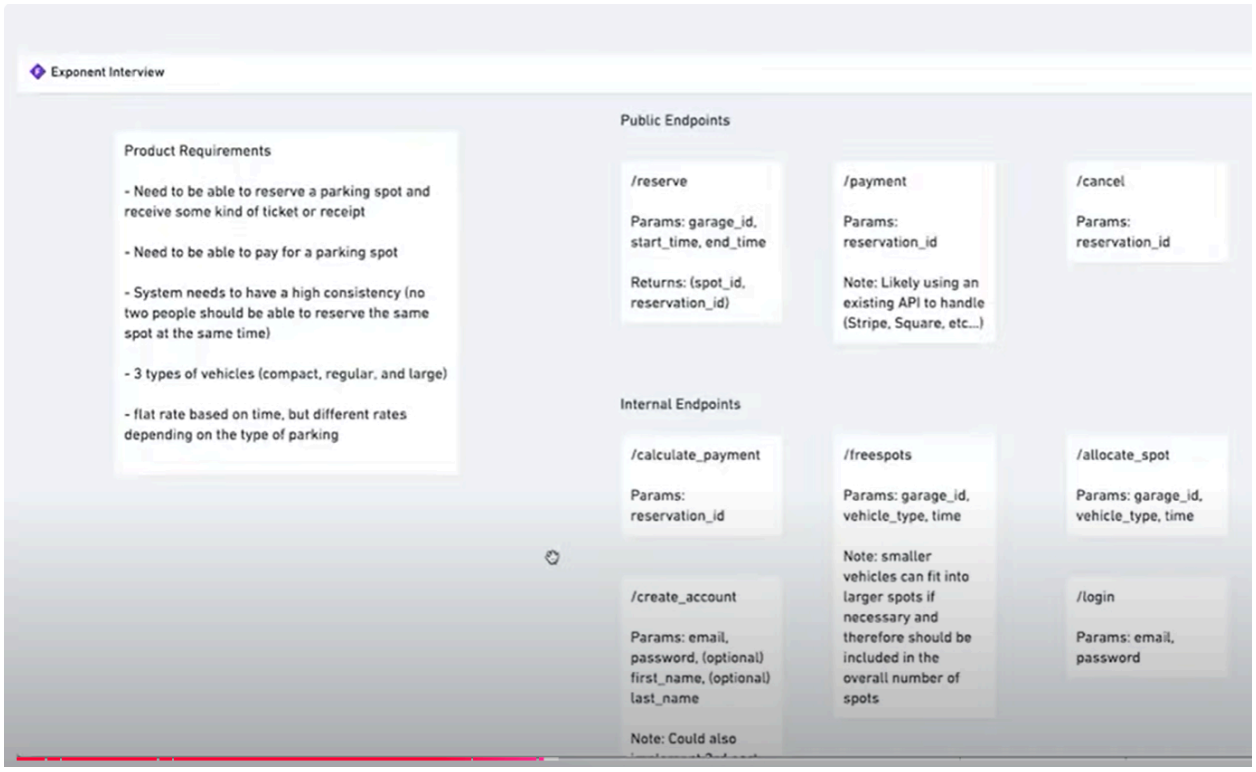
1. Requirements

The first thing i like to do generally is sort of clarify the requirements a little bit
just make sure i'm not making any like incorrect assumptions so to start off
i guess i'll write some of them down just to make sure I understand them
let's see so for this type of reservation system presumably we need
a way to be able to like reserve a parking spot and maybe receive some kind of i guess uh ticket or receipt.
You obviously need to be able to pay for the parking spot and then I guess this might be more of like a scale requirement but at a higher level if we think about this type of like product uh we'd probably prefer uh to err on the side of higher consistency because like if i'm thinking
about it we probably don't want two people to be able to reserve the same parking spot
obviously and then i'm curious are there different types of parking because if i think about like a normal parking garage you know there's like there's various there's like maybe a handicap spot
maybe some larger spots maybe some smaller spots so i think for for this question let's restrict it to three types of spots depending on the size of the vehicle so compact regular and large vehicles large.
And then for the are the prices for the garages would be flat rate let's say flat rate and but different rates for the three different types of parking and flat rate based on time. this
should be some form of i guess web app or mobile app interfacing with a back-end server so with fees i guess product requirements and requirements in mind.



2. Defining Api

Defining the sort of interface or the api we're going to be working with so i think this type of thing is going to have some public endpoints and then some internal endpoints so i'll maybe start us off with some of the more public endpoints so we need as we mentioned in the product requirements a way to reserve so let's call this one slash reserve and i'm just calling it this in reality it doesn't have to be like i'm immediately restful it could be like graphql or anything this is just so we have general understanding of what this should look like and so what should this type of thing take we obviously want to know which garage we're referring to. We want to know i guess when the time duration during which this should be reserved as well so that seems pretty reasonable so let's go with those something along those lines and then as we mentioned in sort of product requirements we should return some type of uh receipt or ticket so let's say it returns a tuple. Obviously we want the specific spot as well as potentially like a way to identify which specific reservation we're referring to so we just call that like a reservation id. There should be some form of payment endpoint and obviously this is probably beyond the scope of the specific problem per se but the actual implementation could probably just be handled by like a third party so yeah we want to pay by reservation id and then i want to say like you know you probably end up using something like stripe or square or like alternatively you can implement your own that's like a little more complex. You gotta be able to cancel your reservation
just because you know things come up in general so we should have a cancel end point as well let's call that here text left and yeah similarly you should be able to pass the reservation id and i think it should handle the rest. I think our public endpoints at least for the functionality that we're describing here we're probably going to have a few more endpoints on the internal app so let's get started on those and so let's taking a look at the public endpoints that we've sort of created here there's like a few internal things we might need one thing we need is a way to calculate the actual individual payments on the internal side yeah and debatably that could just be part of the initial endpoint but personally i prefer to separate them out so let's see here and yeah so similarly i should just take uh the reservation id because presumably and and this we can delve into this more later when we define our actual data schema but presumably the reservation id tells us like which specific garage we're referring to and then like what the actual times are that way we can sort of get the individual rates based on like what garage it is like what time yeah so that should be fine okay and then on the reserve side we probably need a
let's say a way to check what spots are currently available that seems pretty reasonable for from a ui perspective so let's say we have a free spots endpoints and one thing i guess when we're considering the parameters of what this specific endpoint might require obviously we need this the garage but one other thing that some other things that might be prudent are like
the type of vehicle or the time and the reason i say that is if we think about it the larger spots like you mentioned that over here we say we defined like there's a compact as a regular and there's a large the larger spots actually end up being able to be used for the smaller vehicles obviously we would prefer to not have to do that down the line like we would prefer to use up all the smaller spots but reasonably if all of those spots are full and someone wants to reserve a spot

they should be able to reserve a large spot at least in my opinion so there's just some things to note i guess and then there's some some of the more complex logic that's going to be held should be in the actual allocation of the spot we don't have to dive into the
minutia of the actual algorithm of that right now but for now let's just define that as let's say allocate spot and that should similarly take a garage idea vehicle type and a time
okay cool and yeah that seems pretty reasonable for the three we have here some of the more optional things i guess we could be adding if i'm thinking more about the actual product obviously the ability to create an account seems pretty reasonable if it's like a garage you're going to be end up using over and over we may want to incorporate that so let's say we have a create an account endpoint some things that we might need there are let's say an email and a password seem like pretty reasonable for the initial things and obviously we do some hashing for
that we don't want to store that in plain text and then some more optional things like let's say first name or like last name yeah yeah just as another thing to like note optionally you can also just implement third party like single sign-on options like google or facebook login and their payment details i'm assuming are gonna be taken care of by whatever third parties handling that exactly
and then let's say we just have like a login endpoint just because you need that once you've created an account cool so i think that seems pretty reasonable at least on the first pass for the specific system interface we're working with so let's maybe consider the actual like scale or scope of the problem and we kind of we tackle that briefly when we discuss how we want like high consistency given the nature of this problem but let's look at like the actual like size or like far reaching extent it's honestly my initial impression is that we're not likely to deal with like many at scale issues with this type of problem like on a per garage basis like even if we upper bound a garage optimistically at like 10 floors like i don't think the average parking garage is going
to have 10 floors but let's upper boundary just for the sake of this problem and let's say there's maybe like 200 spots per floor that's like 2 000 relevant spots per garage and the system like i guess what 2 000 relevant spots per garage i don't think there's that many parking garages in the world so i'm not particularly concerned about designing this as a more distributed system like i think even if there's like a million parking garages i don't like that's still not within the realm of like big data in my opinion so i don't think that really factors into the considerations here
um that being said as we mentioned before we care most about the system being up to date we don't really want to run into a race condition where like we happen to allocate to the same spots at the same time so those are just something to keep in mind

3. Data Schema

Data schema side of things how we're going to design the database so i guess what are some of the first things we probably want to tackle here reservations are obviously the point of this entire product so we can start with that let's go with reservations create a table here cool and to start us off we have i guess as most tables will we have an id and this is our primary key we can define it as a serial within most relational databases and i apologize the i tend to lean towards relational databases for most of this type of problem reasonably speaking you could probably go

into more of a no-sql type solution but just more of my expertise lies in the relational database side so presumably like some type of postgres or mysql is what we'll be working with here in the same thing so i'll be designing the data schema with that sort of perspective in mind okay that's totally fine awesome so yeah uh back to the reservations initial field id the serial that way it sort of increments as you add more and more it seems totally reasonable what else would we want so the other information we'd want in this sort of table we need like which specific garage it's referring to which spot we actually ended up allocating to it and then the start and end time seem like pretty reasonable things as well as also whether or not i guess this has been paid depending on like how you want to format the product you might be able to pay like on site as opposed to like in the app per se so that sort of status might be reasonable yeah i think that's this garage id and i think we can presumably we're going to be making a table down the line with this so we'll call this foreign key as well then let's see we have the spot id
and another thing about it's probably not unreasonable to make a spot table as well that way it could sort of have the potential vehicle types and whether or not it's like reserved at a specific time yeah so let's go with that also and then yeah we want start time we want the end time and uh yeah i guess it's just paid or not paid so we just call that paid cool see that seems pretty reasonable uh for a reservation table and obviously we can end up adding like user id or such if we end up having those but like for now this is like the important part yeah i think this is good to start with let's i guess move over to creating a garage table or garage table there's this whole debate over whether or not you should be using singular or plural and table names honestly i'm probably in the singular camp even though i just wrote reservations but i don't think it matters either way for the sake of this problem that's fine yeah yeah so this will also obviously have an id similarly merit key and serial so what are some of the things we want for the garage where it is it's i guess something that's generally pretty important and that can also help down the line when it comes to actually like starting specific like replicas of the database or load balancing things like that just an easy way off the bat to do that to handle that is a zip code yep that's pretty universal uh so let's go with that and to be able to incorporate i guess the extensions of zip codes that requires a hyphen so maybe let's go with uh bearcare instead of like an in and then what else do we want oh we want the uh the individual rates we mentioned that this is flat rate based on size yes so for now yeah it seems relatively reasonable to have a size
and then obviously these are like nullable uh columns in the case that they don't have that specific type of spot in the specific garage yep and one thing to note here is i'm using decimal i know there's obviously some issues uh down the line regarding like floating point errors and calculations and payments i think decimal is the postgres type that should be able to handle that i'm not confident that there's an equivalent mysql type but i'm sure there is so i think that seems like a pretty reasonable approach i think that's fine and then let's go with spots like we mentioned we have a spot table that is here similar to the other ones we want id with a primary key a serial and now what do we want for the spot we want the relevant garage obviously we want the type of vehicle and then we want i guess the specific status like what is it like maybe it's being reserved maybe it's like unavailable right now like maybe they're like up like i don't know paving over the spot or maybe it's just totally empty so
i think those are pretty reasonable things to have there okay so the garage id obviously it's a foreign key and i guess it's a secondary key and then we want the vehicle type and uh for now i'm gonna call this an enum and i'll get into this a little later when we actually

define oh actually i can get into this now i guess there's sort of like a trade-off there when you end up adding enums particularly in like a relational database just simply because so you you gain you become more performant because you have all the benefits of sort of being an int as opposed to like a you know like a character string or whatever but the trade-off there is uh there's some like flexibility issues down the line i know in postgres specifically when you add an enum you can never uh remove it like you can never remove the type afterwards
um so that's like something to consider when you end up choosing them i think for this specific case like compact regular and large i don't think those are things that are like ever going to be like totally out of phase in a parking garage type system so i think it's totally reasonable yeah i mean i'm here but like i mentioned before i don't think we're going to run these issues at scale here it would be totally reasonable uh to be like just a varicare as well for like flexibility and then sort of have that vetting on the application layer side yeah i think that's fine you can make it make a call over here on the trade-off that you're willing to go for for sure cool and then what else oh we had the i guess supposedly the optional table given that we optionally want to be able to create an account so let's say we have a users table and uh that should just have you know all the familiar things we have the id with the uh primary key it's a serial and then as we mentioned uh in the previous endpoints that were optional maybe an email or i guess fair care because we're in postgres and then okay as well and uh note that this is probably like something conventional let's say like a shot 256 hashing like that okay and then uh yeah first name obviously nullable as well and last name cool and then i guess arguably uh if i'm thinking about like the nature of this type of application the whole reason we wanted uh and optional users in the first place is just uh so they could like recurringly reserve a spot and if we want to be able to recurringly reserve a spot we probably also want the type of vehicle actually we might want the type of vehicle in general so i think a vehicle table is like a totally reasonable thing to have in this sort of scenario yep i think that's mine and then let's go with id as usual and then uh yeah so optionally like a user id that can be a foreign key and and then let's say we want their license seems reasonable and then yeah probably the vehicle type again similar to the one that was in the spots table so yeah i think that probably covers our requirements here i mean obviously i can add more as we go through the problem but for now does that seem like a i think this seems good yeah this looks reasonable maybe we can dive into what the

Exponent Interview

Data Schema

Reservations

id	primary key, serial
garage_id	foreign key, int
spot_id	foreign_key, int
start	timestamp
end	timestamp
paid	boolean

Garage

id	primary key, serial
zipcode	varchar
rate_compact	decimal
rate_reg	decimal
rate_large	decimal

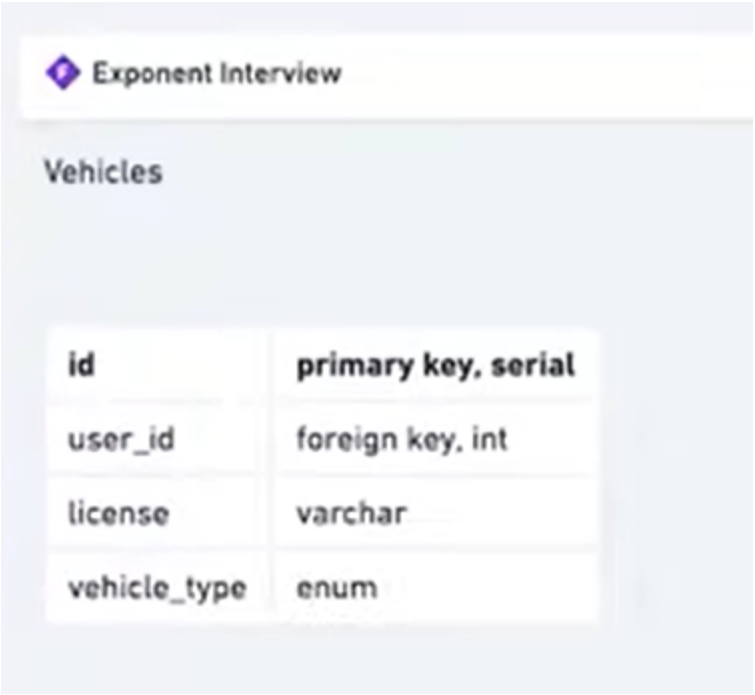
Spots

id	primary key, serial
garage_id	foreign_key, int
vehicle_type	enum
status	enum

Users

id	primary key, serial
email	varchar
password	varchar (note that this is probably SHA-256 hash)
first_name	varchar
last_name	varchar

Vehicles



4. High Level Architecture

high level architecture would look like yeah for sure so yeah we can sort of get into that here we start us off with you know the very basics we want a like a
application ui i mean it can be either web or mobile and then we want some sort of like back-end server so let's say here we have a
i don't know web app or mobile ui depending on the sort of me or maybe even both and then yeah over here let's go with a sort of a back-end server yeah
this will talk to this vice versa cool now some of the more the deeper considerations to have i guess here are obviously we're gonna have a
database as we mentioned before the one that i end up leaning towards is sort of like a postgres db so let's just call this ep and then go here and so
that's just like off the bat in reality we can end up doing a number of different adjustments here i guess one thing that we can consider
is that a potential consideration is splitting the back-end server into like a reservation server and a slot server given the complexity of like
allocating slots that seems probably a little unnecessary here a bigger thing that might be more reasonable is given that we sort
of described up here the uh that the system needs to have a high consistency we notice that in this type of problem we probably want to be able to read
more like way more than write so if we're going to be reading a lot more than writing we probably want a number of different read replicas for the db so we can
actually let's say separate out this here move it up this way and then can i just copy this oh yes i can awesome so yeah let's uh yeah let's call this a

read replica i'm going to read replica so yeah the server would kind of work like this essentially go here and uh one thing to have or that would be prudent
to have in this sort of scenario given that
we're probably hosting like a number of read replicas we probably want some form of load balancer in between just to sort of allocate where the individual
calls are going it would probably be prudent for this type of problem to shard that based on location i
know that we mentioned earlier like the garage might have like a zip code that seems like a pretty reasonable way to approach that so let's yeah maybe
this here and then from here we can go up here call this read call this read um
then this one will be right yeah and then what else uh we mentioned oh go for it i was gonna ask do you anticipate any any trade-offs based on this decision
yeah well not specifically the read replica portion one thing that i guess is likely going to end up being a specific trade-off as i was sort of a
pathing towards we mentioned that uh this type of thing uh sort of wants a high consistency so i think it's like pretty reasonable to take a strong consistency approach versus
like an eventual consistency
approach just simply because if we think about the nature of someone using this application like from a customer perspective it's likely not the case
that they're going to be super frequently reserving things like probably not more than like even twice a day at most and
that already seems a little excessive so in reality they're probably very willing
to wait or deal with like a higher latency which is i guess the trade-off you make when you have uh
strong consistency versus like eventual consistency where you like increase the performance i guess the execution on the uh the customer side like in the
customer's view but the trade-off you make there is like you might not have the exact data we want exact
data because we want them to be able to know that the spot that they're reserving is like specifically the spot that they'll
get and the way that that's tackled in this sort of setup is you basically read luck that's like an initial
approach that a lot of people use seems pretty reasonable as well um
especially for something that's not necessarily at an aggressive scale i'm in the sense that when someone's writing
to the postgres db uh you read lock the replicas that way they sort of wait for the result to finish that way you know the data is up to date as up-to-date as possible one potential consideration the
future of that might be depending on how you wanna split up and charge your database you might uh wanna consider uh the logic of read locking based on location because we think about it
um why would we read luck i don't know something in wisconsin if like someone's reserving something in california but that's like a little more of a

complex decision down the line okay yeah i think that's that's a good that would be a good next step cool and then yeah i
think that's oh one other thing uh we mentioned up here that we you know we have like a payment since likely to be using an existing third-party payment so yeah we just draw that in here let's say it's like uh place that here
have that here both ways don't really know how to do this oh there we go cool nice okay i think this this is pretty good and a good place to stop
what did you think about that tim ah it's uh it's sort of an interesting weird problem i guess i think a lot of these types of problems end up actually
having issues at scale so i think i was lucky in the sense that i i think it's reasonable to say with some like back of the envelope for me estimation that this uh
sort of problem doesn't tackle that and that made it a lot more of i guess how do we design the data problem got it