What kind of APIs?

* HTTP based APIs
* REST APIs, JSON APIs, GraphQL, etc.

Two principles for building APIs:

* Designing the minimal API surface
* Designing from Strict to Loose

Design the minimal API surface:

* Most devs try to be speculative about what they might need in the future, but this works ends up not being used and ends up having to be maintained
  + Ex: If we want to expose an email field but that field gets removed. Then we will have to update all code that expects that field
* It will also break clients for those that can choose fields (graphQL or sparse json)
  + Clients are others who consume the API
* A lot of redundant cycles to mark the field and remove it

Bloated API surface:

* Redundant fields
* Redundant relationships
* Redundant input fields

Don’t expose redundant relationships:

* Ex: Showing that a post was reviewed. We only need to expose the boolean field not the actual reviewer
* This makes development costs more expensive:
  + Batch/eager loading
  + Testing
  + Performance impacts
* We should delay decisions if possible so that we have better knowledge
* It is easier to add in the future than to remove

Don’t expose redundant input fields:

* Payload used for update/mutate data for POST/PUT requests
* Devs might want to have parity where if a certain endpoint accepts a field, then we should add that field to other methods
* Ex: creating an endpoint for blog comments that accepts the blog post id and adding that field to the update for the comment
  + Why would we expose it? The associated blog for the comment shouldn’t change. If it doesn’t, then we have to handle want happens if clients change the blog post id

Avoid anemic data modeling:

* https://xuorig.medium.com/graphql-mutation-design-anemic-mutations-dd107ba70496

Ambiguity deteriorates developer experience:

* Clients get more confused and this requires more clarification

2nd Principle: Strict to loose:

* Avoid extra flexibility
* Breaks first

Avoid extra flexibility:

* A comments endpoint that requires a post id, but a dev might suggest that someone might want to get ALL comments so let’s make post id optional
  + If there is a post id, get all comments associated with that post. If not, then return all comments
  + This requires unneeded application logic
  + More code requires more maintenance
  + More code requires more tests
* On inputs, it’s easy to go from required to optional, not the other way around
* Strict to loose gives you more control
  + Strict = required
  + This gives more flexibility to you (not clients) for changing
* Loose to strict is a breaking change

Break first:

* Ex: showing comments for a post. Let’s show all comments instead of pagination
  + Later down the road, there’s a scenario where there are 100s of comments and this requires work to refactor to make comments paginated
  + This will be a breaking change for clients
* Add pagination limits is a best practice
  + It could exhaust the server
* Super hard to add retroactively, but super easy to adjust in the future

Adopt Schema First Design

* Figure out what is your domain and who the API is for
* Don’t couple API to specific implementation
* Talk to your clients; find what their needs are
* Be responsive to your clients requests
* There’s always exceptions
  + Platform API (don’t know clients or competitors)
  + Constraints due to release cycles

Conclusion

* Redundant work slows down progress on important features