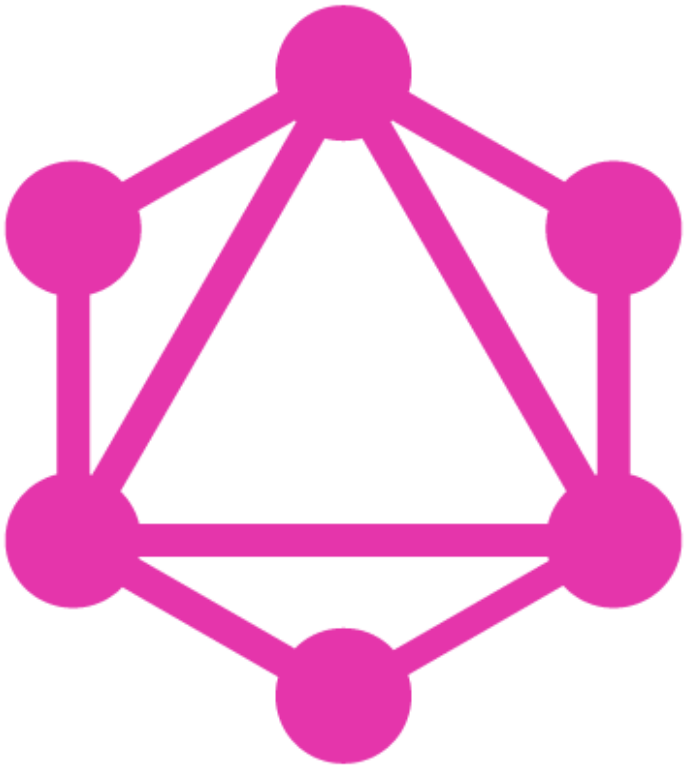
Open Sourcing Lacinia, our GraphQL Library for Clojure

Brandon Carrell Follow

Mar 23, 2017 · 6 min read



We're excited to release Lacinia, our GraphQL library for Clojure! We've been using GraphQL in production for over a year for multiple services.

As part of the release, we'd like to discuss the rationale behind our team's adoption of GraphQL, what problems we faced, how GraphQL has solved them, and where we're headed from here.

This is intended to be a non-technical explanation of why we think GraphQL is a great way to solve problems. For documentation on Lacinia, refer to the Github repository.

Our problems

One of our team's primary responsibilities is to provide Walmart and Sam's Club customers with instant access to their entire history of in-store receipts, directly from their iOS/Android smartphone or web browser. We manage a real-time feed of every purchase and return from over five thousand stores to the tune of 500 receipts per second and up to quadruple that number on Black Friday and throughout the holiday season.

Ultimately, we have a sizable Cassandra database of customers, receipts, and the associations linking them. Our entire server-side stack is written in Clojure, and built to service that incoming data feed efficiently and reliably. For some behind-the-scenes on how all of this works, check out Anthony Marcar's talk at Clojure/West 2015, Clojure At Scale at WalmartLabs.

In the pursuit of serving Walmart and Sam's Club customers the very best we can, we offer our immense amount of data to lots of groups within the company. Each has its own needs and concerns, for example:

- The mobile application teams only want the fields required to populate a concise view. They care about things like a receipt's total purchase price, number of items, date, and location. They also need the ability to pull more detailed information through subsequent calls. Mobile engineers want optimized payloads and the ability to iterate quickly during the design process without waiting for back-end changes. Any extra data sent down to the client is wasted bandwidth.
- The teams developing the websites provide a richer shopping experience targeting a desktop web browser. They're usually working with more viewing space and want to ask for more detailed information than the mobile application teams.
- Other teams often have one or two core questions they need answered and want to be able to query our system for just those specific parameters. Examples include Savings Catcher, which credits customers for purchased items if they have a lower price at a competitor's store, and Customer Protections, which notifies customers if a purchased item is later recalled.

Each of these teams has vastly different requirements on what subset of transaction data it needs to access. Making our data easily consumable, especially since we're a small group of engineers, is a core challenge that a traditional REST API structure wasn't solving for us.

Over time, we found ourselves in the unenviable position of maintaining, extending, and documenting a collection of APIs, each initially built for a specific use case. Each of these API used a different HTTP stack, was configured differently, and had its own ad hoc conventions for URL schemes, query parameters, and so forth, as inspired by the practices at the time they were built. Newer services used Pedestal and Component; older services used Ring handlers and scattered mutable state.

At the same time, we had more teams coming to us interested in building services on top of ours—a good problem to have. Evolving any of these existing APIs was a hard problem. Adding a new field to a JSON response was risky, as it could have unforeseen consequences for some clients that relied on the response being bound to an exact set of known keys. Since our services leverage shared internal libraries, a change in one of these libraries intended for a single service could inadvertently change the behavior of another unrelated service.

We want to be free to evolve and improve our APIs without complex versioning schemes and complicated migration plans. We need to be able to confidently make changes to supporting code without stressing out about undiscovered side-effects in far-flung services. As a back-end team, we decided to stop making product decisions for product teams. Instead, we have empowered each team to query our system in ways that best support the specific product they're building.

. . .

The solution

After Facebook delivered conference talks and writings about GraphQL and its rationale, we were confident the technology would serve as an elegant way of improving our services. There was an issue, though: we're a Clojure team. At the time, GraphQL didn't have a good story for Clojure. Luckily, GraphQL has a robust specification, so we set out to build our own fully realized implementation—Lacinia.

The GraphQL specification is immensely important to us. Our philosophy is that our library needs to speak and understand GraphQL as a lingua franca and have established execution semantics, but we're striving for an idiomatic solution that works best within the Clojure ecosystem. One example of this philosophy is to be as data-driven as possible: the schema definition that declares a GraphQL server's capabilities is written in EDN and uses Clojure data structures. Manipulating this data structure with Clojure code is not only supported, it is encouraged.

Meanwhile, the execution of GraphQL queries using Clojure is very natural. GraphQL, in essence, is a very functional approach: a filtering and transformation process that starts with all possible data and narrows to just the client's requested results. Clojure's use of pure functions and persistent data structures ensures these operations are safe and efficient.

After developing an internal version of Lacinia, we were able to deprecate nearly all of our existing APIs and onboard current and future clients onto our GraphQL-powered services. When you open your Walmart store receipts on your phone or visit samsclub.com and view your receipts on the web, you're being served by Lacinia. If you use Walmart Grocery, you're using Lacinia.

We designed our schemas around our data models and clients can simply ask for what they need. Of course, we can't know *all* possible use-cases, but when a client tells us they can't query for what they want, we can can typically add the missing field or relationship, and deploy it in a matter of minutes. Often, our internal data model already includes the information, and it's only a matter of updating the schema to expose the new field. Because clients control what data they see, adding new fields and types to an existing schema is always safe and always backwards compatible.

Not only that, everything in the schema is fully discoverable and documented through introspection. We serve an instance of the Graph*i*QL (with an *i*!) web-based IDE: this allows developers on other teams to interactively build and execute queries. Lacinia automatically implements the introspection parts of the GraphQL specification; all we have to do is include user-facing documentation on fields and types.

Previously, when assisting other teams, we would throw around ad hoc curl commands, and it was always a challenge reproducing our client's problems. Today, we pass around a Graph*i*QL link which can include the entire query—instant reproducibility.

The vast majority of the time, onboarding a new client to our service is just a matter of giving them the URL for Graph*i*QL, and they can quickly and interactively sketch out their queries and learn about all the fields and types through the interface. If they have problems or questions, they can send us their problematic queries. A surprising offshoot of this is how much I, as a developer of the service, have come to rely on Graph*i*QL for my own daily workflow: it's faster and easier to build a GraphQL query than it is to access the underlying Cassandra database directly. Graph*i*QL, on top of GraphQL, constitutes an exceptionally powerful tool.

We think clojure.spec is the future of robust Clojure systems. Lacinia embraces clojure.spec and strives to leverage it throughout. For instance, custom scalars are defined using conformers. In order to remain on Clojure 1.8 stable while Clojure 1.9 iterates through various alpha versions, we've pulled in clojure-future-spec, a backport of the various clojure.spec features available to Clojure 1.9.

. . .

The future

GraphQL represents a critical piece of our system from both a technical and cultural perspective. Walmart's a big place, and when we engage people in other areas of the company about GraphQL, they're interested in learning more about what we're doing with it and how they can benefit from our work and experiences. We're excited to be continuing development on Lacinia in the open. Lacinia isn't fully compliant with the released specification yet, but we're hoping to get there soon. We're eager to work together with the community, and we'll happily accept and discuss any issues or pull requests.

For more information and technical documentation, visit the Github repository.