Make the switch from REST to GraphQL

Learn best practices for schema design when migrating from REST to GraphQL

APOLLO

CONTENTS

APOLLO

THE JOURNEY FROM REST TO GRAPHQL

Developers have rapidly adopted GraphQL since it open sourced in 2015. GraphQL focuses on creating APIs that are designed around how data will be consumed—improving the way developers build applications and enabling product teams to deliver new experiences faster. This differs from previous API technologies that focused on creating endpoints that were very data centric and not concerned with how data was going to be consumed.

How is GraphQL different from REST?

For developers, GraphQL reduces the amount of time they spend writing boilerplate, data-fetching code so they can focus on building features. Application performance also improves thanks to a reduced payload size and fewer roundtrips.

While many engineering teams rely on a REST architecture to deliver digital experiences for their users, REST does not easily scale to integrate new capabilities that organizations need to stay current and competitive. With GraphQL, client engineering teams have a more flexible way to fetch data and integrate additional capabilities into new and existing platforms as they pop up. This allows them to deliver more value to customers and drive business growth.

GraphQL is inherently declarative, which means you can query the exact fields you're interested in, instead of over-fetching, under-fetching, and making more than one API call.

GraphQL exchanges data at a single endpoint, allowing developers to have a single source of truth for all their business capabilities.

GraphQL offers a better developer experience with strong typing and a more agile approach to versioning. Teams use GraphQL metadata to provide deprecation hints when they add, remove, and change fields on an existing graph.

GraphQL has an active community of developers that are developing open source and commercial solutions that power the majority of experiences at companies like Expedia, Walmart, PayPal. GraphQL is also becoming the preferred data layer for fast-growing startups.

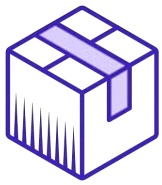
Organizations that have a vast set of REST APIs can still benefit from GraphQL by deploying it as a layer in between applications and existing APIs. The remainder of this ebook outlines best practices to keep in mind when designing your GraphQL schema, ensuring you don't simply replicate your REST API patterns and instead harness the full benefits of GraphQL.

APOLLO

4 TIPS TO MAKE SURE YOUR GRAPH DOESN'T FEEL REST-Y

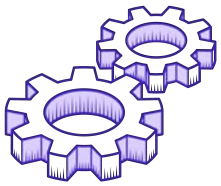
The real benefit of GraphQL comes from rethinking API design to be demand driven based on client needs. For teams moving to GraphQL for the first time, there is often a temptation to either auto-generate GraphQL schemas or mimic REST API patterns. But simply replicating your REST API patterns results in API designs that aren't optimized for client needs.

Here are four tips to make sure your GraphQL isn't just a one-to-one mapping of your existing capabilities.



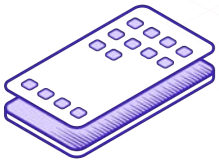
01 Think about the business capabilities you want in your graph

While service teams typically build APIs, GraphQL schemas work best when they're shaped for their primary customer—the client. Before designing your schema, work with your product team to understand how clients will consume the data and what capabilities need to be available to them.



02 Don't bulk convert all your APIs to GraphQL

Often, when an API team is mandated to implement GraphQL, they expose capabilities en masse without considering the relationships between data. This can lead to a graph that essentially operates as REST APIs. If you turned your whole JSON object into a GraphQL object type and put it in your graph, you'd end up with additional fields that wouldn't be consumed on the client. It would also be extra noise in your graph that would obfuscate the capabilities you want to drive. Instead, talk to your client developers about what they'd like to see in the API response and think about how you make it easier for them to consume the response.



03 Deliver performant client experiences with a microservices architecture

With the shift to microservices, many organizations moved away from large, single endpoint API responses to more numerous, finer-grained responses tailored around data domains. This came at a cost though: client development became harder, as clients now had to orchestrate numerous data calls. GraphQL lets you extract this orchestration problem from clients and provide them with a single, large response while still using a microservices-based architecture for the backend.

04 Avoid CRUD

When you start writing queries and seeing how the client consumes them, there's a tendency to replicate just how users would interact with your REST API. However, you shouldn't be able to read create, read, update, and delete in your graph. We'll explain in detail on page 8.

DEVELOP GRAPHQL SCHEMA ON TOP OF EXISTING REST APIS

One of the benefits of GraphQL is that you don't have to rip and replace all of the REST APIs that you've built over the years. Instead, you can incrementally adopt GraphQL on top of existing REST APIs to harness these capabilities in a way that is easier for your frontend developers to build new experiences.

If you're coming from REST, designing your GraphQL schema correctly requires you to **understand the differences between REST API methods and GraphQL API operations**. With REST APIs, the focus is on GET and POST methods, but with GraphQL, you can think in terms of two operations: query and mutation. Because of the nature of GraphQL schema, you can group functionality in ways that make sense for your business and allow client engineers to write operations bespoke to the requirements of the features they're working on.

Instead of creating mappings on top of existing database collections or REST API data response fields, think about how you can break down a current API response into more usable parts. Identify what the frontend users want in terms of business capabilities and build the schema based on that. This way, you can design a schema that reflects how frontend developers are querying it and you don't constrain yourself to the exact shape of data returned by your existing APIs. When **frontend and backend teams agree on a schema first**, it serves as a contract between the UI and the backend before any API engineering happens.

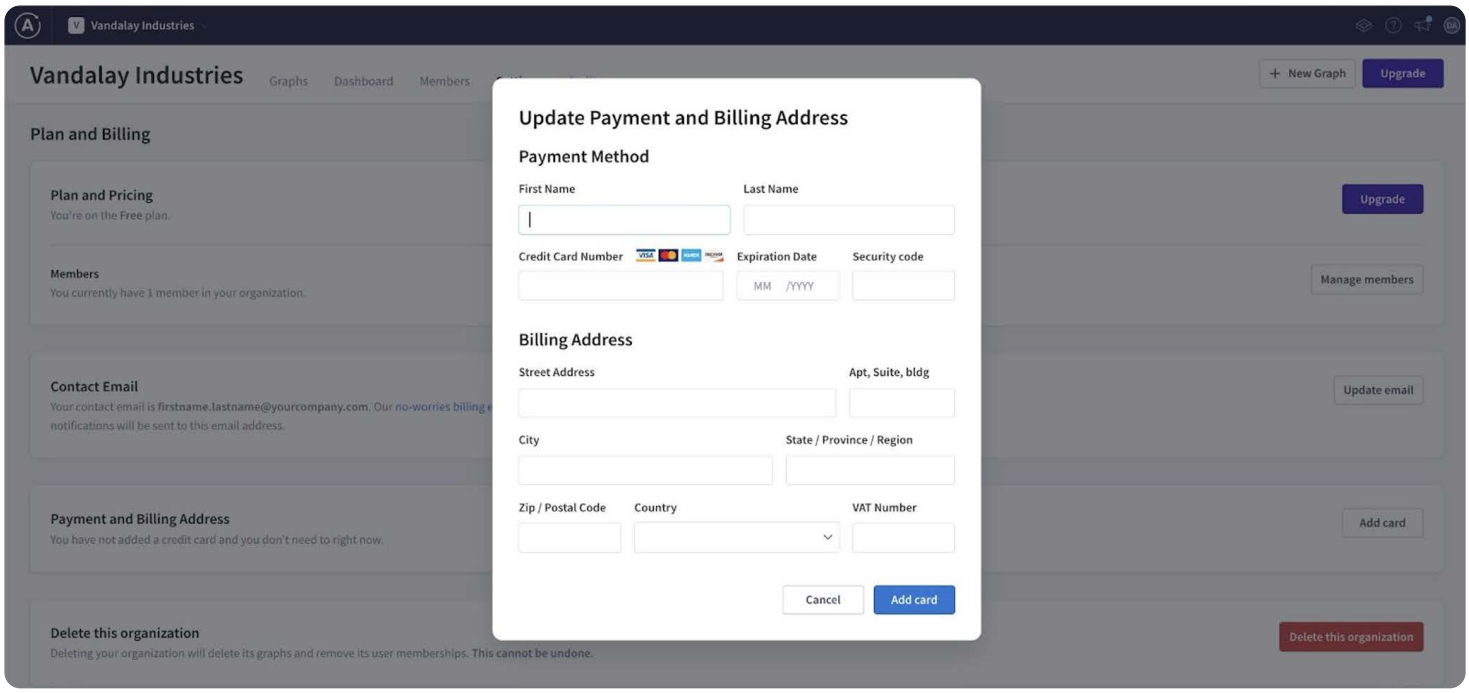
Before writing any code, ask yourself: **how do you want to expose your data?** A lot of teams want to know how to break apart their API that returns everything. Think about what will actually be displayed on the UI and bring those objects over first. Because GraphQL is flexible, you can easily add or remove fields when you need to.

Next, think about mutations. Queries are used to fetch data, while mutations modify server-side data. One benefit of mutations is that due to the declarative nature of GraphQL, developers can easily see the structure of the fields they can modify. This more granular approach also helps observability because you gain full visibility into how all the fields of your API are being used. This way, you can evolve your API based on actual consumption, deprecate unused fields, and provide easy navigation.

BEWARE OF CRUD WHEN DESIGNING YOUR GRAPH

When migrating to GraphQL, you have an opportunity to start breaking your API apart, grouping fields together, and putting them into a different format. But as a general rule, you should avoid CRUD operations in your graph.

We'll use an example that is common to many applications: account management. In most cases, the application has an account settings page that allows you to take actions such as updating billing information, changing the email address, and setting a default shipping address.



This account settings page lets you update payment and billing information.

Let's walk through an example of how we might do this if we were using a REST-y approach. Then we'll show a better approach that takes advantage of GraphQL's declarative nature.

```
type Query {
  account(id:ID!): Account
}
type Mutation {
  createaccount(input: AccountInput!): Account
  updateAccount(account: AccountInput!): Account
  deleteAccount(id: ID!): Void
}
```

This REST-style GraphQL mutation looks a lot like CRUD.

We have a query for an account and a mutation to create, update, or delete an account. But this looks a lot like CRUD. It isn't any different than REST and updateAccount is definitely not a consumer-friendly schema because it's unclear what's associated with an account. Consumers don't know if they need to provide all the inputs, or, if they only provide some of them, will they update? And you can't tell if any are protected. This forces the server developer to require multiple fields in the input type and a consumer must have a higher level of knowledge to work with an account.

On the other hand, a GraphQL schema makes those capabilities easy to consume. In the example left, we've modified our query to provide more clarity as to what updateAccount actually does. We've grouped the functionality of accounts under a mutation type.

```
type Query {
  account: Account #ID is inferred from bearer token
}
type Mutation {
  account: Account #ID is inferred from bearer token
}

type AccountMutation {
  updateEmail(email: String!): Account
  addAddress(address: AccountInput!): Account
  addAddressAsDefault(address: AccountInput!): Account
  setDefaultShippingAddress(addressID: ID!): Account
}
```

The second mutation in this schema is consumer friendly because it explains the functionality of update Account.

With GraphQL developers get first-class tooling out of the box. This makes it easier for new team members to add value faster, eliminating guesswork and allowing them to leverage robust autocompletion when writing operations—just begin to write an operation and GraphQL will show you exactly what you need to satisfy the relevant type.

EXPOSE YOUR REST CAPABILITIES WITH APOLLO TOOLS

Now that you know there's no need to throw away existing API infrastructure, with GraphQL and developer tools from Apollo GraphQL you can build on your existing REST APIs and services.

Apollo Server can function as your standalone GraphQL server, an add-on to your application's existing Node.js middleware (such as Express or Fastify), or a gateway for a federated GraphQL. In any case, it acts as a thin layer between your application and your existing REST APIs.

Apollo Workbench is a VS Code extension that helps you design GraphQL schema without writing any code. Apollo Workbench was created to help organizations better understand the composition of their GraphQL schemas during the design phase, as opposed to implementation. This way, engineers don't have to write code when planning a schema design, they can just change the schema directly.

Apollo Federation is the industry-standard open architecture for building a distributed graph that scales across teams. Use Apollo's graph router to compose a unified graph from multiple subgraphs, determine a query plan, and route requests across your services.

EXPAND TO APOLLO FEDERATION

Once your initial GraphQL server is up and running, you might have another—and another. And maybe even another after that. GraphQL is contagious! If you haven't done so from the beginning, with several subgraphs up and running, now would be a great time to consider taking an incremental approach to adopting Apollo Federation and re-unifying your data in a supergraph! This allows you to implement a multi-team graph that serves as a single source of truth, while allowing your backend teams to evolve their capabilities independently.

Conclusion

GraphQL is a powerful tool to drive business value–both in the short term and the long term. But fully realizing the potential of GraphQL requires a mindset shift with respect to API design. This is acutely true when migrating from REST. Focusing on access patterns—how your data is or will be consumed—puts clients first and helps ensure that the schema you design is consumer friendly and unlocks maximum value, both now and in the future.

Additional resources

Learn more about shifting your REST mindset and the differences in API design with these resources.

- **Adopting a GraphQL Mindset when Migrating from REST**: Watch this webinar to hear from an Apollo GraphQL solution architect on the best practices for migrating from REST. This webinar goes into more detail on the concepts illustrated in this e-book and also features a Q&A.
- **GraphQL vs. REST**: Read this blog post to learn the benefits of GraphQL versus REST.
- **Odyssey**: Check out free, hands-on tutorials on Apollo's official learning platform. Find out how to integrate GraphQL on top of a REST API, as well as how to build a federated supergraph.

About Apollo GraphQL

Apollo GraphQL is the leader in open source and commercial GraphQL technologies. Apollo helps engineering teams build unified graphs to accelerate application development and deliver better, more cohesive experiences. If you want to learn how you can partner with Apollo to scale your GraphQL initiatives.

Contact Sales