Core Concepts

API Design

Learn about API design for system design interviews

In system design interviews, you'll want to define how clients interact with your system as part of the **API step** in the **Delivery framework**.

API design follows predictable patterns. You'll pick a protocol, define your resources, and specify how clients pass data and get responses back. This article won't make you a master in API design, but it should cover all the basics needed to impress in this 5-minute period of your system design interview.

> Before we go deep here, I want to make one thing super clear: most interviewers don't care deeply about your API design being perfect. They want to see that you can design a reasonable API and move on to the more complex parts of your system.
>
> That said, if you're interviewing for frontend or product roles, API design matters more since you'll be working closely with APIs daily. Also, for junior roles, there's less expectation on your ability to design distributed systems, so there may be more time spent in the interview on APIs.

API Types

In an interview, you'll typically choose between three main API protocols:

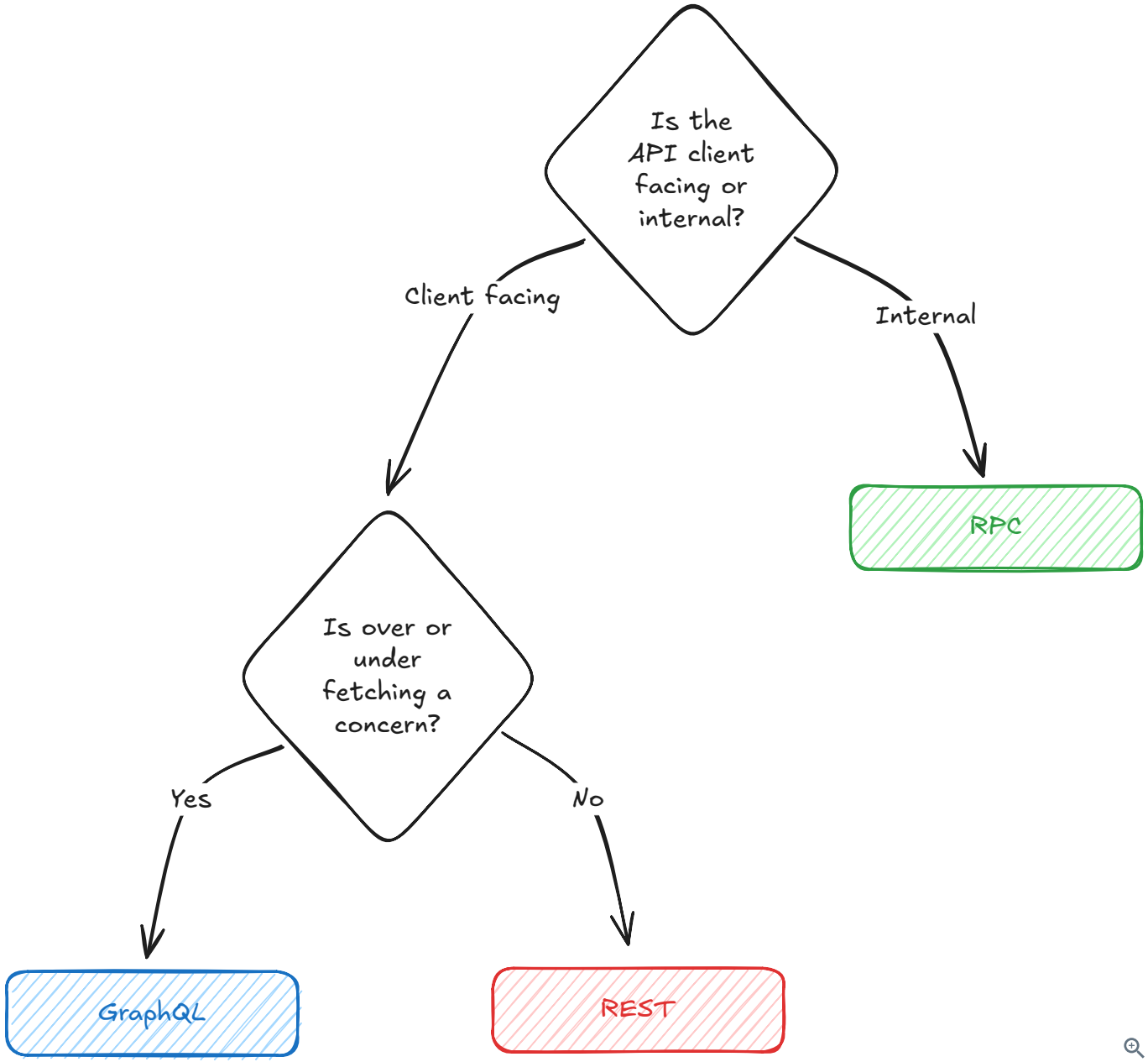
1. **REST (Representational State Transfer)** - REST uses standard HTTP methods (GET, POST, PUT, DELETE) to manipulate resources identified by URLs. For standard CRUD operations in web and mobile applications, REST maps naturally to your database operations and HTTP semantics, making it the go-to protocol for most web services. This should be your default choice.

2. **GraphQL** - Unlike REST's fixed endpoints, GraphQL uses a single endpoint with a query language that lets clients specify exactly what data they need. Think about a mobile app that needs only basic user information versus a web dashboard that displays comprehensive analytics - with REST, you'd either create multiple endpoints or force clients to fetch more data than they need, but GraphQL lets each client request exactly what it needs in a single query. If your interviewer mentions "flexible data fetching" or talks about avoiding over-fetching and under-fetching, they're signaling you to consider GraphQL.

3. **RPC (Remote Procedure Call)** - RPC protocols like gRPC use binary serialization and HTTP/2 for efficient communication between services. While REST treats everything as resources, RPC lets you think in terms of actions and procedures - when your user service needs to quickly validate permissions with your auth service, an RPC call like `checkPermission(userId, resource)` is more natural than trying to model this as a REST resource. If the interviewer specifically mentions microservices or internal APIs, consider RPC for those high-performance connections. Use RPC when performance is critical (see **Networking Essentials** for deeper protocol details).

> Default to REST unless you have a specific reason not to. It's well-understood, has great tooling, and works for 90% of use cases. If you're unsure, just say "I'll use REST APIs" and move on.

> For real-time features like notifications, chat, or live updates, you'll need different protocols like WebSockets or Server-Sent Events. These aren't traditional APIs - they're persistent connections. But they're important to know and you can learn all about them in our Real-time Updates Pattern.



API Types Flowchart

Let's go through each of these one-by-one and call out what matters in your interview.

REST

Since REST is your default choice, let's spend most of our time here understanding how to design REST APIs that work well in system design interviews.

Resource Modeling

The foundation of good REST API design is identifying your resources correctly. If you've followed the **delivery framework**, resources are just your **core entities**.

Take Ticketmaster as an example. Your core entities might be events, venues, tickets, and bookings. These naturally map to REST resources:

```
GET /events                    # Get all events
GET /events/{id}               # Get a specific event
GET /venues/{id}               # Get a specific venue
GET /events/{id}/tickets       # Get available tickets for an event
POST /events/{id}/bookings     # Create a new booking for an event
GET /bookings/{id}             # Get a specific booking
```

Importantly, REST resources should represent *things* in your system, not *actions*. Instead of thinking about what users can do (like "book" or "purchase"), think about what exists in your system (events, venues, tickets, bookings).

> Resources should always be plural nouns, i.e., bookings, events, tickets, etc. Most interviewers don't care about this, but some do, and it's easy enough to get right, so you might as well.

When handling relationships between resources, you have two main approaches. You can nest resources when there's a clear parent-child relationship, like `/events/{id}/tickets` for all tickets belonging to a specific event. Alternatively, you can keep resources flat and use query parameters for filtering, like `/tickets?event_id=123` .

The key difference is whether the relationship is required or optional. Use path parameters (or nested resources) when the value is required - like `/events/{id}/tickets` where you always need to specify which event's tickets you want. Use query parameters when the filter is optional - like `/tickets?event_id=123&section=VIP` where you might want all tickets, or tickets filtered by event, or tickets filtered by both event and section.

> In interviews, this distinction helps you make the right choice quickly. If the relationship is always required for the query to make sense, use a path parameter. If it's an optional filter among many possible filters, use a query parameter. The interviewer cares more about whether you can identify the right resources than perfect URL structure, but showing this understanding demonstrates good API design intuition.

HTTP Methods

Once you've identified your resources, you need to decide how clients interact with them. HTTP provides a set of methods (verbs) that map naturally to common operations, and understanding when to use each one is crucial for interviews.

GET is for retrieving data without changing anything. Use GET for `/events/{id}` to fetch event details or `/events` to list all events.

POST creates new resources. When a user books tickets, you'd POST to `/events/{id}/bookings` with the booking details in the request body. The server assigns an ID and returns the newly created booking. POST is neither safe nor idempotent. In other words, calling it multiple times creates multiple bookings.

PUT replaces an entire resource with what you send. If you're updating a user's profile completely, PUT to `/users/{id}` with the full user object. Unlike POST, PUT is idempotent, so sending the same data multiple times results in the same final state.

PATCH updates part of a resource. When a user changes just their email address, PATCH to `/users/{id}` with only the email field.

DELETE removes a resource. DELETE `/bookings/{id}` cancels a booking. It's idempotent - deleting an already-deleted resource should return the same result.

The key here is idempotency. Operations that can be safely repeated without changing the outcome (GET, PUT, PATCH, DELETE) are idempotent. This matters when networks fail and clients retry requests, you don't want duplicate bookings from a retry.

Passing Data To APIs

API endpoints need input to tell the server what to do. This can be which resources to fetch, what data to update in the database, or how to filter results. Understanding where to put different types of input is crucial for designing clean, intuitive APIs.

You have three main options for passing data to your REST API, and each serves a different purpose.

Path parameters identify which specific resource you're working with. When you want to get event details, you put the event ID in the path: `/events/123`. The ID is part of the URL structure itself, making it clear that you're asking for a specific event, not a collection of events. Use path parameters when the value is required to identify the resource - without it, the request doesn't make sense.

Query parameters filter, sort, or modify how you retrieve resources. When you want to search for events in a specific city or date range, you use query parameters: `/events?city=NYC&date=2024-01-01`. These are optional - you could ask for all events without any filters, or apply multiple filters together. Query parameters work well for pagination too: `/events?page=2&limit=20`. Note that the first option is separated via a `?` and all subsequent parameters are separated by `&`.

Request body contains the actual data you're sending to create or update resources. When a user books tickets, you POST to `/events/{id}/bookings` with the booking details in the request body. Things like how many tickets, seating preferences, and so on. The request body is where you put complex data structures and anything that might be too large or sensitive for a URL.

Each type of input serves a different role in the API's contract. Path parameters are structural, they determine which endpoint you're hitting. Query parameters are modifiers, they change how the endpoint behaves. Request body is payload, it's the data you're actually working with.

Here's a practical example. If you're building a booking system and a user wants to book VIP tickets for a specific event, you'd structure it like this:

```
POST /events/123/bookings?notify=true
{
  "tickets": [
    {"section": "VIP", "quantity": 2},
    {"section": "General", "quantity": 1}
  ],
  "payment_method": "credit_card"
}
```

The event ID (123) is in the path because you need to specify which event you're booking. The notification preference is a query parameter because it's optional behavior. The actual booking details go in the request body because they're the core data you're creating.

Returning Data

An API response is made up of two parts:

1. The status code, which indicates whether the request was successful or not.
2. The response body, which contains the data you're returning to the client (typically JSON)

For status codes, stick to the common ones: 200 for success, 201 for created resources, 400 for bad requests, 401 for authentication required, 404 for not found, and 500 for server errors.

> Don't overthink this in interviews - interviewers care more about whether you understand the difference between client errors (4xx) and server errors (5xx) than memorizing every status code. Even using X's to obscure the status code is typically totally fine.

GraphQL

GraphQL emerged from Facebook in 2012 to solve a specific problem: their mobile app needed different data than their web app, but they were stuck with REST endpoints that returned fixed data structures. The mobile team kept asking for new endpoints or modifications to existing ones and this was slowing down development on both sides.

With REST, you typically have two unpleasant choices when different clients need different data. You can create multiple endpoints for different use cases, leading to endpoint proliferation and maintenance headaches. Or you can make your endpoints return everything any client might need, leading to over-fetching where mobile clients download megabytes of data they don't use.

GraphQL consolidates resource endpoints into a single endpoint that accepts queries describing exactly what data you want. The client specifies the shape of the response, and the server returns data in that exact format.

How GraphQL Works

Here's a simple example using our Ticketmaster scenario. Instead of separate REST endpoints for events, venues, and tickets, you'd have a single GraphQL endpoint that can handle queries like this:

```
query {
  event(id: "123") {
    name
    date
    venue {
      name
      address
    }
    tickets {
      section
      price
      available
    }
  }
}
```

The server returns exactly what you asked for, nothing more, nothing less. If the mobile app only needs event names and dates, it can request just those fields. If the web dashboard needs comprehensive event details with venue information and ticket availability, it can request all of that in a single query.

When To Use GraphQL In Interviews

GraphQL is the right choice when you have diverse clients with different data needs. If your interviewer mentions scenarios like "the mobile app needs different data than the web app" or asks about "avoiding over-fetching and under-fetching," they're likely looking for you to bring up GraphQL.

It's also a good choice when frontend teams need to iterate quickly without backend changes. With REST, adding a new field to a mobile screen often requires backend changes and API deployments. With GraphQL, the frontend team can request additional fields as long as they exist in the schema.

However, GraphQL adds complexity. You need to implement query parsing, schema validation, and often sophisticated caching strategies. For most system design interviews, REST is simpler and more straightforward unless the problem specifically calls for GraphQL's flexibility.

GraphQL Schema Design

If you do choose GraphQL, you'll need to think differently about design. Instead of REST's resource endpoints, you design a schema that defines your data types and their relationships.

For our Ticketmaster example, you'd start by modeling your core entities as GraphQL types:

```
type Event {
  id: ID!
  name: String!
  date: DateTime!
```

```
  venue: Venue!
  tickets: [Ticket!]!
}

type Venue {
  id: ID!
  name: String!
  address: String!
}

type Query {
  event(id: ID!): Event
  events(limit: Int, after: String): [Event!]!
}
```

The key difference from REST is that you define relationships directly in the schema. An Event has a Venue, and clients can traverse that relationship in a single query.

But this flexibility creates the **N+1 problem**, the biggest GraphQL gotcha. When a client queries events with their venues, you might execute one query for events, then N separate queries for each venue. With 100 events, that's 101 database queries instead of 2. The solution is batching/dataloader patterns that group related queries together, but it adds complexity you don't have with REST.

GraphQL also handles authorization differently. Instead of securing entire endpoints like REST, you secure individual fields. A user might see an event's name and date but not the revenue data. You can control this at the field level in your schema resolvers.

> In interviews, mention GraphQL when you see clear over-fetching or under-fetching problems, but don't default to it. Most interviewers appreciate that you know about GraphQL, but they usually prefer to see you solve the core architectural challenges with simpler tools first.

RPC

RPC (Remote Procedure Call) is a protocol that allows a client to call a procedure on a server and wait for a response without the client having to understand the underlying network details. It's faster than HTTP for service communication, especially when you need high performance and low latency.

How RPC Works

Unlike REST's resource-oriented approach, RPC is action-oriented. You're essentially calling functions across a network as if they were local functions in your codebase. Here's how the same Ticketmaster operations might look with RPC:

```
// Instead of GET /events/123
getEvent(eventId: "123")

// Instead of POST /events/123/bookings
createBooking(eventId: "123", userId: "456", tickets: [...])

// Instead of GET /events/123/tickets
getAvailableTickets(eventId: "123", section: "VIP")
```

The most popular RPC protocol today is **gRPC**, which uses Protocol Buffers for serialization and HTTP/2 for transport. This combo is much faster than REST's JSON-over-HTTP approach, especially for service-to-service communication. Another notable RPC framework is **Apache Thrift**, originally developed at Facebook and now open source, which supports multiple programming languages and serialization formats.

Protocol Buffers And Type Safety

gRPC uses Protocol Buffers (protobuf) to define service contracts. You write a `.proto` file that describes your service methods and data structures:

```
service TicketService {
  rpc GetEvent(GetEventRequest) returns (Event);
  rpc CreateBooking(CreateBookingRequest) returns (Booking);
  rpc GetAvailableTickets(GetTicketsRequest) returns (TicketList);
}

message GetEventRequest {
  string event_id = 1;
}

message Event {
  string id = 1;
  string name = 2;
  int64 date = 3;
  Venue venue = 4;
}
```

From this single definition, gRPC generates client and server code in multiple programming languages. This means your Go backend service and your Java payment service can communicate with compile-time type safety, catching mismatches before deployment.

When To Use RPC In Interviews

RPC shines in microservice architectures where services need to communicate frequently and efficiently. If your interviewer mentions internal service communication, high-performance requirements, or polyglot environments (different services in different languages), RPC is likely a good choice.

Consider RPC when:

- **Performance is critical**: Binary serialization and HTTP/2 make RPC significantly faster than JSON REST
- **Type safety matters**: Generated client code prevents many runtime errors
- **Service-to-service communication**: Internal APIs between your own services don't need REST's resource semantics
- **Streaming is needed**: gRPC supports bidirectional streaming for real-time features

For our Ticketmaster example, you might use REST APIs for your public endpoints that mobile apps and web clients consume, but use gRPC for internal communication between your booking service, payment service, and inventory service.

> Unless explicitly asked, you won't typically outline your internal APIs during the API step of the interview. Instead, focus on just the user facing APIs here. At most, you'll call out that internal services communicate over RPC during your high-level design.

Common API Patterns

Regardless of whether you choose REST, GraphQL, or RPC, there are some patterns that apply across all API types. These are worth knowing since they come up in most real-world systems.

Pagination

When you're dealing with large datasets, you can't return everything at once. Imagine an API that returns all events ever created, that could be millions of records which would be many gigabytes of data.

Instead, you need pagination to break large result sets into manageable chunks. There are two main approaches to pagination: offset-based and cursor-based.

Offset-Based Pagination

Offset-based pagination is the simplest approach and used by most websites. You specify how many records to skip and how many to return: `/events?offset=20&limit=10` gets records 21-30. This is intuitive and easy to implement, but it has problems with large datasets. If someone adds a new event while you're paginating through results, you might see duplicates or miss records as the data shifts.

Cursor-Based Pagination

Cursor-based pagination solves this by using a pointer to a specific record instead of counting from the beginning. Here's how it works in practice:

First request: `/events?limit=10`
Response includes the events plus a cursor pointing to the last record:

```
{
  "events": [...],
  "next_cursor": "cmd9atj3p000007ky19w1dpy2"
}
```

Next request: `/events?cursor=cmd9atj3p000007ky19w1dpy2&limit=10`

The cursor is typically an encoded reference to a specific record (like an ID or timestamp). This is more stable because it's not affected by new records being added, but it's harder to implement features like "jump to page 5." In the example, `cmd9atj3p000007ky19w1dpy2` is the id of the last event in the first page.

For interviews, offset-based pagination is usually fine unless you're dealing with real-time data or the interviewer specifically asks about high-volume scenarios. Most interviewers care more about whether you remembered to include pagination than which specific approach you choose.

Versioning Strategies

APIs evolve over time, and you need a strategy for handling changes without breaking existing clients. This is particularly important for public APIs where you can't control when clients update their code.

The most common approach is **URL versioning**, where you include the version number in the path: `/v1/events` or `/v2/events`. This is explicit and easy to understand. Clients know exactly which version they're using just by looking at the URL. It's also simple to implement since you can route different versions to different code paths.

Header versioning puts the version in an HTTP header instead: `Accept-Version: v2` or `API-Version: 2`. This keeps URLs cleaner and follows HTTP standards better, but it's less obvious to developers and harder to test in browsers.

For interviews, URL versioning is usually the safer choice because it's more widely understood and easier to explain quickly. Unless the interviewer specifically asks about header versioning, stick with the URL approach.

> You'll see that in our breakdowns we don't even include versioning in the API design. This is more a product of it just not being important to most interviewers rather than a statement that it's not important in practice (it is).

Security Considerations

Security is often treated as an afterthought in interviews, but demonstrating security awareness can set you apart. You don't need to design a bulletproof security system, but showing that you understand basic API security principles signals that you think about production-ready systems.

Authentication and Authorization

The first question your API needs to answer is: "Who is making this request, and are they allowed to do what they're asking?"

Authentication verifies identity - proving the user is who they claim to be. **Authorization** verifies permissions - checking if that authenticated user is allowed to perform the specific action they're requesting.

For our Ticketmaster example, authentication might verify that the request comes from user "john@example.com", while authorization checks if John is allowed to cancel the specific booking he's trying to cancel (he should only be able to cancel his own bookings, not everyone's).

API Keys Vs JWT Tokens

When designing authentication for your API, you'll typically choose between two main approaches depending on who will be using your API and how they'll access it.

> For most (but not all), interviews, authentication and authorization are not a focus. My advice would be to call out which endpoints require the user be authenticated and say that you'd rely on a JWT or store the user's session in a database to authenticate the user.

API Keys

API keys are long, randomly generated strings that act like passwords for applications rather than humans. When a client makes a request, they include their API key in the Authorization header, and your server looks up that key to identify which application is making the request.

Here's how they work: you generate a unique API key for each client (like `sk_live_abc123def456...`), store it in your database along with any permissions or rate limits for that client, and then verify each incoming request by looking up the key. They're perfect for server-to-server communication where you control both sides. When your booking service needs to call your payment service, an API key is straightforward and effective. They also make sense when you're exposing your endpoints to 3rd party developers who need programmatic access to your system.

> If you're building a user-facing product with user-facing APIs, API keys are almost never the right choice. Users shouldn't be managing long cryptographic strings, and API keys don't expire or carry user context the way user sessions need to.

```
GET /events
Authorization: Bearer sk_live_abc123...
```

JWT (JSON Web Tokens)

JWT tokens, on the other hand, encode user information directly into the token itself rather than storing session state on your server. When a user logs in successfully, your server creates a JWT containing their user ID, permissions, and an expiration time, then signs the entire token with a secret key.

Conveniently, when that JWT comes back with future requests, you can verify it's authentic by checking the signature, and you can read the user information directly from the token without any database lookups. The token itself carries all the context you need to authorize the request.

JWTs work particularly well for distributed systems because any service that knows your signing secret can validate tokens independently. If your mobile app sends a JWT to your API gateway, the gateway can verify the user's identity and forward the request to your booking service with confidence.

```
// JWT payload
{
```

```
  "user_id": "123",
  "email": "john@example.com",
  "role": "customer",
  "exp": 1640995200
}
```

Use API keys for internal service communication and external developer access. Use JWT tokens for user sessions in web and mobile applications. JWT tokens can be stateless (no database lookup required) and can carry user context, making them ideal for user-facing applications.

Role-Based Access Control (RBAC)

Real systems have different types of users with different permissions. In our Ticketmaster system, customers can book tickets and view their bookings, venue managers can create events and view sales reports, and system admins can access everything.

RBAC assigns roles to users and permissions to roles:

```
Roles:
- customer: can book tickets, view own bookings
- venue_manager: can create events, view sales for their venues
- admin: can access everything

User: john@example.com → Role: customer
User: manager@venue.com → Role: venue_manager
```

In your API design, you'd check both authentication and authorization:

```
GET /bookings/{id}
1. Is the user authenticated? (valid JWT token)
2. Is the user authorized? (owns this booking OR is admin)
```

> At most, in an interview you'd just mention which endpoints can be accessed by which roles, though more times than not this distinction is not relevant.

Rate Limiting and Throttling

Rate limiting prevents abuse by restricting how many requests a client can make in a given time period. This protects your system from both malicious attacks and accidental overuse.

Common strategies include:

- **Per-user limits**: 1000 requests per hour per authenticated user
- **Per-IP limits**: 100 requests per hour for unauthenticated requests
- **Endpoint-specific limits**: 10 booking attempts per minute to prevent ticket scalping

You typically implement rate limiting at the API gateway level or using middleware in your application. When limits are exceeded, return a `429 Too Many Requests` status code.

> In interviews, mentioning rate limiting shows you understand production concerns, but don't spend time designing the specific algorithms unless asked. A simple "we'll implement rate limiting to prevent abuse" is usually sufficient.

Conclusion

API design in system design interviews is about demonstrating solid engineering judgment, not creating perfect specifications. Focus on choosing the right protocol for your use case (usually REST), modeling your resources clearly, and showing you understand the basics of authentication and security.

It's all about balance in an interview. Spend enough time to show you can design a reasonable API, but don't get bogged down in details when there are bigger architectural challenges to tackle. Your interviewer wants to see that you can build systems that work, not that you've memorized every HTTP status code. In practice, candidates mess up more often by spending too much time on API design than by underinvesting. Do your best to not spend more than 5 minutes outlining your APIs in the interview.