Design a ticket sales site where people can buy tickets to sporting events and concerts.

Like Ticketmaster.

To get started, let's ask our interviewer some clarifying questions to figure out what's in scope.

What should we ask?

Features

Are people buying specific seats? Or just general admission? For now, let's focus on the case where they're buying specific seats.

How many events are we selling tickets for? Come up with a reasonable estimate.

How many tickets are associated with each event/attraction? It depends. Some venues are quite small, with 100 to 1,000 seats. Others are huge, with over 50,000 seats.

How quickly do tickets sell? In general, about half of the tickets sell within the first few minutes, and the rest sell gradually over the next few weeks. But for some of the most popular events all the tickets sell out within 10 minutes, and competition for seats is fierce.

Can users sell tickets on our site? Or just buy them? To keep things simple, we'll only support buying tickets for now.

Do we store user profiles on our site? Or just ticket information for events? To make things easier for users, they have the option to create a user profile for storing their payment information.

How will tickets be validated at the event? Ultimately, we'll want some sort of barcode that can be scanned and checked at the event entrance. For now though, let's focus on designing the part of the system that deals with purchases.

Okay, so here are the specifics. We're building a scalable **ticket reservation site** where users can:

- browse events, select seats, and buy them, and
- optionally maintain user profiles for storing payment information.

Now that we've outlined what sort of things our system has to do, let's crunch some numbers for how large the system is going to be.

How many events do we have? How many tickets are we managing?

Estimate System Size

How many events?

Since we're selling admission tickets by date, our events are time bound. Let's say we allow users to buy tickets up to three months in advance.

As a rough estimate, let's say we have 200 attractions on any given day. That means we'll be selling tickets for roughly $200 * 30 * 3 \approx 18,000$ events at a time. Just to keep the numbers round and make sure we can handle some future growth, let's say 20,000 events.

How many ticket sales?

Let's say a small event has 1,000 tickets and a large event has 50,000 tickets. Let's say a quarter of our events are large.

That means over three months we'll be keeping track of 5,000 large events (250,000,000 tickets) and 15,000 small events (15,000,000 tickets).

Observation 1: Larger events completely dominate our traffic. So, let's focus on making larger events work, since the smaller events are kind of like a drop in the bucket comparatively.

With that in mind, we're focusing on managing 250,000,000 tickets per month, or roughly 3 million tickets per day. Let's round up to 4 million, just to make sure we're on the safe side.

Observation 2: Our traffic won't come at a constant rate. When a popular event comes online, we can expect a flood of traffic as people scramble to get seats. Just to get some numbers, we'll say a quarter of our events are "popular" and sell out within 10 minutes. That means we need to be prepared for 1 million ticket sales in 10 minutes, which is 100,000 ticket sales per minute or roughly 1,600 ticket sales per second.

Those are some large numbers, and it's easy to get overwhelmed.

Instead of tackling such a large system off the bat, let's do this in two steps:

1. First, we'll focus on designing a minimum viable product (MVP) that'll handle a small number of users and events.
2. Then, we'll scale up our prototype to handle larger numbers.

Okay, to get started, what's the bare minimum we'll need?

Minimum Viable Product

First, we'll have a **web server** that handles incoming requests from our user. These requests could be to list all the events, choose a specific seat, log in, or buy seats. We'll also need some sort of **database** to track information about the event, like its date, which seats are available, and ticket prices.

We could draw these components out on a whiteboard:

> Drawing this out might feel kind of silly. Isn't it obvious we'll want a server and a database?
>
> Maybe, but it's still worth it to establish the basic components that make up your system before diving in. It gives you some structure to work with moving forward and helps your interviewer understand what you're picturing so far.

Let's dig into the details of the database. What kind of information do we need to store? And how should we organize it? Let's run through a few user requests to figure out what tables and fields our database needs.

Data Model

First, we need to be able to store basic information about events. When a user comes to our site, we'll want to show them a list of upcoming events.

Events
int event_id
TIMESTAMP event_data
char[256] event_name
char[256] event_venue
char[2048] event_description
2,572 bytes

> Where did those numbers come from? They're rough guesses.
>
> 256 characters is about two lines of text. That should be enough to hold an event's name and venue.
>
> 2,048 characters is about a half a typed page, which should be long enough to describe an event.
>
> Add in 8 bytes for the timestamp and 4 bytes for the event ID and you'll get the total size of a record in our database: 2,572 bytes.

TABLE EDIT: should be event *date*. (Applies to all.)

Once a user chooses a specific event, we'll want to show them available seats and prices:

Events	Seats
int event_id	int seat_id
TIMESTAMP event_data	int seat_section
char[256] event_name	int seat_row
char[256] event_venue	int seat_number
char[2048] event_description	double seat_price
2,572 bytes	24 bytes

Wait a second though. We need a way to indicate which seats are available. Let's add a `bool seatAvailable` to keep track of that.

Events	Seats
int event_id	int seat_id
TIMESTAMP event_data	int seat_section
char[256] event_name	int seat_row
char[256] event_venue	int seat_number

char[2048] event_description	double seat_price
	bool seat_available
2,572 bytes	25 bytes

Also, we only want to show users seats for the event they're interested in, not all events. Let's add a field linking seats to specific events: `seatEventId`.

Events	Seats
int event_id	int seat_id
TIMESTAMP event_data	int seat_section
char[256] event_name	int seat_row
char[256] event_venue	int seat_number
char[2048] event_description	double seat_price
	bool seat_available
	int seat_event_id
2,572 bytes	29 bytes

Now we're getting somewhere. With this database setup, we can easily find events that haven't happened yet, and once a user picks a specific event, we can pull up all of its available seats.

What happens when a user actually chooses a seat?

We'll need to get the user's payment information. Once we have that, we can charge them, mark the seat as sold (by clearing `seatAvailable`), and finalize the purchase.

> What happens if two people try to buy the same seat at the same time?
>
> Interviewer: We definitely need to handle that. Let's build out the system a bit more first, and then circle back and make sure we cover that.

Okay, how should we store purchase information in our database?

There are a few different ways we could do it:

Since each seat is purchased at most once, we could add purchase information into the `Seats` table:

Events	Seats
• int eventId • TIMESTAMP eventDate • char[256] eventName • char[256] eventVenue • char[2048] eventDescription	• int seatId • int seatSection • int seatRow • int seatNumber • double seatPrice • bool seatAvailable • int seatEventId • char[32] seatPurchaseCreditCardNumber • int seatPurchaseCreditCardExpiration • int seatPurchaseCreditCardCvv
2,572 bytes	69 bytes

> Storing credit cards in our database is a **huge** security liability. If we're hacked, we could compromise financial information for millions of users. There are online payment tools, like Stripe, that let us sidestep this issue and avoid storing our users' financial data. (That's what Interview Cake does.)

This could work, but it's inefficient. What if the same person buys 100 tickets? We'll be storing their credit card information 100 times.

Ideally, we'd like to store each user's payment information exactly once. So instead, let's add in a new table for storing user information, and we'll add a field for the user's ID in the `Seats` table:

Events	Seats	Users
• int eventId • TIMESTAMP eventDate • char[256] eventName • char[256] eventVenue • char[2048] eventDescription	• int seatId • int seatSection • int seatRow • int seatNumber • double seatPrice • bool seatAvailable • int seatEventId • int seatPurchaseUserId	• int userId • char[64] userName • char[32] userPasswordHash • char[32] userCreditCardNumber • int userCreditCardExpiration • int userCreditCardCvv
2,572 bytes	33 bytes	140 bytes

Another option would be to store purchase information separately from seating information in a fourth table:

Events	Seats	Users	Purchases
• int eventId • TIMESTAMP eventDate • char[256] eventName	• int seatId • int seatSection • int seatRow • int seatNumber • double	• int userId • char[64] userName • char[32] userPasswordHash • char[32]	• int purchaseUserId • int purchaseEventId

• char[256] eventVenue • char[2048] eventDescription	• double seatPrice • bool seatAvailable • int seatEventId	• userCreditCardNumber • int userCreditCardExpiration • int userCreditCardCvv	• int purchaseSeatId
2,572 bytes	29 bytes	140 bytes	12 bytes

Both options have their pros and cons. Can you think of reasons to choose one over the other?

There's not a clear winner.

- If most of our events are sold out, then linking each seat to its user is more space efficient—we're only storing 4 bytes per purchase instead of 12. But, if *lots* of seats are unsold, then this field won't be used most of the time, which is a waste of space.
- Keeping a separate `Purchases` table is a bit more flexible. We didn't ask if our system needs to handle returns and refunds, but if it did, then we could have more than one purchase for each seat if the seat is purchased, returned, and purchased again. Breaking out purchase data into its own table helps us keep our options open.

Between space efficiency and future flexibility, we'll choose the second option.

> Does every buyer *have* to have a user account?
>
> When talking through our early requirements, we decided that profiles would be optional: "To make things easier for users, they have the option to create a user profile for storing their payment information."
>
> This suggests some people will have user accounts, but others might not. How can we handle purchases from people who don't have accounts? (Assuming we don't want to force everyone to register.)
>
> One option would be to make the `purchaseUserId` field `NULL` in those cases. But that could be tricky if we need to look up payment information for an earlier purchase (to refund it, for instance).
>
> Another option would be to create a new entry in the `Users` table that stores the payment information and assigns a unique `userId` but leaves the `userName` and `userPasswordHash` fields blank. We'd record this **"anonymous" user account** as the user who made the purchase. This would allow us to track payment information for all purchases, even those where the user hasn't actually made an account.
>
> So, does every user need to register for a user account? No. But all users who purchase end up with an account—either the one they already made, or an anonymous one to record their purchase.

Okay. This is starting to feel like a minimum viable product. To make sure we covered everything, let's run through the steps of a ticket purchase.

Here are the steps to buying a ticket:

- User navigates to our site, and chooses one of the listed events.
- User picks an available seat at that event.
- User either logs in (and we'll pull their payment information from their account), or enters in their payment information and confirms the purchase.
- We mark the seat as sold, issue the payment, and update the `Purchases` table.

Putting all that together, here's an updated picture of our system so far:

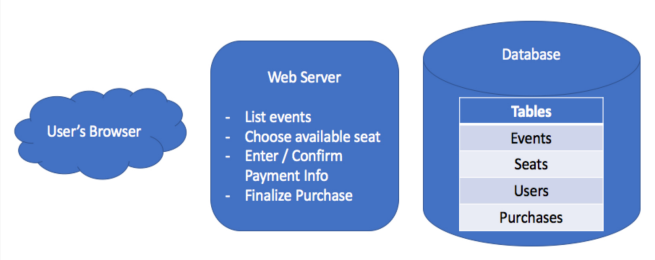


IMAGE EDIT: Browser window for user's browser, cloud for internet.

IMAGE EDIT: Third bullet for web server text should read "Confirm purchase" (no period). Fourth bullet should read "Update databases" (again, no period.)

Looks good.

Now that we know we have the basic logic down, let's try to build up our system to handle more users. Based on our earlier estimates, we can expect about 1,600 ticket requests per second, so we'll need something scalable. What are potential bottlenecks or issues?

To start, let's come back to a question we put off earlier. What happens if two users try to buy the same seat?

Scaling Up: Handling Multiple Users

In many cases, one user will finish their purchase before the second user starts:

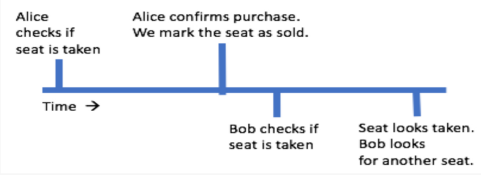


IMAGE EDIT: Replace "Alice" with "Alicia". Replace "Bob" with "Juan". (Applies to all timeline images.)

But there's actually a **race condition** in our logic that can cause one seat to be sold twice. Take a

look at this ordering:

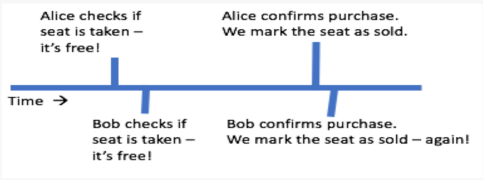


IMAGE EDIT: *"Alice / Alicia enters payment info and confirms purchase. We mark the seat as sold."*

That's really bad. We sold the same seat twice.

How can we handle this?

There are a few different options. Here's a simple one: when someone submits their payment info to purchase a seat, do a last-minute check to see if we already sold the seat. That would prevent us from selling Juan the seat we already sold Alicia:

NEW IMAGE: *timeline like the above, just different copy on the last item: "Juan submits payment info —whoops, we've already sold that seat! Don't charge Juan's card, and tell him he has to pick another seat."*

We'd just need to update the logic for accepting payments:

```
when the user submits their payment for a seat:
    if the seat is marked as available in the seats table:
        charge their card
        and mark the seat as unavailable
    else:
        redirect the user to the seat selection page
        and say "Sorry, that seat's already taken!"
```

This'll get us *most* of the way there.

There's still a potential issue if two people actually submit their payments *super close to each other*. Both users could hit this line at the same time:

```
if the seat is marked as available in the seats table:
```

They'd both pass that check, continue into the block, and charge the user's card. So we'd be selling the same seat twice again. Oof.

What database feature do we need to prevent this?

We need our database to make sure that:

- Only one user can update the entry for a seat at a time, and
- Everyone accessing the `Seats` table sees the update.

To get these features, we'll need an ACID- database with **transactions** around our logic that manipulates the `Seats` table.

```
when the user submits their payment for a seat:
    start a database transaction

    if the seat is marked as available in the eats table:
        charge their card
        and mark the seat as unavailable
    else:
        redirect the user to the seat selection page
        and say "Sorry, that seat's already taken!"

    end the database transaction, committing the changes
```

The way we've structured our data model—focusing on relationships between entries in different tables—suggests we want a relational database, not a NoSQL one.

> If you're familiar with specific databases, feel free to name which database you'd use. We're a big fan of Postgresql. It's a well respected open-source relational database.

> ACID vs. BASE. SQL vs. NoSQL. What's the difference?
>
> It's easy to confuse these distinctions. Don't mix them up in your interview.
>
> ACID and BASE describe the **guarantees** a database provides. An ACID database provides strong guarantees about data consistency and integrity. A BASE database provides weaker guarantees in order to gain flexibility and scalability.
>
> SQL and NoSQL describe how data in the database are structured. SQL databases (aka relational databases) organize data in terms of tables, entries, and relationships. NoSQL is a catch-all term to describe everything that's *not* relational.
>
> Most SQL databases provide ACID guarantees. And many NoSQL databases are characterized as BASE.
>
> But some NoSQL databases provide ACID guarantees, too!

One thing though: ACID databases are difficult to scale across many machines.[3] That could make things more complicated if we're trying to store more data across multiple machines. Can we fit our information fit on one beefy database server?

Interviewer: **"Let's come back to that question later. I think there are still a few details about the reservation system to iron out. How can we improve the purchase process?"**

> Some interviewers like to jump in and steer the discussion one way or another. It doesn't mean you're making a mistake—it just means they want to do things in a certain order. Go with the flow and follow their cues.

With our current system, sometimes a user will enter their payment information only to find out the seat has been sold to someone else.

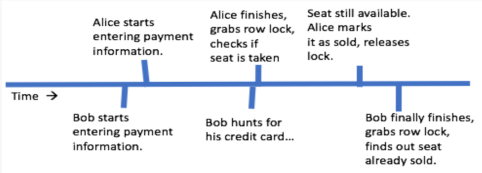


IMAGE EDIT: No references to row locks. "Alice starts transaction, checks if seat is taken" "Alice marks it as sold, commits transaction" and "Bob finally finishes, starts transaction, finds out seat already sold"

That's not very user-friendly. Most people would probably think they'd gotten a seat (since they weren't prompted to pick a different one) only to be told later on that someone else beat them to it. (It also doesn't seem particularly fair. In the timeline above, Alicia got the seat even though Juan picked it first!)

One way to fix this is to mark the seat as sold earlier on, as soon as a user *selects* a seat.

That way, when you're prompted for your payment information, you're guaranteed nobody else can snatch it from you.

Interviewer: **"That seems promising. Any problems you can think of with that approach?"**

Interviewer: **"What happens if a user selects a seat but never finishes their purchase?"**

Hm. That's a good point. We'll have marked that seat as sold (so nobody else can buy it), but we won't actually have sold it. We'll be leaving money on the table!

How can we fix this?

We could add in a **timeout**. Once a user selects a seat, we'll block anybody else from choosing the same one, but only for a short amount of time (say, 5 minutes). After that, the user's hold on the seat goes away, and they're no longer guaranteed the seat.

How could we implement purchase timeouts?

Well, before, we assumed our seats were either available or purchased. Now, we're recognizing that seats could also be *reserved but not yet purchased*. And we're making sure a seat doesn't stay in this reserved-but-not-yet-purchased state for too long.

What changes do we need to make to our tables? First, instead of having a boolean value (`seatAvailable`), it would be helpful to have a field that specifies the status of the seat. A simple integer—call it `seatStatus`— should do the trick. We'll set it like this:

- 0, to mean "Available for purchase"
- 1, to mean "Reserved, but not yet purchased"
- 2, to mean "Purchased"

Also, if we're going to have a timeout on state (1), we need a timestamp that specifies when the seat was chosen by the user (it'll be `NULL` if the seat is available or purchased). Let's call that field `seatSelectionTime`.

Here's what our `Seats` table looks like now:

Seats
• int seatId • int seatSection • int seatRow • int seatNumber • double seatPrice • ~~bool seatAvailable~~ • int seatEventId • int seatStatus • TIMESTAMP seatSelectionTime
36 bytes

> Can we get rid of either `seatStatus` or `seatSelectionTime` to save space?
>
> No, we need both. We could use the reservation time to implicitly tell us whether a seat is reserved or not, but that doesn't tell us if its been *purchased*.

Now, how should we go about actually *enforcing* reservation timeouts?

One option would be to periodically run through all the entries in the `Seats` table, find all the entries with a `seatStatus` of 1 and a `seatSelectionTime` more than 5 minutes ago, and clear the reservation. Like this:

```
for all seats
    if both - seat status is 1 (reserved but not purchased) AND
            - seat selection time is more than 5 minutes ago, then
        - set seat reservation time to null
        - set seat status to 0 (unreserved)
```

Another option is to let seats linger in the "reserved but not yet purchased" state until someone

else tries to select the seat. Then, we'll let them select the seat if it is:

- Available (with a `seatStatus` of 0), or
- Reserved (with a `seatStatus` of 1) but expired (`seatSelectionTime` is more than 5 minutes ago)

Both of these options are reasonable. The first one could be preferable because it'll keep our database tidier (we'll have fewer seats in the intermediate reservation step). But it also adds extra overhead to the `Seats` table due to the scanning step, which requires making extra queries and updates that aren't strictly needed. The second option avoids this overhead entirely, only timing out tickets that have been requested by someone else.

> Won't we get faster lookups if we're just using the status field? Isn't it more expensive to do queries on `TIMESTAMP` fields than `int` fields?
>
> It depends on how precise our timestamps are. MySQL stores timestamps as 4 byte integers, with extra bytes for fractional second precision. If we're okay with 1 second granularity on our timestamps, then working with them is just like working with integers.

As a starting point, let's try to minimize overhead and go with the second option.

Interviewer: **"Okay, thanks. That should work. Let's come back to your earlier train of thought about whether our database would be too big for a relational ACID database."**

Well, first off, how big are each of our tables?

Scaling Up: Databases and Storage

`Users`: In the schema above, each user takes up 140 bytes. Let's say that, on average, each user buys two tickets a month. If we're selling 100,000,000 tickets per month, that's 50,000,000 users. (For scale, that's *roughly* 20% of people in the US who have internet access.) That's about 7 gigabytes of user data—not a whole lot.

`Events`: Each event is about 2,600 bytes in the above schema. We have 18,000 of those at a time, so we're generating a mere 48 megabytes of event data every three months. We can definitely keep event information around for a long time before worrying about space constraints.

`Purchases`: Each purchase is 12 bytes, and we have 300,000,000 of those every three months. That's just over three gigabytes of purchase data. Again, not too big.

`Seats`: Each seat is 29 bytes in our database, so that's about 9 gigabytes of data every three months.

Okay, so none of these tables are large. In fact, compared to a service like Facebook, we're storing a *tiny* amount of data. We should be fine using an ACID database, and we can fit all of our data comfortably on solid state drives (SSDs).

In fact, we might even be able to store most (maybe even all) of this data in memory (RAM), periodically flushing the data to persistent storage on an SSD.

That said, even if space isn't an issue, we should consider whether our SSDs will be fast enough to handle all our incoming queries.

Remember, we need to be able to handle around 1,600 transactions per second at peak times. Will our SSDs be fast enough for our load?

If we were buying from Amazon, then we could use general purpose SSDs and get around 3,000 input/output operations (IOs) per second. If we pay a bit more, we can get specialized storage that has up to 30,000 IOs per second.

Each transaction is one or two IOs: we have to (1) read the `Seats` database to check if a seat is still free and, if it is, (2) update the `Seats` database to mark the seat as taken.

If we're targeting 1,600 transactions per second, we could come close to that limit of 3,000 IOs per second. So, we can start off with general purpose storage, but we should keep an eye on the database to see if we need to pay for additional IOs.

Okay, so it looks like we can store our tables on SSDs. And our tables aren't super big, so we don't need to do anything fancy like sharding to get them to fit. So, we'll have a SQL server with four SSDs—one for each of our tables (`Users`, `Purchases`, `Events`, and `Seats`).

Interviewer: **"That seems reasonable. How could we speed up our database though?"**

Scaling Up: Database Optimizations

For tables that are frequently read, we can add a **caching layer**, like Memcached, to make retrieving popular records fast. Which tables are frequently read? The `Events`, `Seats`, and `Users` tables are queried on every seat purchase, so those are places where we could see a big benefit to a caching layer.

What about an **index**? A table index speeds up the process of searching the table for a particular set of matching fields. Any time we're making the same sort of query frequently, an index can help to make that query run quickly.

> Why not just add an index on every field?
>
> Each index takes up space. And, every time we insert a new record, we need to update any indices that refer to it. So there's a space/time tradeoff here—more indices *may* mean faster lookups, but they *also* mean more space overhead and slower updates.

Looking at the steps for buying a ticket, here are a few indices we could consider:

- An index on the `Seats` table on the `seatEventId`, `seatStatus`, and `seatSelectionTime` fields. (So we can quickly find seats for a particular event that are available.)

- An index on the `Users` table on the `userId` field. (So we can quickly lookup user payment information.)
- An index on the `Events` table by `eventDate`. Since we'll often be looking for events happening in a specific time period, our index should support efficient range queries.

Obviously, once we've gotten things set up, we can profile our database queries to make sure they're all happening quickly. If needed, we can always add or tweak indices.

For tables that're frequently read, we can add a caching layer, like Memcached, to make retrieving popular records fast. Which tables are frequently read? The `Events`, `Seats`, and `Users` tables are queried on every seat purchase, so those are places where we could see a big benefit to a caching layer.

Interviewer: **Okay, good. We've spent a lot of time on the database side of things. Any other bottlenecks you see?**

Scaling Up: Front End Servers

We're expecting 1,600 requests per second at peak loads. Can one server handle all that?

It's hard to know, since the number of requests a server can handle depends on how powerful it is, how much time each request takes to process, how much network traffic is involved, etc. But it's a fair bet we'll probably need more than one server, even if we can't say *exactly* how many servers we'll need.

Having multiple servers is *super* common in system design. Since we might not know how many servers we want, we'll want to be able to spin up extra servers on the fly when we're getting lots of requests. An **autoscaler** like Amazon EC2 can take care of automatically adding more servers for us. We can also use a **load balancer** to distribute incoming requests among all of our available servers.

> Load balancers and autoscalers are pretty standard approaches to scaling up servers. They're definitely worth keeping in your back pocket when working through system design interviews.

Let's go ahead and draw out what our system looks like now:

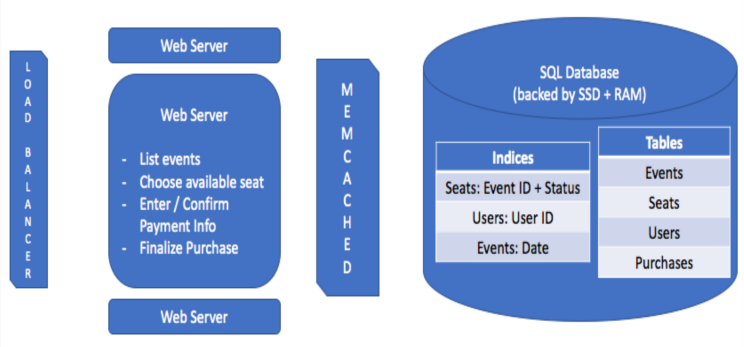


IMAGE EDIT: Add in an autoscaler -- a diamond box labeled "Autoscaler" with lines connecting it to the load balancer and web server.

Interviewer: **Getting back to that front-end logic. Any ideas for speeding that up?**

Front End Optimizations

One way we could save is by using **static HTML** for common pages, instead of dynamically generating every page.

For example, when a user comes to our site, we'll show them a list of events they can buy tickets to. Realistically, we don't need to generate that page every time someone comes to our site, since it'll look the same for everybody.

So, every minute, let's regenerate our home page and store the result as static HTML. Then, when someone visits the homepage, we can make the page load super fast, just sending back what we've already made.

Interviewer: **Finally, how can we make our site reliable?**

Reliability: Remove Single Points of Failure

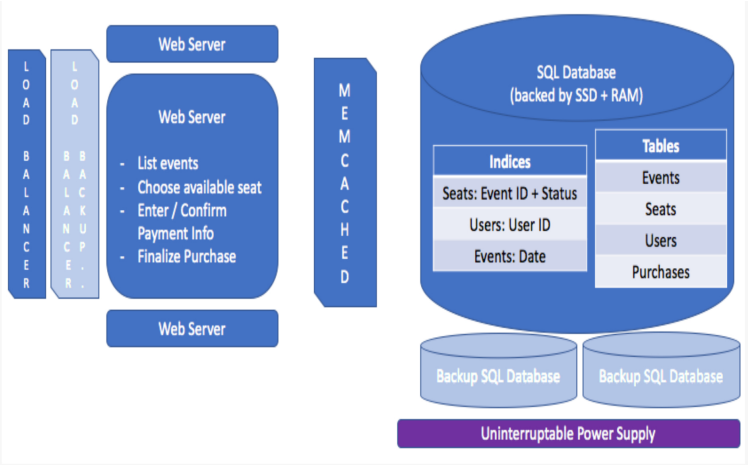
As designed, our system has a number of **single points of failure**. These are pieces of our system that could bring everything else down if they break.

As a good rule of thumb, the answer to single points of failure is adding **backups / failovers**. Instead of having one copy of our database, keep two (or even three), so that we can use the extra copy if the first one breaks. Instead of just one load balancer, use two.

What about the power going out? The main reason this could be an issue is that we talked about storing some of our data in RAM. Since RAM is volatile storage, it doesn't stick around if the power is turned off. That means if we lose power, our data in memory could be lost. Whoops.

One way to fix this is to add in some backup power. An **uninterruptible power supply (UPS)** is a large battery that can keep a machine running for a few minutes—just long enough to copy our data from RAM onto persistent storage, like those SSDs.

Putting it all together, here's what our system looks like:



Bonus

Having fun? Here are some additional pieces of the system to think about:

1. What if we wanted to allow users to cancel their tickets and get refunds?
2. Our traffic comes in bursts. We've provisioned for peak loads ... but we can probably get by with less during off-peak times. Which parts of our system could we scale *down* dynamically?
3. How would we design firewalls/routing rules within our system? Which machines should be able to talk to each other, and over what protocols?

What We Learned