GraphQL: A success story for PayPal Checkout

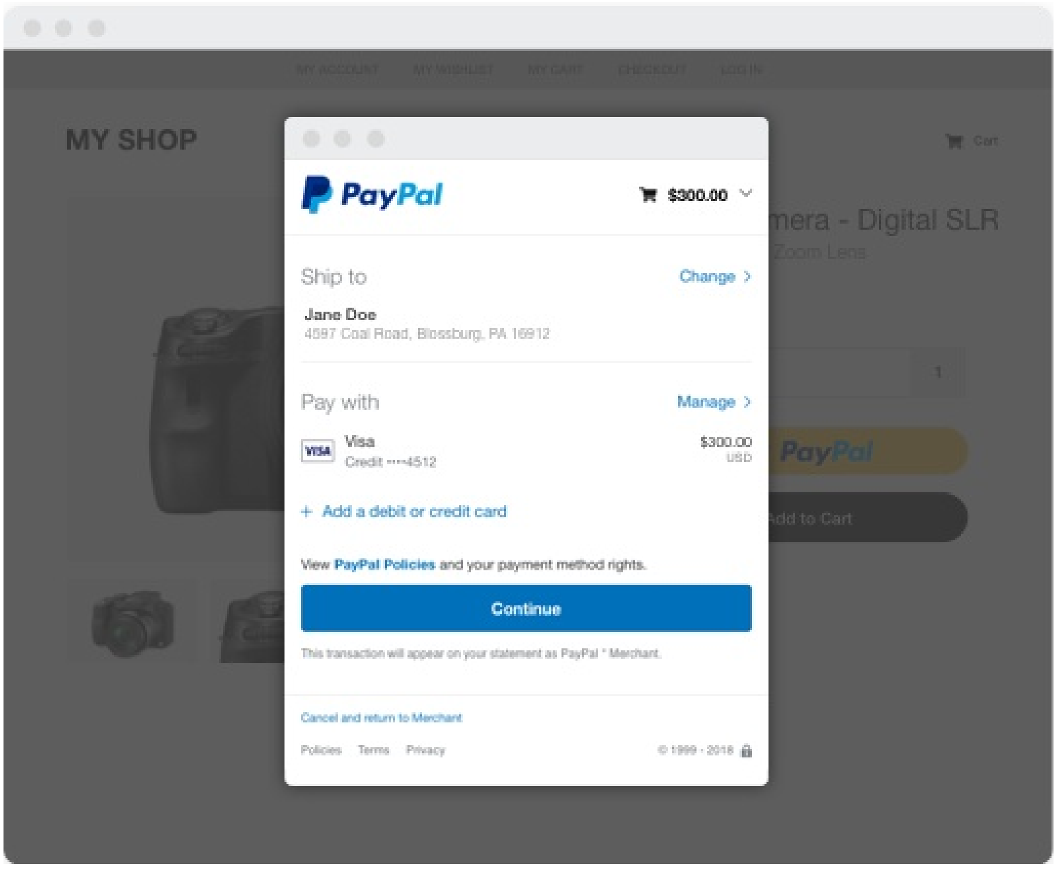
medium.com/paypal-engineering/graphql-a-success-story-for-paypal-checkout-3482f724fb53

At PayPal, we recently introduced GraphQL to our technology stack.

If you haven't heard of GraphQL, it's a wildly popular alternative to REST APIs that is currently taking the developer world by storm!

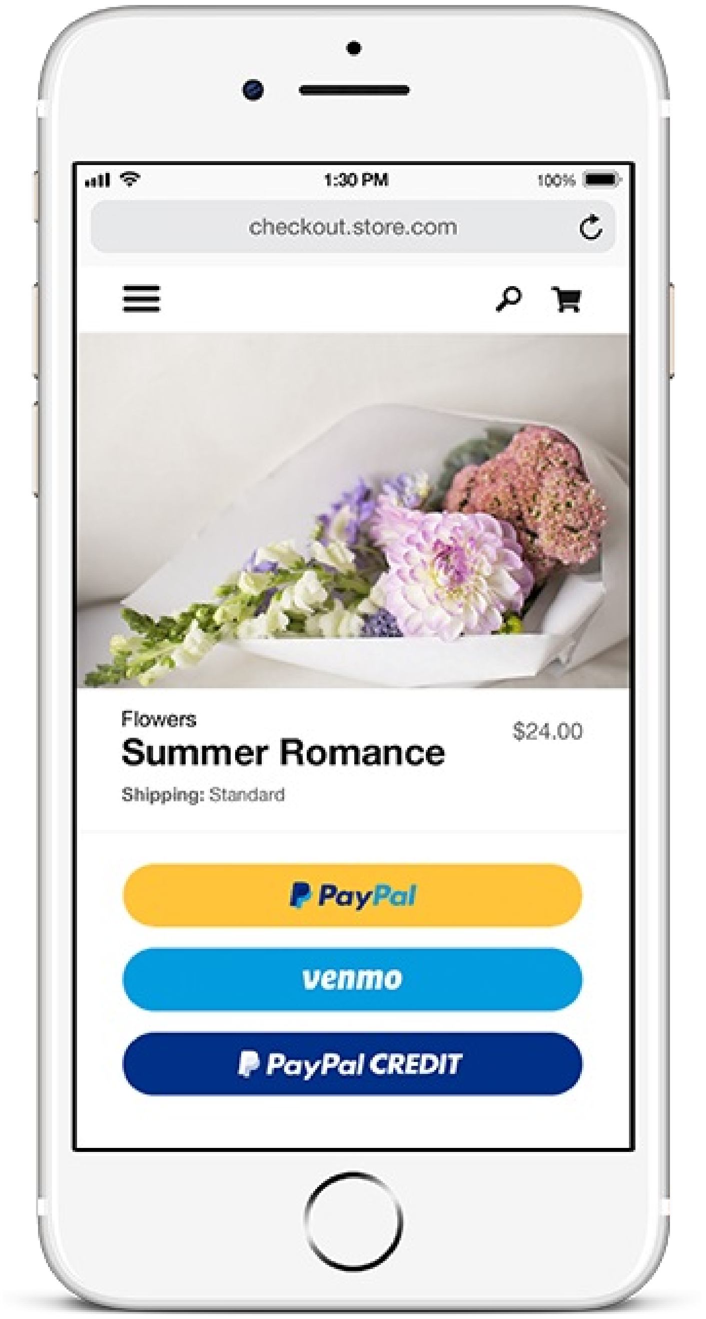
> ***At PayPal, GraphQL has been a complete game changer to the way we think about data, fetch data and build applications.***

This blog post takes a close look at PayPal Checkout and explains our journey from REST to Bulk REST to GraphQL and lessons learned along the way.


MY SHOP
PayPal
$300.00
Ship to
Change
Jane Doe
4597 Coal Road, Blossburg, PA 16912
Pay with
Manage
Visa
Credit ••••4512
$300.00
USD
+ Add a debit or credit card
View PayPal Policies and your payment method rights.
Continue
This transaction will appear on your statement as PayPal * Merchant.
Cancel and return to Merchant
Policies Terms Privacy
© 1999 - 2018
Add to Cart

PayPal Checkout on Web

PayPal Checkout on Mobile



Our API journey

PayPal’s Checkout products spread across many web and mobile apps, supporting millions of users across ~200 countries and has hundreds of experiments running at any time. These apps leverage the same suite of REST APIs to fetch data needed for building UI.

REST

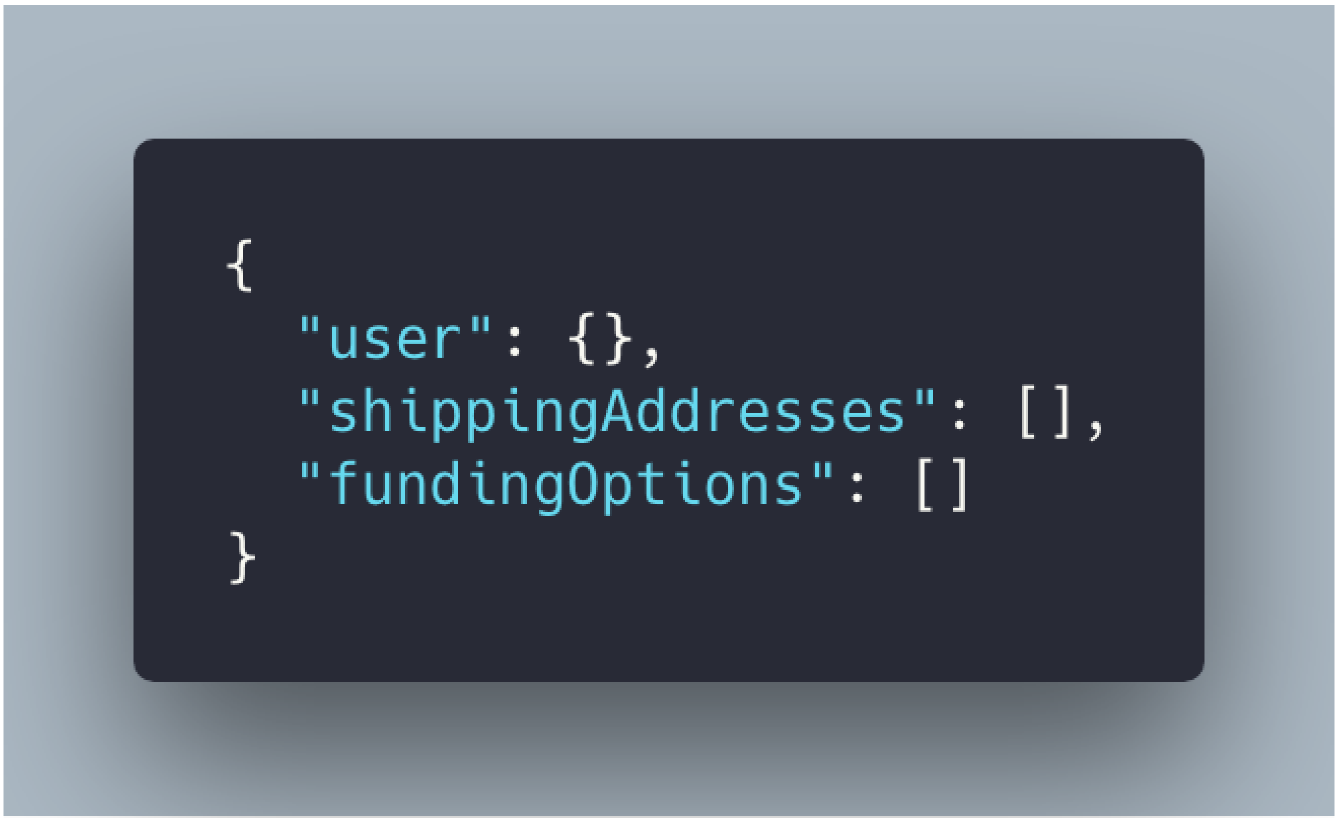
About 4 years ago, we went all in on REST. Our APIs were pretty clean, small and atomic. Things were great in the beginning. REST has strict design principles that are widely understood. REST is a great way to design and implement APIs for your domain.

However, REST's principles don't consider the needs of Web and Mobile apps and their users. This is especially true in an optimized transaction like Checkout. Users want to complete their checkout as fast as possible. If your applications are consuming atomic REST APIs, you're often making many round trips from the client to the server to fetch data. With Checkout, we've found that every round trip costs at least 700ms in network time (at the 99th percentile), not counting the time processing the request on the server. Every round trip results in slower rendering time, more user frustration and lower Checkout conversion. Needless to say, round trips are evil!

Sure, you can build an orchestration API to return all the data you need, but that comes with trade-offs. What do you name it? *GET /landing-page*? Now this seems like a JSON API, not a REST API. With orchestration APIs, your clients are coupled to your server. Any time you add a new feature or experiment, you add it to your API. Now you're overfetching, your performance suffers, and users pay the price.

When new requirements come along, developers face a choice: Should we create a new endpoint and have our clients make another request for that data? Or should we overload an existing endpoint with more data?

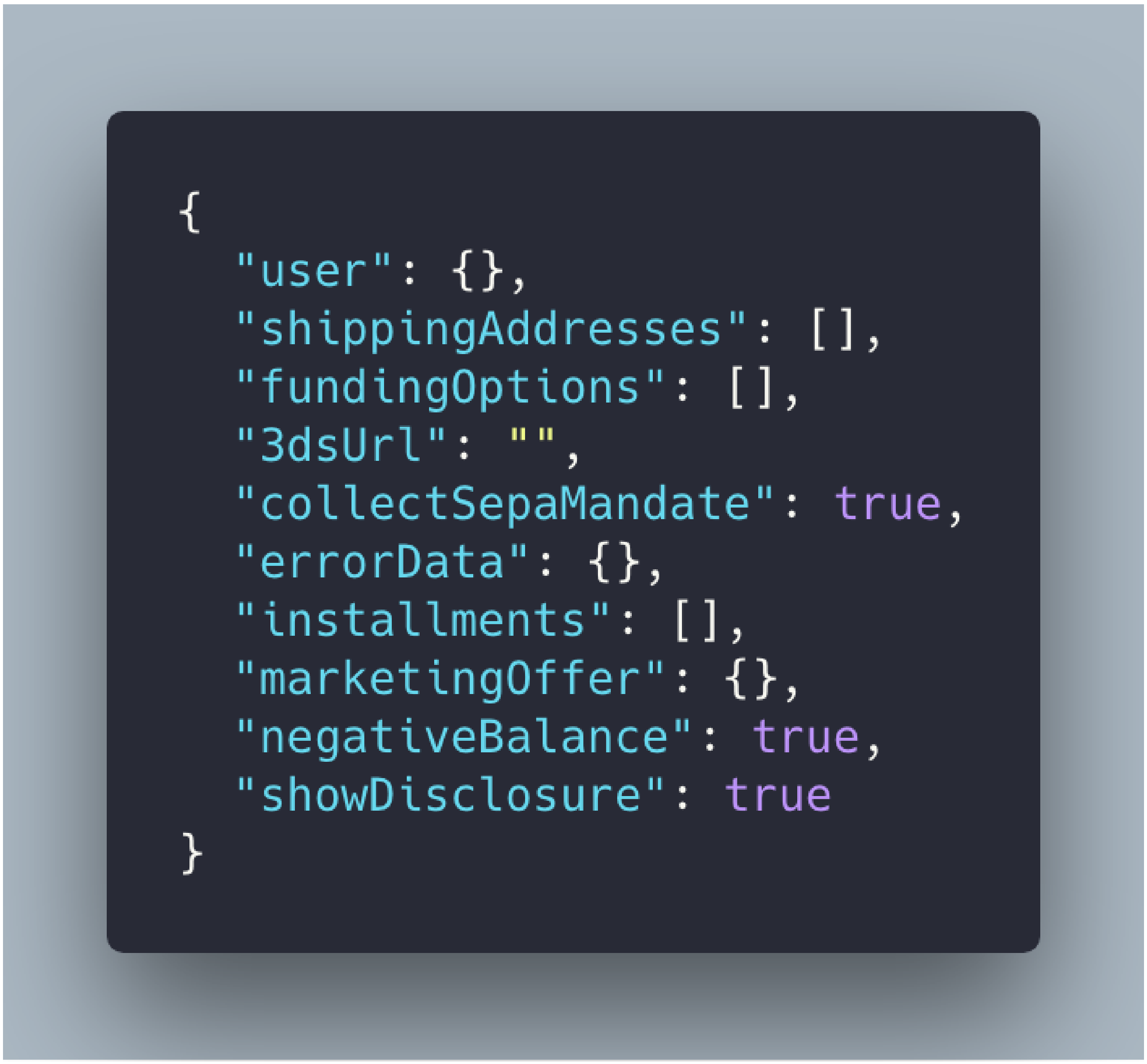
Developers often choose the 2nd option because it's easier to add another field and prevent a client-to-server round trip. Over time, this causes your API to become heavy, kludgy, and serve more than a single responsibility.



```
{
  "user": {},
  "shippingAddresses": [],
  "fundingOptions": []
}
```

The start of an orchestration API. Looks great, right?

We started with great intentions. Our API returned user information, a shipping address and funding options. Everything you need to build a Checkout app. Over time, use cases pile up. Every user pays the cost of these fields even if they don’t need them.



```
{
  "user": {},
  "shippingAddresses": [],
  "fundingOptions": [],
  "3dsUrl": "",
  "collectSepaMandate": true,
  "errorData": {},
  "installments": [],
  "marketingOffer": {},
  "negativeBalance": true,
  "showDisclosure": true
}
```

Yuck.

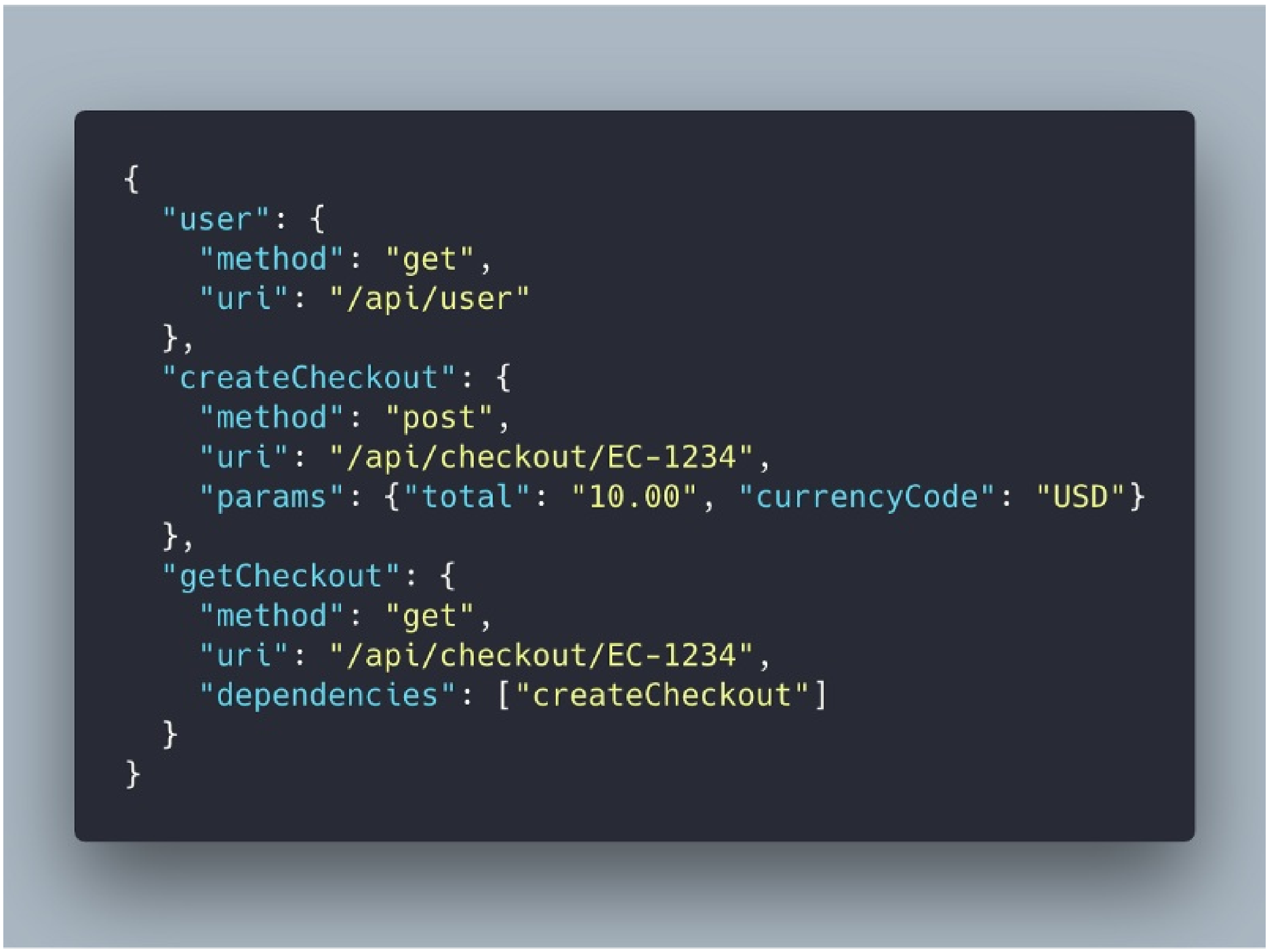
Orchestration APIs seem great at first, but we always regret them later.

Bulk REST

REST wasn't working for us, so we built a Bulk REST API that solves some of these problems. It's an on-the-fly orchestration API that allows clients to determine the size and shape of the response. Bulk REST allowed us to combine atomic REST requests and reduce round trips.

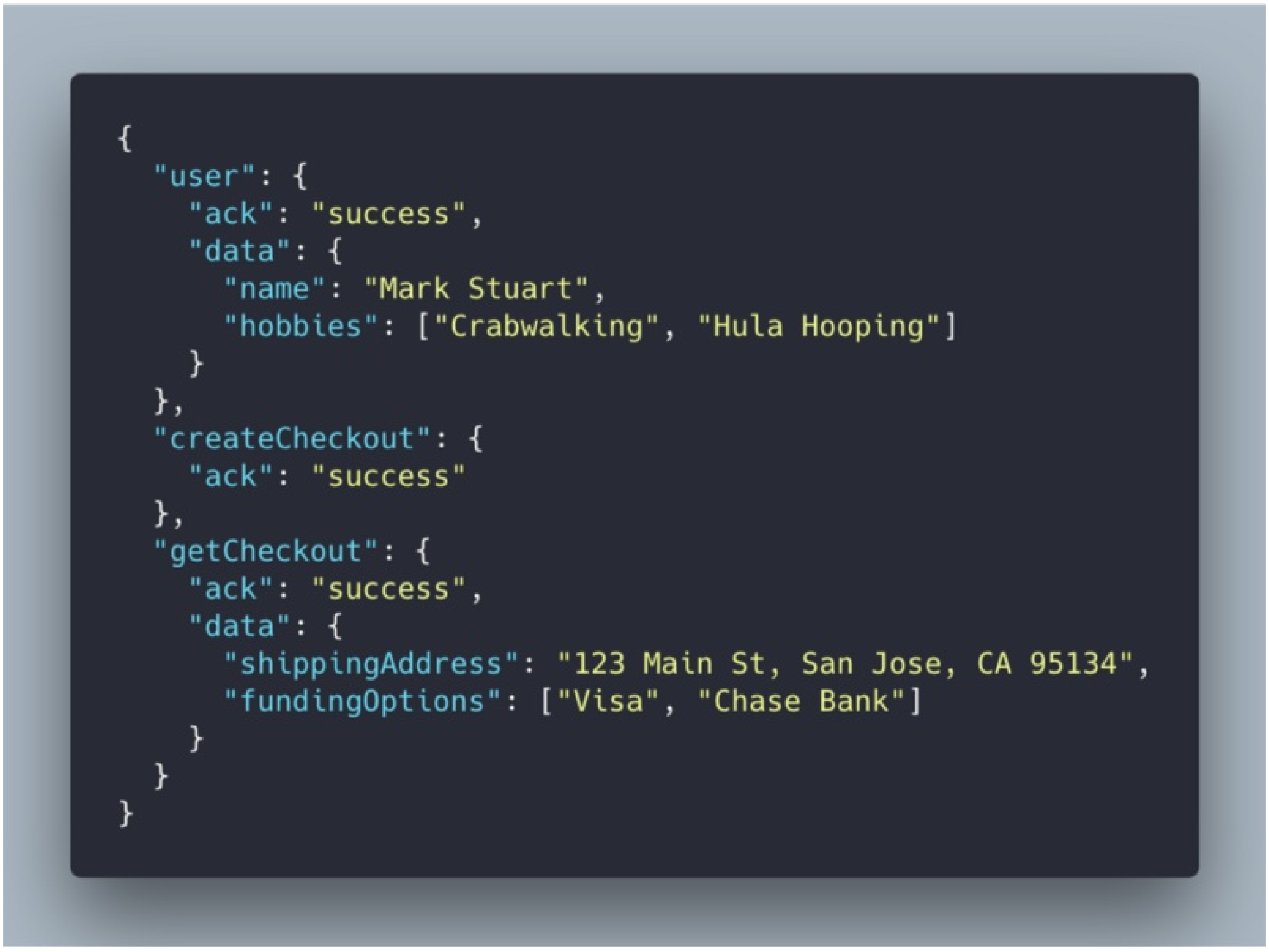
NOTE: This isn't to be confused with Batch REST, our external API offering for clients, which is the much better alternative to Bulk REST that we introduced earlier this year.

Here's an example of a Bulk REST request and response. The request contains a map of REST operations where you specify verbs, endpoints, parameters and dependencies between them.



```
{
  "user": {
    "method": "get",
    "uri": "/api/user"
  },
  "createCheckout": {
    "method": "post",
    "uri": "/api/checkout/EC-1234",
    "params": {"total": "10.00", "currencyCode": "USD"}
  },
  "getCheckout": {
    "method": "get",
    "uri": "/api/checkout/EC-1234",
    "dependencies": ["createCheckout"]
  }
}
```

Bulk REST API Request



```
{
  "user": {
    "ack": "success",
    "data": {
      "name": "Mark Stuart",
      "hobbies": ["Crabwalking", "Hula Hooping"]
    }
  },
  "createCheckout": {
    "ack": "success"
  },
  "getCheckout": {
    "ack": "success",
    "data": {
      "shippingAddress": "123 Main St, San Jose, CA 95134",
      "fundingOptions": ["Visa", "Chase Bank"]
    }
  }
}
```

Bulk REST API Response

But, it was rarely used.

With Bulk REST, we liked that clients control the size and shape of data, not servers. That freed us from having to tweak an orchestration API every time a client's requirements changed. We didn't like that clients had to intimately know how the underlying APIs worked. The request structure was painful enough for developers to not use it. We also wanted to be able to fetch specific fields, not large resources or objects.

Many companies (such as Facebook and Google) that have a Bulk REST offering have the same problem. It's treated as an optimized, but undocumented feature, not the first-class way to fetch data.

Bulk REST was a step in the right direction, but wasn't a game changer.

GraphQL

```
type Project {
  name: String
  tagline: String
  contributors: [User]
}
```

```
{
  project(name: "GraphQL") {
    tagline
  }
}
```

```
{
  "project": {
    "tagline": "A query language for APIs"
  }
}
```

From graphql.org

Last year, our manager (Brian Crescimanno) tasked us with two questions:

> “What’s the best-in-class way to build apps in 2018?”

> “How can we change the way we build apps at PayPal?”

We knew that performance was a problem. We felt we weren’t as productive as we wanted to be. We knew that we weren’t spending enough time building great UI experiences.

When we took a closer look, we found that UI developers were spending less than 1/3 of their time actually building UI. The rest of that time spent was figuring out where and how to fetch data, filtering/mapping over that data and orchestrating many API calls. Sprinkle in some build/deploy overhead. Now, building UI is a nice-to-have or an afterthought.

At that time, we started hearing more and more about GraphQL. We spent a week taking a close look at GraphQL and were impressed. Luckily, we had a new product spun up offering a Mobile SDK for developers wanting to integrate PayPal Checkout into their app. We had 6 weeks to ship it and 3 developers to build it.

> “Let’s use GraphQL!”—Someone

As the team was building out the UI, we were building out the API in parallel. Despite the API not being ready for them to consume, we were *still* feature/code complete ahead of schedule with almost no PayPal tribal knowledge necessary. Developers didn’t have to discover which internal API to call, how to call it and transform that data into something they could use. They inspected our schema to figure out what's possible, wrote a GraphQL query (JSON, without the double quotes) for what they needed, and kept iterating on the UI.

We realized we were onto something. Our developers were productive, our app was quick and our users were happy. **We went all-in on GraphQL.**

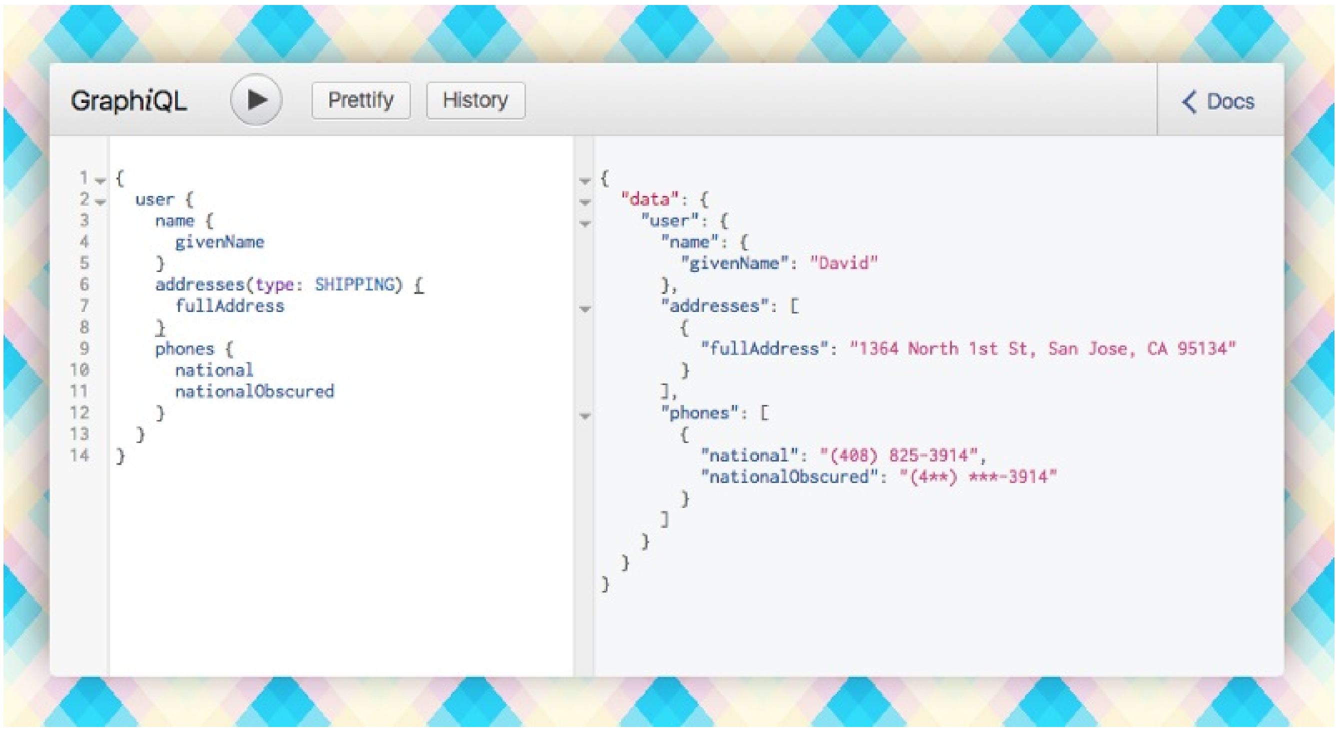
> *We had some big developer experience and performance problems. GraphQL solves this and more.*

Performance — *You only get the data you asked for. No more, no less.* Single round trip. You can query as part of a mutation too!

Flexibility — Clients determine the *size* and *shape* of data, not servers.

Developer productivity — Immediately *productive*, without tribal knowledge. *If you know JSON, you know GraphQL.* Incredible tooling, too!

Evolution — With GraphQL, API developers know *exactly* what *fields* their clients are using and are able to make better choices when deprecating or evolving their APIs. With REST, this isn't possible so you extend and never delete.



```
{
  user {
    name {
      givenName
    }
    addresses(type: SHIPPING) {
      fullAddress
    }
    phones {
      national
      nationalObscured
    }
  }
}
```

```
{
  "data": {
    "user": {
      "name": {
        "givenName": "David"
      },
      "addresses": [
        {
          "fullAddress": "1364 North 1st St, San Jose, CA 95134"
        }
      ],
      "phones": [
        {
          "national": "(408) 825-3914",
          "nationalObscured": "(4**) ***-3914"
        }
      ]
    }
  }
}
```

GraphIQL, a Postman-like IDE for testing queries

Only the beginning

The Mobile SDK was only the beginning. We’re now re-vamping our flagship PayPal Checkout products on top of GraphQL and React. If you’re in the US, there’s a good chance you are using our new stack. It’s *much* faster. 💨

GraphQL is taking PayPal by storm too. A year later and we have over 30 apps/teams either building APIs or consuming APIs. It’s not all sunshine ☀ and rainbows 🌈. We’ve made some mistakes along the way that are worth sharing.

Stay tuned for more posts about GraphQL at PayPal! We will share our thoughts on: schema design best practices, authentication and authorization, instrumenting your GraphQL API, a production-ready checklist, remote schema stitching in practice, and deploying GraphQL in a large organization.