes a **MSG(#1)** to the queue about the order so that the downstream services can start processing.
2. **NS** reads **MSG (#1)** and notifies the corresponding Restaurant and the customer that an order is placed and pending their acceptance.
3. Once the order is accepted by the restaurant (using the ***OFS)***, **OFS** will publish a **MSG(#2)** to the queue regarding the acceptance of the order.
4. **DDS** will read the **MSG (#2)** and post an area-specific **MSG(#3)** to the queue. **DDS** could also auto-assign an order to an available doordasher and in which case it will post a related **MSG**.
5. **NS** will read **MSG(#2)** and notify the customer that the order was accepted by the restaurant.
6. **NS** will read **MSG(#3)** and notify the doordashers in the area about the availability of the order for their acceptance to process.
7. The doordasher will accept the order using the **DDS. DDS** will then post a **MSG(#4)** to the queue about doordasher assignment/acceptance.

10. **NS** will read **MSG(#5)** and notify the assigned doordasher as well as the customer that the food is ready to be picked up.
11. After the doordasher picks up the food, Restaurant staff will update the status of the order using **OFS**. **OFS** will then post a **MSG(#6)** to the queue that the order has been picked up.
12. **NS** reads **MSG(#6)** and notifies the customer that the order is picked up.
13. Doordasher reaches the customer and delivers the food and marks the order as complete! ***The food is here!! Yaaaay!*** :) **DDS** posts a **MSG(#7)** to the queue about delivery completion**.**
14. **OFS** reads **MSG(#7)** and updates the status of the order to **completed**.
15. **NS** reads **MSG(#7)** and notifies the restaurant and the customer about the delivery of food.

System APIs

Some of the customer-facing APIs exposed by the various services:

Ordering food:

- search (array of search terms)
- addToCart(menu-item)
- order(cart)
- status(orderId)
- retrieveOrder(orderId)
- cancelOrder(orderId)

Profile management:

- createProfile
- updateProfile
- updateAddress
- updatePaymentMethod

Sometimes the technical discussion could veer towards defining rest APIs, URIs, identifying HTTP methods, sample entities, sample payloads, request-response, etc.

Data Partitioning

As the data grows, it becomes impossible to store all the data in just once instance of the database. Restaurant data can be partitioned based on:

- Area code
- RestaurantId
- Menu items/cuisines/dishes
- Combination of area code & restaurantId

Each of the above partitioning schemes has its own benefits and disadvantages. The strategy needs to be well thought out considering all

Each service should be horizontally scalable. The NoSQL infrastructure will also have multiple nodes. The search system can have multiple nodes set up. Queues can have partitions and replication as well. Each and every service/component can be individually scaled if needed. ***Autoscaling*** could be enabled to handle loads on individual components, for example, spinning up more instances in the face of a high peak load. Furthermore, in case any of the nodes go down, or any partition in the queueing infrastructure goes down, another instance can take up the job. The crashed node can then do a cleanup and restart using a process called ***self-healing***.

Caching

Based on the recent orders in an area, or the most ordered items, or searched items, data could be cached so that the Restaurant Search Service could look up such information from a distributed cache, instead of hitting the search infrastructure, and immediately return few recommendations. Images of restaurants and dishes could be cached as well, instead of hitting the object storage all the time. The cache will hold the popular or most-ordered menu items/restaurants in a particular area and the search screen should show those options by default. The usage of cache definitely speeds up the viewing of search results as mentioned in this article.

Some of the popular cache technologies available in the market are ***Redis***, ***hazelcast***, and ***Memcached***. ***Least Recently Used (LRU)*** or ***Least Frequently Used (LFU)*** algorithm, or a combination of both can be our cache eviction strategy for our use case. This article describes caching in more detail.

A ***content delivery network*** (CDN) could also be used as a cache in order to make local content available to users based on their geography. In our current application scenario, using a CDN will be an overkill.

Security

- HTTPS/SSL -TLS for securing the wire. All communications between the web or mobile clients need to be on TLS.
- OAuth 2.0 for token authorization/refresh/generation.
- In case the queuing system is Kafka — SASL per topic/SSL.

Load Balancing

Every service will have multiple instances running, in scenarios where the communication is over https POST/GET, etc load balancers should be placed in front of the individual services. Load balancing helps make sure that no one instance is inundated with a deluge of requests and the response time is faster overall. There are several load balancing techniques that could be used like Least Connection Method, Round Robin method, etc. More can be found here.

In case the service solely subscribes to a particular channel/topic in the queue, the responsibility of load balancing is on the queue itself. For example, in Kafka consumer consumption divides partitions over consumer instances within a consumer group. Each consumer in the consumer group is an exclusive consumer of a ***fair share*** of partitions. This is how Kafka does load balancing of consumers in a consumer group.

Conclusion

System design interviews are very challenging and thought-provoking and really gives the interviewer a sneak peek into the technical acumen of the interviewee. The best way for preparing for this kind of interview is to start enjoying designing big popular day-to-day applications. Thinking about millions of users, huge datasets, points of failures, transactions, and so forth. I hope this discussion gives you some ideas about how you could go about tackling such interviews. Educative.io has several examples and this post, in part, is influenced by some of the examples presented there.

References:

- Wikipedia
- Kafka Architecture
- Educative.io — Grokking the System Design interviews.

Disclaimers:

- **This is exclusively my personal design exercise and only for demonstration purposes. I am not affiliated to DoorDash in any way.**
- **The design and the views expressed here are my own and are not from my employer.**

If you have reached this far, a BIG thank you! :) ❤I hope you enjoyed reading this piece. This is my first attempt at taking up a system design problem. Please feel free to suggest improvements to the design, or areas of potential problems/bottlenecks and how to avoid those in the comments. I will try to incorporate some design changes if required. Finally, thanks to ***Sunita Sharma*** *and* ***Tarun Bharti*** *for salient inputs on the data-model and flow diagrams.*