Question 1
Q: Uber has a lot of different services: there's the core ride-hailing Uber service, there's UberEats, there's UberPool--are we designing the API for all of these services, or just for one of them? A: Let's just design the core rides API -- not UberEats or UberPool.

Question 2
Q: At first thought, it seems like we're going to need both a passenger-facing API and a driver-facing API--does that make sense, and if yes, should we design both? A: Yes, that totally makes sense. And yes, let's design both, starting with the passenger-facing API.

Question 3
Q: To make sure we're on the same page, this is the functionality that I'm envisioning this API will support: A user (a passenger) goes on their phone and hails a ride; they get matched with a driver; then they can track their ride as it's in progress, until they reach their destination, at which point the ride is complete. Throughout this process, there are a few more features to support, like being able to track where the passenger's driver is before the passenger gets picked up, maybe being able to cancel rides, etc.. Does this capture most of what you had in mind? A: Yes, this is precisely what I had in mind. And you can work out the details as you start designing the API.

Question 4
Q: Do we need to handle things like creating an Uber account, setting up payment preferences, contacting Uber, etc..? What about things like rating a driver, tipping a driver, etc.? A: For now, let's skip those and really focus on the core taxiing service.

Question 5
Q: Just to confirm, you want me to write out function signatures for various API endpoints, including parameters, their types, return values, etc., right? A: Yup, exactly.

1. GATHERING REQUIREMENTS

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

We're designing the core ride-hailing service that Uber offers. Passengers can book a ride from their phone, at which point

- they're matched with a driver;
- they can track their driver's location throughout the ride,
- up until the ride is finished or cancelled;
- and they can also see the price of the ride as well as the estimated time to destination throughout the trip, amongst other things.

The core taxiing service that Uber offers has a **passenger-facing side** and a **driver-facing side**; we're going to be designing the API for both sides.

2. COMING UP WITH A PLAN

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

We're going to center our API around a *Ride* entity; every Uber ride will have an associated *Ride* containing information about the ride, including information about its passenger and driver.

Since a normal Uber ride can only have one passenger (one passenger account--the one that hails the ride) and one driver, we're going to cleverly handle all permissioning related to ride operations through passenger and driver IDs. In other words, operations like *GetRide* and *EditRide* will purely rely on a passed userId, the userId of the passenger or driver calling them, to return the appropriate ride tied to that passenger or driver.

We'll start by defining the *Ride* entity before designing the passenger-facing API and then the driver-facing API.

3. ENTITIES

ENTITY	DETAILS
1. Ride	• rideId: *string* • passengerInfo: *PassengerInfo* • driverInfo?: *DriverInfo* • rideStatus: *RideStatus* - enum CREATED/MATCHED/STARTED/FINISHED/CANCELLED • start: *GeoLocation* • destination: *GeoLocation* • createdAt: *timestamp* • startTime: *timestamp* • estimatedPrice: *int* • timeToDestination: *int*
2. Passenger Info	• id: *string* • name: *string* • rating: *int*
3. Driver Info	• id: *string* • name: *string* • rating: *int* • ridesCount: *int* • vehicleInfo: *VehicleInfo*
4. Vehicle Info	• licensePlate: *string* • description: *string*

4. PASSENGER API

The passenger-facing API will be fairly straightforward. It'll consist of simple CRUD operations around the *Ride* entity, as well as an endpoint to stream a driver's location throughout a ride.

API ENDPOINTS wrt PASSENGER	WHAT ENDPOINT RETURNS	USAGE
1. CreateRide(userId: string, pickup: Geolocation, destination: Geolocation)	Ride	Called when a passenger books a ride; a *Ride* is created with no *DriverInfo* and with a **CREATED** *RideStatus*; the Uber backend calls an internal *FindDriver* API that uses an algorithm to find the most appropriate driver; once a driver is found and accepts the ride, the backend calls *EditRide* with the driver's info and with a **MATCHED** *RideStatus*.

2. GetRide(userId: string)	Ride	Polled every couple of seconds after a ride has been created and until the ride has a status of **MATCHED**; afterwards, polled every 20-90 seconds throughout the trip to update the ride's estimated price, its time to destination, its *RideStatus* if it's been cancelled by the driver, etc.
3. EditRide(userId: string, [...params?: all properties on the Ride object that need to be edited])	Ride	
4. CancelRide(userId: string)	Void	Wrapper around *EditRide* -- effectively calls *EditRide(userId: string, rideStatus: CANCELLED)*.
5. StreamDriverLocation (userId: string)	Geolocation	Continuously streams the location of a driver over a long-lived websocket connection; the driver whose location is streamed is the one associated with the *Ride* tied to the passed *userId*.

5. DRIVER API

The driver-facing API will rely on some of the same CRUD operations around the *Ride* entity, and it'll also have a *SetDriverStatus* endpoint as well as an endpoint to push the driver's location to passengers who are streaming it.

API ENDPOINTS wrt DRIVER	WHAT THE ENDPOINT RETURNS	USAGE
1. SetDriverStatus(userId: string, driverStatus: DriverStatus)	Void	DriverStatus: enum UNAVAILABLE/IN RIDE/STANDBY Called when a driver wants to look for a ride, is starting a ride, or is done for the day; when called with **STANDBY**, the Uber backend calls an internal *FindRide* API that uses an algorithm to enqueue the driver in a queue of drivers waiting for rides and to find the most appropriate ride; once a ride is found, the ride is internally locked to the driver for 30 seconds, during which the driver can accept or reject the ride; once the driver accepts the ride, the internal backend calls *EditRide* with the driver's info and with a **MATCHED** *RideStatus*.
2. GetRide(userId: string)	Ride	Polled every 20-90 seconds throughout the trip to update the ride's estimated price, its time to destination, whether it's been cancelled, etc..

3. EditRide(userId: string, [...params?: all properties on the Ride object that need to be edited])	Ride	
4. AcceptRide(userId: string)	Void	Calls *EditRide(userId, MATCHED)* and *SetDriverStatus(userId, IN_RIDE)*.
5. CancelRide(userId: string)	Void	Wrapper around *EditRide* -- effectively calls *EditRide(userId, CANCELLED)*.
6. PushLocation(userId: string, location: Geolocation)	Void	Continuously called by a driver's phone throughout a ride; pushes the driver's location to the relevant passenger who's streaming the location; the passenger is the one associated with the *Ride* tied to the passed *userId*.

6. UberPool

As a stretch goal, your interviewer might ask you to think about how you'd expand your design to handle UberPool rides.

UberPool rides allow multiple passengers (different Uber accounts) to share an Uber ride for a cheaper price.

One way to handle UberPool rides would be to allow *Ride* objects to have multiple *passengerInfo*s. In this case, *Ride*s would also have to maintain a list of all destinations that the ride will stop at, as well as the relevant final destinations for individual passengers.

Perhaps a cleaner way to handle UberPool rides would be to introduce an entirely new entity, a *PoolRide* entity, which would have a list of *Ride*s attached to it. Passengers would still call the *CreateRide* endpoint when booking an UberPool ride, and so they would still have their own, normal *Ride* entity, but this entity would be attached to a *PoolRide* entity with the rest of the UberPool ride information.

Drivers would likely have an extra *DriverStatus* value to indicate if they were in a ride but still accepting new UberPool passengers.

Most of the other functionality would remain the same; passengers and drivers would still continuously poll the *GetRide* endpoint for updated information about the ride, passengers would still stream their driver's location, passengers would still be able to cancel their individual rides, etc..

DESIGN UBER API

Pool Ride
- driverInfo
- Rides [Ride]

Ride
- rideId : str
- passengerId : str
- driverInfo : DriverInfo → driverId : str, driverName : str, driverRatin double
- timeValues
- estimated ise
- rideStatus : RideStatus → created, matched, started, finished, cancelled

API wrt PASSENGER
- CreateRide(userId : str, pickup : loca^n, destina^n : loca^n)
  - → findDriver
  - → EditRide
- GetRide (userId : str)
- CancelRide (userId : str)
- Stream Driver Loca^n (userId)

API wr.t DRIVER
- DriverStatus (D.S.) → unavailable, in ride, standby
- SetDriverStatus (userId : str, driverStatus : D.S.)
  - → FindRide
    - → Ride/null
  - → AcceptRide (userId : str)
  - → EditRide
- Push Loca^n (userId : str, loca^n)