TUTORIAL

Use domain-driven design to architect your cloud apps

Integrate DDD into your stacks and build powerful event-sourcing systems

By Xavier Bruhiere | Updated April 12, 2018 - Published April 11, 2018

Cloud Microservices Platform as a Service

Domain-driven design (DDD) is a set of strategies and tools that can help you design systems and manage complexities. You can learn a lot more online about it than we can cover in one tutorial, so we won't dive too deep into the theory and roots of DDD. Instead, we will focus on how it can be used to architect cloud applications—specifically how to integrate it into existing stacks, and especially how it can help you build powerful event-sourcing systems.

Why DDD is worth it

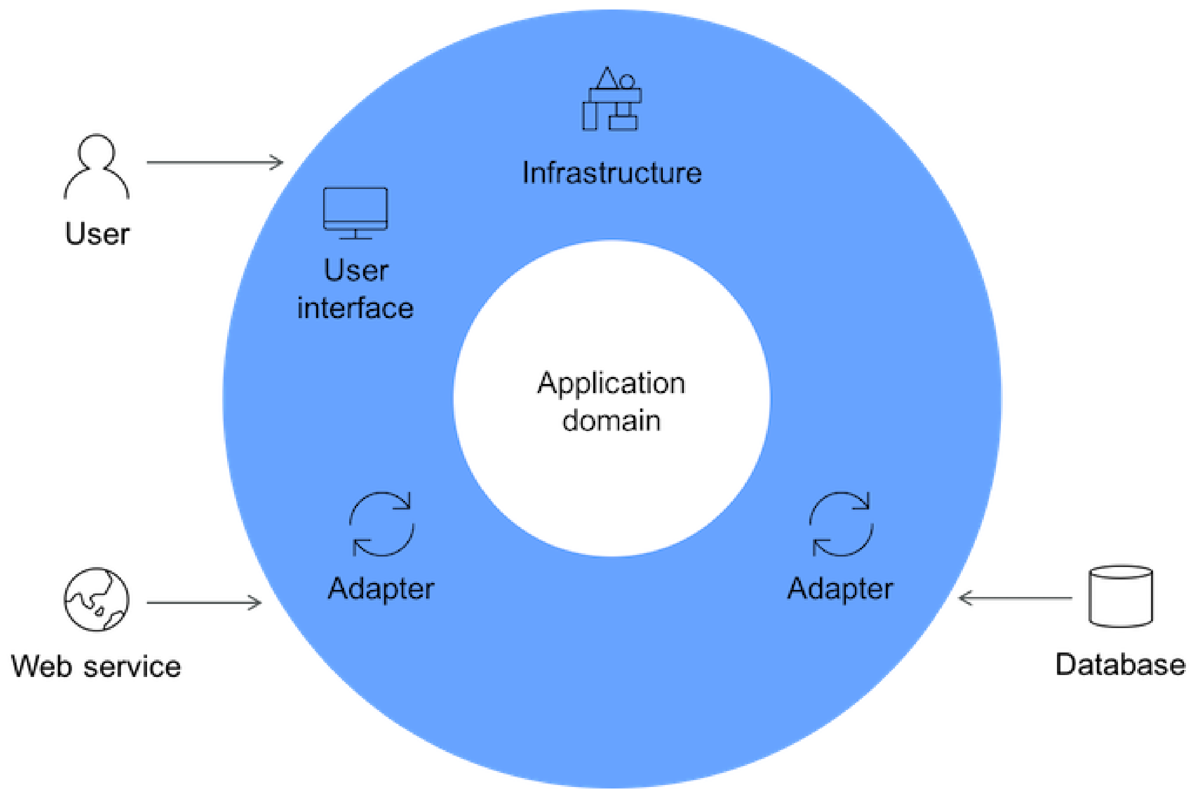
DDD is most effective at doing two things:

- **Minimizing technical debt**—By developing in a way that is easy in the short run but complex in the long run, you may be creating a minimum viable product (MVP). By cutting corners, you may win market share but the project can grow into what is often referred to as a "big ball of mud," which is "a haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle."
- **Controlling inherent project complexity**—Like Paul Graham says, you are trying to solve something so complex that your competitors don't figure out how to outrun you in the market.

Martin Fowler explains how DDD helps solve these problems (see Figure 1):

"DDD is about designing software based on models of the underlying domain. A model acts as a Ubiquitous Language to help communication between software developers and domain experts. It also acts as the conceptual foundation for the design of the software itself—how it's broken down into objects or functions."

Figure 1. Domain-driven design overview



Thinking about software in this way provides a couple of advantages:

- **The project is simplified**—Big problems are cut into more manageable problems that you can focus on.
- **The project is completed sooner**—This is especially true in the long term, because the team can move in parallel and progress with a clearer understanding of the goals of the project.

Bringing the right people together to think about what domains need to be modeled and how they interface with each other is a great playbook for designing services topology. DDD is also a good proxy to use during development when serving real business needs. All too often engineers fall for beautiful architectures where quicker and less exciting approaches would have been enough. Defining the application's context and usage with other characters and in the perspective of real-life models is certainly a good reality check when diving into the project's technical details.

DDD also defines practical implementations to help you reason against real-world problems. Many existing cloud projects involve teams, third-party services, and distributed servers, yet we can identify common architectures and therefore common good practices. Following DDD, you can build more standardized approaches that can be more future proof, resilient to changes out of your control, etc. Bounded Context, for example, deals with large projects by suggesting how to split them into more specific contexts and organize their relationships.

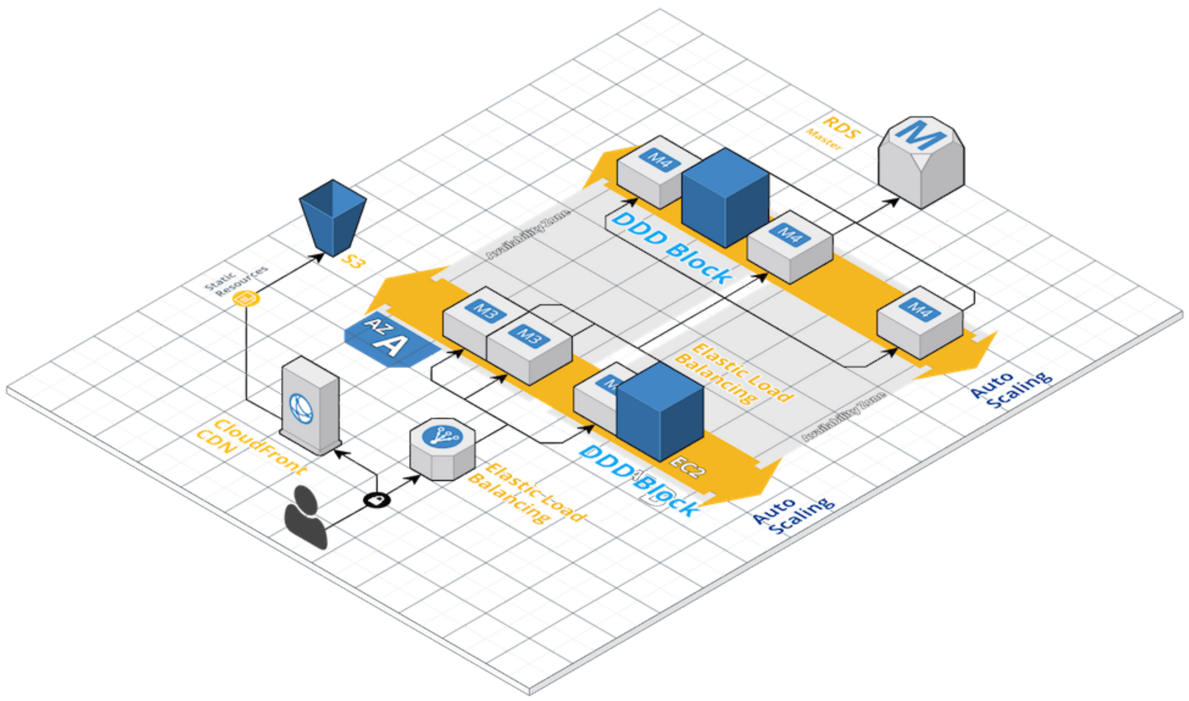
Microservices can help

Many engineering leaders and DDD enthusiasts acknowledge its potential, but don't know how to bring it to their stack. As Eric Evans states, this can be a costly and invasive process and one that is necessary to justify the need and make it right. I will let you judge whether your domains require this level of management, and whether your software needs DDD patterns to avoid being easily broken.

One common pattern among those who have successfully made DDD part of their stack is applying one service at a time. When you gather people and define your domains, you will probably find opportunities for new services and then you can design those services in a DDD fashion. This is powerful, because you can recognize all of the agile benefits. If you start with only one service, it makes the process of integration easier to learn for newcomers and easier to iterate on and alter to the way you already work. This process also makes it easy to reverse. Chances are, most of the code will still be useable and the knowledge gained to identify domains will still be useful.

What's more, all of these pattern and process details will be hidden behind some kind of remote interface. Since you're working with a CRUD/REST API or a GraphQL endpoint, the other parts of your stack should be easy to modify. And as you design code following DDD practice and decouple clients from core logic, it will become even simpler (see Figure 2).

Figure 2. Decoupling clients



This is especially true if microservices are already in place, because you can recognize the same advantages—specifically, smaller and better bounded components. These components can be easily modified, as long as the other components know how to interface with them. This notion of interface is key, and is directly linked to the importance of good bounded contexts in DDD and to good decoupling of services in microservices.

But just because your architecture has smaller and better bounded components doesn’t necessarily mean it will be easier to integrate DDD. Here are a couple of examples:

- **You already have microservices in place**—In this case, the good news is you probably don’t need to develop all of the details that are required for microservices to work, such as service discovery, log management, and deployment process. However, bad services topology, or the architecture and manner in which the services scale, can create a substantial technical debt; this may be harder to fix because you might be forced to move features between pieces of code that don’t run on the same servers.
- **You have a monolith in place**—This is probably one of the most common startup cases currently in use. In this case, you are moving brittle code into its own service, so DDD can be very helpful. But creating your project from the ground up with code from scratch poses its own challenges. For example, developing on only one repository can limit coordination and consistency between teams. Nonetheless, DDD still provides many benefits, such as limiting technical debt.

Communication within companies can be difficult to get right. It is usually valuable to have a common framework that makes it easier for departments to talk to one another about business issues. This can help software engineers working together on a team to better understand why their code makes sense and how to prioritize problems. It can make it easier for project managers to use the interface. And sales professionals and executive leaders can better understand where the IT is on the roadmap and what the challenges and expectations are.

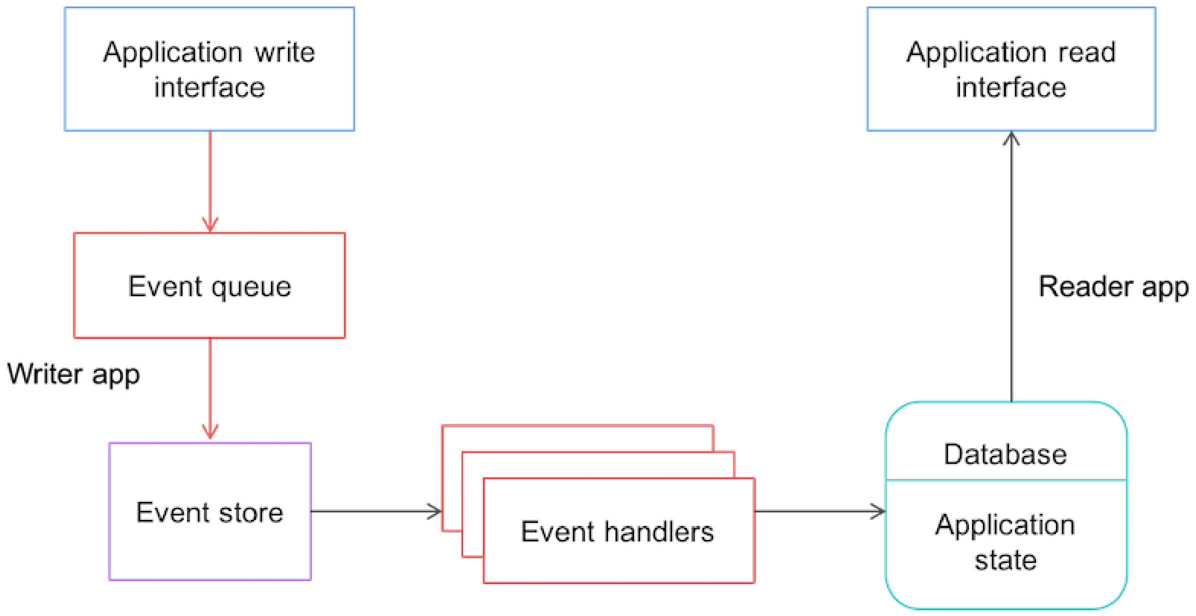
So, investing time and energy in persuading developers to embrace the DDD philosophy can provide many more benefits than just following a new software pattern or strategy. This approach can help team members better communicate within the company, and it gives them a great set of tools for solving problems. All of the trade-offs of implementing microservices to solve your model problems can bring clarity and a new vocabulary under DDD.

Now that we have covered in depth the ways in which DDD interacts with microservices, let's take a look at a less obvious and more generic area: event sourcing.

Event sourcing: A match made in heaven

Event sourcing is a strategic pattern that forces you to think of your application as a series of events that you need to store any time something changes in your architecture (see Figure 3). This enables you to review, understand, replay, debug, and do whatever is required for your history of events. This is discussed today as data streaming, so state management becomes relevant to modern development. Read this blog post by Jay Kreps to get a deeper understanding of all the implications and benefits of real-time data's unifying abstraction. This is the kind of idea that powers relational databases and companies like LinkedIn.

Figure 3. Event sourcing



You can indeed see this pattern as an atomic, append-only stream of logs—meaning there is no deletion or update of what happened during the application's lifetime. This means you can read the log history and reconstruct the application state at any time in the past. A pattern is useful if you fear, for example, losing data while processing it, but still have the original raw materials. You can also partition this storage and still be able to reconcile a consistent history, pretty much the same way Kafka makes its messaging system scalable and useful.

This can be difficult to implement and it can take many shapes (specific storage, state management), but the types of architectures and tools that are possible on top of the abstraction make it worth the effort. Everything I have detailed in this article can give you a better understanding of event sourcing. If, for instance, you have implemented microservices that solve a specific problem by defining clear interfaces, vocabularies, and bounded contexts, then you are in a very good position to look into event sourcing.

The decoupling, as explained in Kreps' post, helps you understand data flow in a clear manner, and you can probably organize it as events that trigger actions. This is often implemented as task queues, so in this case you can get started by ensuring that you are storing all of the original events, and you can stop polling for changes and instead wait for inputs.

The consistent vocabulary will help you define what those events actually are. Technical constraints may come into play (like caching database queries or limiting connection bottlenecks), but you can think about the problem with the right business goal and context in mind. This will make it much more future-proof. Basing your reasoning on your core values, which remain the same throughout the project development lifecycle, makes for a consistent evolution. This integration is also safer for the same reasons that I covered in the microservices section above. Again, there are many benefits, such as easier team management and building tools on top of existing architectures.

Conclusion

Although the content of this article is theoretical in nature, it reflects the work that I have been engaged in during the last few months at my current company, Kpler. We have been working on approaching our processes through DDD, so I have seen first-hand the benefits that such an approach can provide to a team. We were very successful at implementing bits an pieces, here and there, of DDD and event sourcing, where we needed to meet strong requirements for data quality and load. Of course, it wasn't easy—we had to educate the team on the new projects and justify the additional time spent on experimentations and failures. But out of it came stronger services with better flexibility for integration into our stack and future features.

So I hope this article inspires you to investigate and experiment with DDD on your own.

SOCIAL

CONTENTS

Why DDD is worth it

Microservices can help

Event sourcing: A match made in heaven

Conclusion

RESOURCES

"How To Make Wealth" by Paul Graham

"What every software engineer should know about real-time data's unifying abstraction" by Jay Kreps

Try IBM Cloud for free