GROWTH ACCELERATION PARTNERS

Heavy Lifting APIs: How a strong backend development strategy can scale and streamline your applications

BACK-END API DEVELOPMENT

When Growth Acceleration Partners was charged with building a document search and retrieval application, the sheer volume of data threatened to overwhelm conventional microservices architecture. Read on to learn how GAP kept things running smoothly and users happy.

APIs (application programming interfaces) are vital in enabling your software to interact seamlessly with third-party services or other applications. Yet it is not just about simply connecting and integrating; the most progressive organizations are thinking bigger.

Gartner outlines several API lessons for software engineering leaders — and the overall takeaway is that a modern API strategy needs to cover lifecycle management, developer enablement and monetization potential among others.

This means APIs and wider back-end development involves a lot more than simply getting from A to B. Different strategies for building and designing custom APIs can help improve operational efficiency; streamline and automate workflows and ecosystems; and provide a more unified user experience.

Conventions to Consider

It is worth first taking the opportunity to clarify some of the concepts involved. In a microservices architecture, a microservice exposes APIs and also contains business logic, a data access layer and a database. One of the benefits of microservices, particularly in working with a trusted technology solutions partner such as GAP, is that specialist teams can work on different parts of each application. If you had an eCommerce website or application, you would have teams for checkout, recommendation, authentication, and so on.

The most common convention used for API design is one that conforms to the REST (Representational State Transfer) architectural style. The style is based on the HTTP protocol, but it is important to note REST is not a protocol in its own right. This allows greater flexibility when compared with the older SOAP (Single Object Access Protocol), which is more associated with legacy applications.

One of the tangible benefits of REST over SOAP is in sending and receiving data across multiple formats. SOAP is limited to XML, while REST can work with XML, JSON, HTML and plaintext. SOAP also has hard-and-fast rules with regard to encryption protocols and other tokens. The result is the packets sent are bigger and heavier, leading to a downturn in performance.

Another convention, GraphQL, is an open source query language that is best applied for specific use cases such as mobile applications, microservices applications and other more complex environments. As a query language with a single endpoint, it can answer user queries such as a specific user with items purchased during a specific timeframe. With REST, such a query would involve many more requests to the backend.

Frameworks for Your Back-End

There are myriad frameworks for backend API development, which will be used based entirely on the programming languages the applications are built in. The expertise of GAP engineers can help break down particular frameworks for use cases.

For example, Django — a Python framework — is used for building strong and secure websites that may need a separate administration section, or a significant amount of authentication. Flask and FastAPI, by comparison, are also Python-based, but are more appropriate for lightweight data. And Spring, built in Java, is very strong for cloud-native applications.

GAP's framework expertise includes, along with the aforementioned, Next.js and Express.js (JavaScript) — the former is ideal for React applications, while the latter is for building REST APIs on top of Node.js — and ASP.NET Core (C#). All of these appear in Stack Overflow's top five most desired web frameworks. But the expertise does not end there.

One GAP back-end development project,

which involved an Austin-based SaaS and data analytics firm serving companies in the energy/hydrocarbon industry client in the energy industry, shows the importance of scalability, cost and managing microservices for resource allocation.

A Challenging API Challenge

The client needed to build an API to connect a search application front-end to a massive back-end database of millions of property records, including legal descriptions, mineral leasing, maps and other documents. Filters in the search function meant users might request very large amounts of data, which in turn was put in a "shopping cart" for retrieval.

To keep the database performant, it was built with records containing partial amounts of actual data. It was similar to the "Look Inside" feature on Amazon.com for their books — the actual physical documents were larger than the extractions contained in the database records. Putting a document into the shopping cart and completing the transaction required a separate service to generate a downloadable PDF from the physical document.

The sheer size of the problem dictated breaking the system into architectural domains; each domain got its own microservice. Domains included the search function, shopping cart, PDF creation, authentication and more.

Breaking it down into microservices

The API was coded in the high-level Scala language, and the framework used was Play Framework, which is suited to Scala and Java and provides predictable and minimal CPU and memory consumption, based on a lightweight, stateless, web-friendly architecture. The client had existing investment and skills in Scala, which led to the language decision.

Each day more information and data sets were added to the database, which affected the performance of the database queries and the resources to process the results and show them to the final user. When the initial design was done, we took into account the amount of data and the potential growth in the future, but due to unexpected new large datasets and company acquisitions that came with more data, our estimates were exceeded, and we had to take action. We improved the database design, queries and indexes, and we isolated the search functionality to its own microservice with new API endpoints, scaling it horizontally by adding extra copies to that microservice and vertically by assigning more resources.

Each record in the database contains data extracted from a real physical document; the extraction was done to be able to easily search in the database. However, once the results are at hand, the users are really interested in the original document, which was also scanned and stored as a PDF document. So, users have the option to download the PDF scanned documents once they search for what they are looking for. Handling one small PDF won't affect performance. But there could be dozens or hundreds of results because they may be interested in investigating a whole region in a county, and they need to export all those hundreds of PDF documents. And more importantly, users needed the option to join all those separate PDF files into one big file with hundreds of pages. Handling this merge of hundreds of PDF files into a single file took a lot of memory ,and multiple users exporting files at the same time took a lot of resources.

The GAP team monitored traffic and noticed that each working day between 10 a.m.-12 p.m. and 3-4 p.m., there were traffic spikes because many people were using the app at the same time. You can manually add more instances (copies of the microservice), but cloud providers offer rules for autoscaling, which means you can configure rules in terms of resource usage (i.e., when 90% of the memory allocation is being used), time (i.e., every day between 10 a.m.-12 p.m.), and other factors to automate when to scale up (vertically, when adding resources, or horizontally, when adding instances) or down (vertical to decreasing resources, or horizontal to removing instances).

In this case, rules were configured to scale up horizontally every working day at 10 a.m. and 3 p.m., and to scale down horizontally at 12 p.m. and 5 p.m. If you scale up manually and then forget to scale down, you will have unnecessary instances running, which will add up to your monthly bill. There are stories in the industry of people who forgot to turn off instances or cloud services, and at the end of the month, they received a huge bill (and experience bill shock). It is better to configure automatic rules to avoid this scenario.

Perspective and Experience

Ultimately, while it is worth noting Play Framework finished rock bottom in the Stack Overflow web framework survey, it was considered the best solution for three primary reasons. Alongside the need for minimal, lightweight consumption, and the solution requiring various asynchronous calls, many of the client's other applications were built in Scala. Choosing a framework for the latter reason is not mandatory with microservices, but it helps from the client's perspective if everyone is speaking the same language.

Perhaps the best metaphor for understanding APIs is to imagine being at a restaurant. The customer is the client; the chefs receive the information the customer asks for from the menu; and the waiters are the API, giving you the menu and then taking your order to the kitchen.

By getting a trusted partner like GAP to build custom API solutions, you will not only get a gourmet meal, but sparkling service alongside it, making the entire experience more enjoyable.

09262023