Blog

29
MAR.
2015

CLOUD-NATIVE MICROSERVICES DESIGN CONSIDERATIONS

Patrick McClory cloud-native, architecture

Among modern and (frankly) trendy application design patterns, the 'microservice' architecture seems to be in the limelight for the time being. This design macro-pattern is massively powerful, but like every engineering pattern, can be massively dangerous when applied indiscriminately across an architecture. As an attempt to articulate some of the benefits, I thought it might be interesting to draw out some of the rationale and reasoning behind how this set of patterns can be useful and how I tend to think through the applicaton of a microservices architecture in the real-world. While I won't represent that this is at all 'THE' right way to think about it, I will say that it's worked quite well for my teams as we work with clients and our own systems.

Cloud Native Microservices Principles and Guiding Rationale

As someone who has been in plenty of enterprise architecture, guidance and roadmapping meetings, I can tell you that I continue to come back to focusing on a very short list of meaningful guiding principles as a framework to make decisions under. What I find fascinating is that companies like Spotify have such an amazing culture (see their engineering culture videos here) of autonomy and collaboration that focusing in on the 'how' and 'why' of technical architectures becomes all the more important. In general, I find that I come back to these five 'filters' that I pass ideas through in order to make decisions when managing both green-field application design and existing system replatforming efforts:

- Focus on the Customer
  - That is, focus on the people who use and depend on your systems, whether your customers are actual end users or internal teams
  - Customers don't care about servers, they care about service levels. Design accordingly.
  - Quality of experience is critical and performance SLA's are the key metrics to success - systems should be designed to be highly available by default

    - Manage your product, not your technology. Implementation fads come, go, come back again and change over the years.

- Dig into the context of what's needed
    - In order to balance effort required, complexity and features implemented, understanding the rationale behind requirements is key.
    - Drive technical decisions from concrete needs rather than rationalizing a decision because of something that's packaged or built-in. Put another way, why bother implementing the latest and greatest solution if you don't really need it?

- Start Simple, Iterate on Complexity
    - There's nothing to debate until you have working functional code (Agile Manifesto's measure of success).
    - Improving something that doesn't exist wastes time and perpetuates an engineering subculture of academic bikeshedding.
    - Even if you have to throw the whole thing out, having a bias for action and focus on solving problems and not on producing unnecessary elegant technical solutions.

- Automate Everything
    - Reducing human error, improving reliability and consistency and improving predictability of deployments are key.
    - Adding speed to delivery and reducing the time from the developers' desk to production are secondary (but still important).
    - Pursue this relentlessly from the deployment pipeline all the way through to operational event remediation.

- Never stop evolving
    - Realize that customers change their mind, technology continues to improve and best practices come and go accordingly. Listening to customers is part of the equation, but keeping an eye out for new and better ways to solve problems is a cultural discipline and not simply a process to be followed.
    - Don't be afraid to challenge assumptions and previous decisions. Some of the biggest breakthroughs and best decisions come from the realization that a closely held assumption or opinion isn't true–even if it was true at some point in the past.
    - Don't be afraid of the duplication of effort, rather embrace engineering Darwinism and let the solution that best fits the environment win.

These really are heuristics for me rather than deterministic 'thou shalt' statements or hard-and-fast rules. Frankly, the variety of needs, teams, people, products and solutions on the market makes it nearly impossible to define an absolute best practice, but I find that I quickly start seeking data (or generating my own) when trying to balance these principles in real-world situations. That being said, there's always more to think about, but in my own 80/20 cutoff, these continue to serve me well and, in the spirit of the final point, continually evolve as I learn more about new technologies and listen to clients talk through their own unique problems.

Concrete Patterns and Practices

Here's the part that everyone jumps to first: 'How do I do that?' I've already belabored the point of why the 'why' matters first, but clearly there are a number of great patterns (for now) that enable a lot of great solutions to focus on delighting customers while still allowing engineers to sleep at night.

Design-by-contract and Loose Coupling

Systems and services should publish a contract that customers (consumers of the system) acknowledge as authoritative and consistent. Systems should be held to this contract and should be considered defective if the interface isn't met. Focusing on this concept of interface both allows distributed systems to trust that the contract will be consistent while at the same time allowing the team managing that system to encapsulate the entire system. Encapsulating the entirety of the system allows for decisions about technical implementation to be more focused on the best fit for purpose rather than a single standard or shared asset (as a developer, I'm particularly guilty of leveraging shared RDBMS databases)

Eliminate session state and curate microservices to delivery horizontally scalable and cost efficient systems

Remove the tie between one user and a particular server. Centrally sharing session state isn't any better as it's predicated on previous knowledge and context of other requests which falls down quickly when that information is sitting on a server that fails. There are a ton of great solutions out there to manage a scalable endpoint which routes traffic to various back ends (reverse proxy pattern). Abstracting the usable interface of a service (think: front-end api contract) from the

actual implementation gives you an incredible amount of flexibility and power in terms of delivering solutions that hit the mark on the 'iterate on complexity' principle.

Embrace and extend provider-level and 3rd party solutions

In terms of accelerating application development efforts, leveraging 3rd party tools (SaaS or otherwise) has an incredible amount of ability to give you a leg-up on some of the harder to deal with problems in deploying highly available and massively durable systems. If you're worried about vendor lock-in, shift your mindset to seeing it as technical debt that you're taking on in the event you need to/want to move. Allowing this concern to add unnecessary complexity to your architecture or solution is a very concrete example of premature optimization in action.

Embrace eventual consistency and time-series data, use ACID compliance judiciously

There's a time and a place for most technology patterns, and while RDBMS solutions and ACID compliance have been cornerstones of data management for decades, they're also significant barriers to highly scalable and performant distributed systems. It's not necessarily true that when you move into an eventually consistent, NoSQL solution that you have to lose the concept of a 'transaction' at the data level, but you may potentially shift the responsibility for that concept into the application layer in order to leverage a greater degree of performance, reliability or scalability in the data layer.

CQRS, Event Sourcing and other new and novel patterns and technologies

Admittedly, I'm a huge fan of CQRS (Martin Fowler (http://martinfowler.com/bliki/CQRS.html)). Frankly, building distributed systems is only getting easier with great patterns like this and event-driven solutions like AWS Lambda, but these solutions tend to get awfully complicated very quickly. I tend to start with a single, simple service template. For Java projects, I really like the Dropwizard and Spring Boot projects as great starting points. In the Python world, I tend to gravitate toward Flask and Flask-Restful as my framework of choice to get things rolling, though there are a ton of other great options. Regardless of the technology platform I'm working in, as I iterate on complexity, I tend to reevalute and add to these solutions to build out some of the more complex sets of requirements. In terms of the CQRS and Event Sourcing patterns being interesting approaches, event-driven models aren't anything new. The major differentiator is that it's now quite a bit easier to deploy and manage these patterns with software-defined infrastructure such as Amazon Web Services' SNS and SQS products. Is it perfect? No, but it has forced me to spend a lot of time thinking about how I can apply certain patterns at a smaller scale that are found within these larger, more prescriptive architectures.

Bringing the pieces together

Taking all this into consideration, I use both the mental map in terms of principles and technical implementation tools and concepts to carve out domain models into blocks and then into microservices where appropriate. Balancing the simplicity of development, deployment and management is a tough effort which requires a view of the world that spans the software development, IT operations and support roles within a given organization. Often, what's easy for one is completely opposite of what would be simplest for another perspective (or both!). This is where the focus on automation becomes a critical tool in making more complicated deployments easier to manage and to operate at scale.

RECENT POSTS

Introducing Rusoto (/aws/api/rust/sdk/2015/10/05/introducing-rusoto.html)
05 OCT 2015

Datapipe Acquires AWS Consulting and Professional Services Expert DualSpark (/datapipe/dualspark/2015/09/09/datapipe-acquires-dualspark.html)
09 SEP 2015

Chef Certified Partner (/chef/certified/2015/08/20/chef-certified-partner.html)
20 AUG 2015

Lambda Out Of Beta: The Alpha and Omega (/aws/lambda/2015/08/10/lambda-out-of-beta.html)
10 AUG 2015

Estimating AWS Aurora Costs (/aws/rds/mysql/2015/07/29/aurora-pricing-first-look.html)
29 JUL 2015

SOCIAL NETWORKS

(https://twitter.com/dualsprk) (https://www.linkedin.com/company/dualspark) (https://plus.google.com/112202187107739413150) (https://www.facebook.com/dualsprk)

TAGS

ANSIBLE (/CATEGORIES/ANSIBLE/) CONFIGURATION-MANAGEMENT (/CATEGORIES/CONFIGURATION-MANAGEMENT/) AWS (/CATEGORIES/AWS/) WEBINAR (/CATEGORIES/WEBINAR/) SSH (/CATEGORIES/SSH/) AUTOMATION (/CATEGORIES/AUTOMATION/) CLOUD-NATIVE (/CATEGORIES/CLOUD-NATIVE/) ARCHITECTURE (/CATEGORIES/ARCHITECTURE/) DESIGN (/CATEGORIES/DESIGN/) MIND-MAPPING (/CATEGORIES/MIND-MAPPING/) EC2 (/CATEGORIES/EC2/) 12FACTOR (/CATEGORIES/12FACTOR/) CHEF (/CATEGORIES/CHEF/) IMMUTABLE-IMAGES (/CATEGORIES/IMMUTABLE-IMAGES/) API (/CATEGORIES/API/) MICROSERVICES (/CATEGORIES/MICROSERVICES/) CONTINUOUS-DELIVERY (/CATEGORIES/CONTINUOUS-DELIVERY/) RDS (/CATEGORIES/RDS/) MYSQL (/CATEGORIES/MYSQL/) LAMBDA (/CATEGORIES/LAMBDA/) CERTIFIED (/CATEGORIES/CERTIFIED/) DATAPIPE (/CATEGORIES/DATAPIPE/) DUALSPARK (/CATEGORIES/DUALSPARK/) RUST (/CATEGORIES/RUST/) SDK (/CATEGORIES/SDK/)

DualSpark

DualSpark is an expert Amazon Web Services consulting partner. We bring cutting-edge strategy, cloud native workloads and Automation Everywhere™ to our clients. Our goal isn't just to deliver the project, but to coach your teams and build skills for long-term ownership.

WE'RE SOCIAL

(https://twitter.com/dualsprk) (https://www.linkedin.com/company/dualspark)

(https://plus.google.com/112202187107739413150) (https://github.com/DualSpark)

(https://www.facebook.com/pages/DualSpark/625484174236779)

OUR CONTACTS

info@dualspark.com (mailto:info@dualspark.com)

Offices in San Francisco,
Los Angeles, San Diego,
Seattle and Portland

(800) 501-3275 (tel:18005013275)