MICROSERVICES VS. MONOLITHS

THE REALITY BEYOND THE HYPE

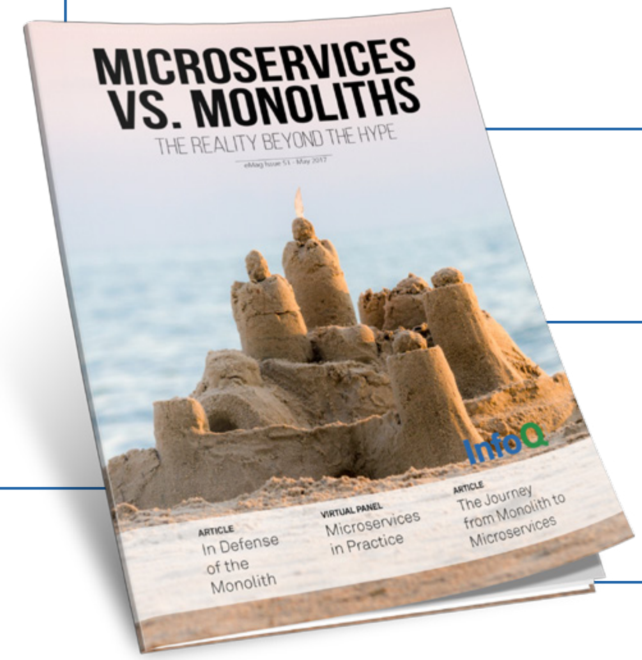
eMag Issue 51 - May 2017

Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS

Chris Richardson lays out clear guidelines for building a microservices architecture around the concepts of aggregates, event sourcing and CQRS. He sees DDD bounded contexts and aggregates as the building blocks for microservices.

In Defense of the Monolith

Dan Haywood believes modular monoliths are a better option than microservices when dealing with a complex domain that doesn't need to support internet-scale traffic. Rather than the "big ball of mud," a modular monolith can be maintainable, given the proper discipline.


MICROSERVICES VS. MONOLITHS
THE REALITY BEYOND THE HYPE
InfoQ
ARTICLE In Defense of the Monolith
VIRTUAL PANEL Microservices in Practice
ARTICLE The Journey from Monolith to Microservices

The Journey from Monolith to Microservices: A Guided Adventure

Mike Gehard describes a journey from a monolith to microservices. Again following DDD principles, bounded contexts help create structure in an existing monolith, which simplifies the transition to microservices.

Evolution of Business Logic from Monoliths through Microservices, to Functions

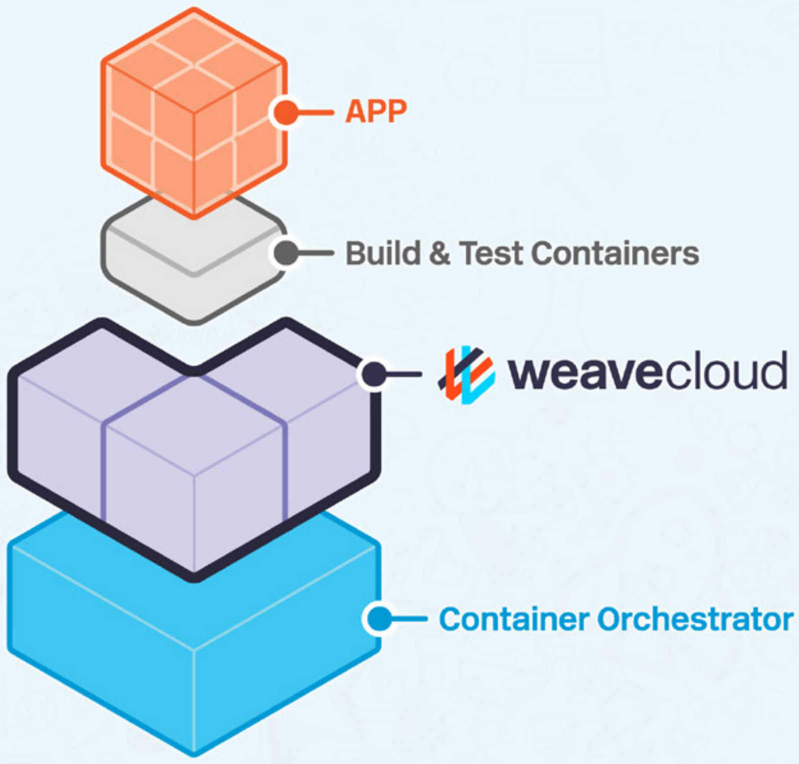
Adrian Cockcroft looks toward the future, going beyond microservices to functions. Significant advancements in technology, coupled with changes in how organizations and development teams are structured, have allowed us to get to where we are today, and are providing a path forward to ever-smaller deployable components.

Virtual Panel: Microservices in Practice

A virtual panel of developers and architects discuss microservices in practice. The panelists highlight where microservices are used successfully, what tools, technologies and patterns developers need to learn, and how microservices are continuing to evolve.

Microservices make your application scale.

We make operating it simple.



Weave Cloud is an Operations-as-a-Service Platform for Application Developers and DevOps teams building containerized applications. It simplifies deployment, observability and monitoring for containers and microservices through a single integrated web console.

START A FREE TRIAL →

weaveworks

THOMAS BETTS

is a Principal Software Engineer at IHS Markit, with two decades of professional software development experience. His focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

A LETTER FROM THE EDITOR

It's clear that microservices are the current hot architectural pattern, covered extensively on blogs, in the tech news at software conferences. Several times a week, InfoQ has a news item, podcast or presentation mentioning microservices.

But is it all just hype and a pattern useful at startups working on greenfield applications? And is the dreaded monolith, the antithesis of a microservices architecture, really dying a slow death, barely limping along until a complete replacement is built?

The reality looks closer to the compromises that any software architect will recognize. Both architectures come with pros and cons, and it is important to understand all the trade-offs before deciding that the monolith has to go, and microservices are the answer. A common theme is managing complexity, and successful solutions (with either architecture) strongly embrace concepts of Domain-Driven Design.

This eMag includes articles written by experts who have implemented successful, maintainable systems across the spectrum of microservices to monoliths.

Chris Richardson lays out clear guidelines for building a microservices architecture around the concepts of aggregates, event sourcing and CQRS. He sees DDD bounded contexts and aggregates as the building blocks for microservices.

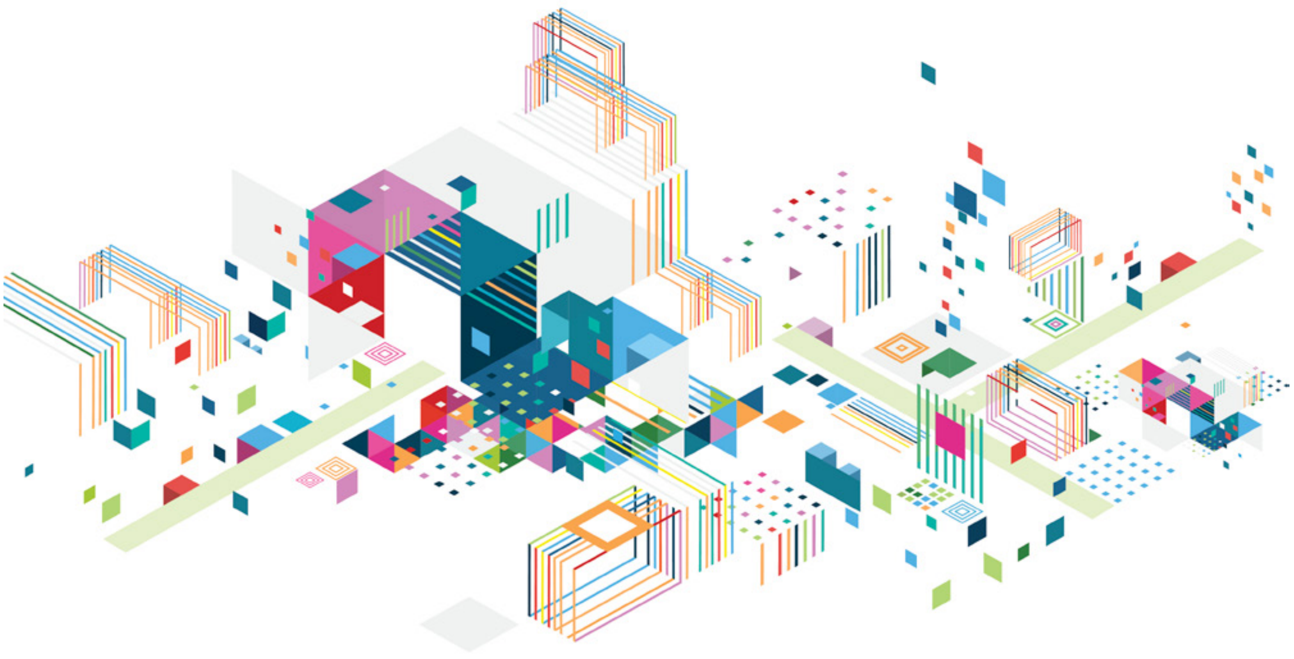
Dan Haywood believes modular monoliths are a better option than microservices when dealing with a complex domain that doesn't need to support internet-scale traffic. Rather than the "big ball of mud," a modular monolith can be maintainable, given the proper discipline.

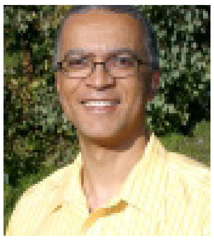
Bridging these two viewpoints, Mike Gehard describes a journey from a monolith to microservices. Again following DDD principles, bounded contexts help create structure in an existing monolith, which simplifies the transition to microservices.

A virtual panel of developers and architects discuss microservices in practice. The panelists highlight where microservices are used successfully, what tools, technologies and patterns developers need to learn, and how microservices are continuing to evolve.

Finally, Adrian Cockcroft looks toward the future, going beyond microservices to functions. Significant advancements in technology, coupled with changes in how organizations and development teams are structured, have allowed us to get to where we are today, and are providing a path forward to ever-smaller deployable components.

Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS





Chris Richardson is a developer and architect. He is a Java Champion and the author of POJOs in Action, which describes how to build enterprise Java applications with frameworks such as Spring and Hibernate. Richardson was also the founder of the original CloudFoundry.com. He consults with organizations to improve how they develop and deploy applications and is working on his third startup. You can find Richardson on Twitter @crichardson and on Eventuate.

Originally published in two parts, on Oct 03, 2016 and Jan 13, 2017.

The microservice architecture is becoming increasingly popular. It is an approach to modularity that functionally decomposes an application into a set of services.

It enables teams developing large, complex applications to deliver better software faster. They can adopt new technology more easily since they can implement each service with the latest and most appropriate technology stack. The microservices architecture also improves an application's scalability by enabling each service to be deployed on the optimal hardware.

Microservices are not, however, a silver bullet. In particular, domain models, transactions and queries are surprisingly resistant to functional decomposition. As a result, developing transactional business applications using the microservice architecture is challenging. In this article, I describe a way to develop microservices that solves these problems by using Domain Driven Design, Event Sourcing and Command Query Responsibility Segregation (CQRS). Let's first look at the challenges developers face when writing microservices.

Microservice Development Challenges

Modularity is essential when developing large, complex applications. Most modern applications are too large to be developed by an individual. They are also too complex to be understood by a single person. Applications must be decomposed into modules that are developed and understood by a team of developers. In a monolithic application, modules are defined using pro- ▶

KEY TAKEAWAYS

The microservice architecture functionally decomposes an application into services, each of which corresponds to a business capability.

A key challenge when developing microservice-based business applications is that transactions, domain models, and queries resist decomposition.

A domain model can be decomposed into Domain Driven Design aggregates. DDD aggregates are the building blocks for microservices.

Event Sourcing is a technique for reliably updating state and publishing events that overcomes limitations of other solutions. Event sourcing is a great way to implement an event-driven microservices architecture.

Event sourcing can create challenges for queries, but these are overcome by following CQRS guidelines and materialized views.

gramming language constructs such as Java packages. However, this approach does not tend to work well in practice. Long lived monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. Each service corresponds to a business capability, which is something an organization does in order to create value. A microservices-based online store, for example, consists of various services including Order Service, Customer Service, and Catalog Service.

Each service has an impermeable boundary that is difficult to violate. As a result, the modularity of the application is much easier to preserve over time. The microservice architecture has other benefits including the ability to deploy and scale services independently.

Unfortunately, decomposing an application into services is not as easy as it sounds. Several different aspects of applications - domain models, transactions and

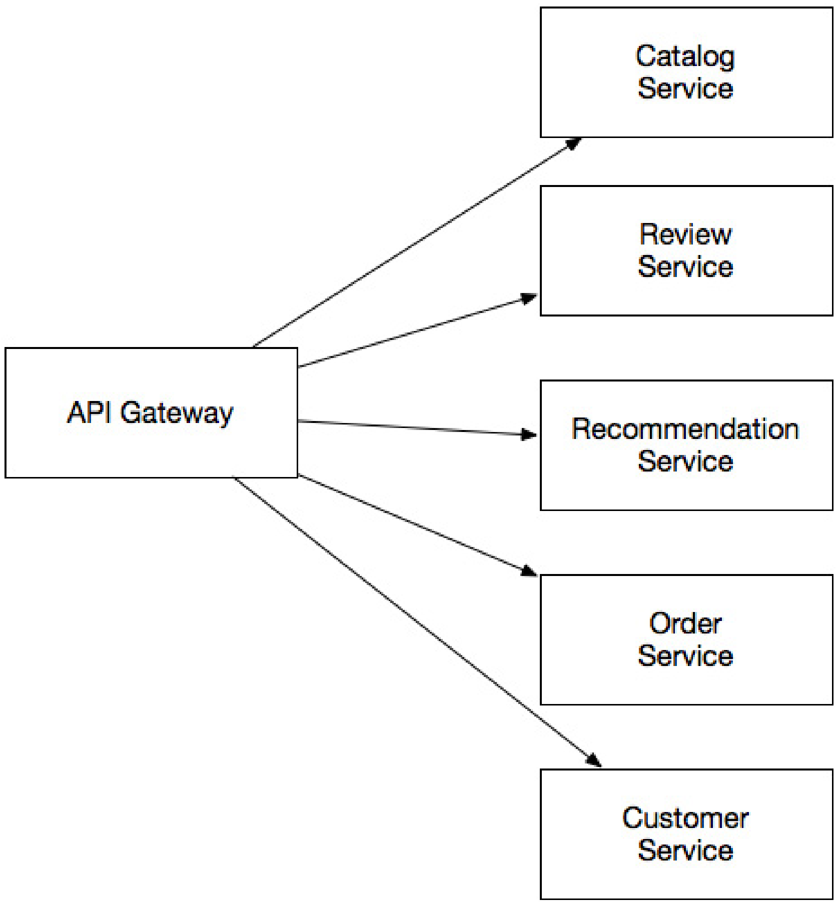


Figure 1 - Microservices within an online store

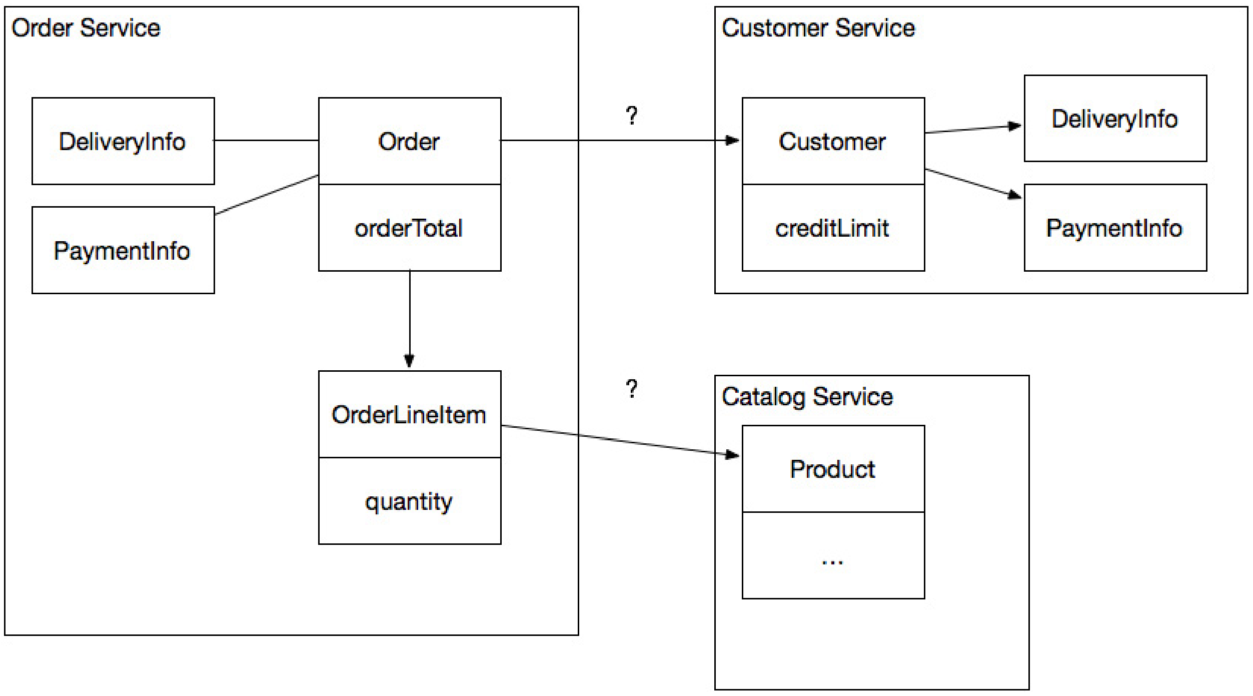


Figure 2 - Domain model for an online store

queries - are difficult to decompose. Let's look at the reasons why.

Problem #1 - Decomposing a Domain Model

The Domain Model pattern is a good way to implement complex business logic. The domain model for an online store would include classes such as Order, OrderLineItem, Customer and Product. In a microservices architecture, the Order and OrderLineItem classes are part of the Order Service, the Customer class is part of the Customer Service, and the Product class belongs to the Catalog Service.

The challenge with decomposing the domain model, however, is that classes often reference one another. For example, an Order references its Customer and an OrderLineItem references a Product. What do we do about references that want to span service boundaries? Later on you will see how the concept of an Aggregate from Domain-Driven Design (DDD) solves this problem.

Microservices and Databases

A distinctive feature of the microservice architecture is that the data owned by a service is only accessible via that service's API. In the online store, for example, the OrderService has a database that includes the ORDERS table and the CustomerService has its database, which includes the CUSTOMERS table. Because of this encapsulation, the services are loosely coupled. At development time, a developer can change their service's schema without having to coordinate with developers working on other service. At runtime, the services are isolated from each other. For example, a service will never be blocked waiting for a database lock owned by another service. Unfortunately, the functional decomposition of the database makes it difficult to maintain data consistency and to implement many kinds of queries.

Problem #2 - Implementing Transactions That Span Services

A traditional monolithic application can rely on ACID transactions to enforce business rules (a.k.a. invariants). Imagine, for example, that customers of the online store have a credit limit that must be checked before creating a new order. The application must ensure that potentially multiple concurrent attempts to place an order do not exceed a customer's credit limit. If Orders and Customers reside in the same database it is trivial to use an ACID transaction (with the appropriate isolation level) as follows:

```
BEGIN TRANSACTION
…
SELECT ORDER_TOTAL
 FROM ORDERS WHERE CUSTOM-
ER_ID = ?
…
SELECT CREDIT_LIMIT
FROM CUSTOMERS WHERE CUS-
TOMER_ID = ?
…
INSERT INTO ORDERS …
…
COMMIT TRANSACTION
```
▶

Sadly, we cannot use such a straightforward approach to maintain data consistency in a microservices-based application. The ORDERS and CUSTOMERS tables are owned by different services and can only be accessed via APIs. They might also be in different databases.

The traditional solution is Two-phase commit (2PC, a.k.a. distributed transactions) but this is not a viable technology for modern applications. The CAP theorem requires you to chose between availability and consistency, and availability is usually the better choice. Moreover, many modern technologies, such as most NoSQL databases, do not even support ACID transactions let alone, 2PC. Maintaining data consistency is essential so we need another solution. Later on you will see that the solution is to use an event-driven architecture based on a technique known as event sourcing.

Problem #3 - Querying and Reporting

Maintaining data consistency is not the only challenge; another problem is querying and reporting. In a traditional monolithic application it is extremely common to write queries that use joins. For example, it is easy to find recent customers and their large orders using a query such as:

```
SELECT *
FROM CUSTOMER c, ORDER o
WHERE
   c.id = o.ID
     AND o.ORDER_TOTAL >
100000
     AND o.STATE =
'SHIPPED'
     AND c.CREATION_DATE
> ?
```

We cannot use this kind of query in a microservices-based online store. As mentioned earlier, the ORDERS and CUSTOMERS tables are owned by different services and can only be accessed via APIs. Some services might not even be using a SQL database. Others, as you will see below, might use an approach known as Event Sourcing, which makes querying even more challenging. Later on, you will learn that the solution is to maintain materialized views using an approach known as Command Query Responsibility Segregation (CQRS). But first, let's look at how Domain-Driven Design (DDD) is an essential tool for the development of domain model-based business logic for microservices.

DDD Aggregates are the Building Blocks of Microservices

As you can see, there are several problems that must be solved in order to successfully develop business applications using the microservice architecture. The solution to some of these problems can be found in the must-read book Domain-Driven Design by Eric Evans. This book, published in 2003, describes an approach to designing complex software that is very useful when developing microservices. In particular, Domain-Driven Design enables you to create a modular domain model that can be partitioned across services.

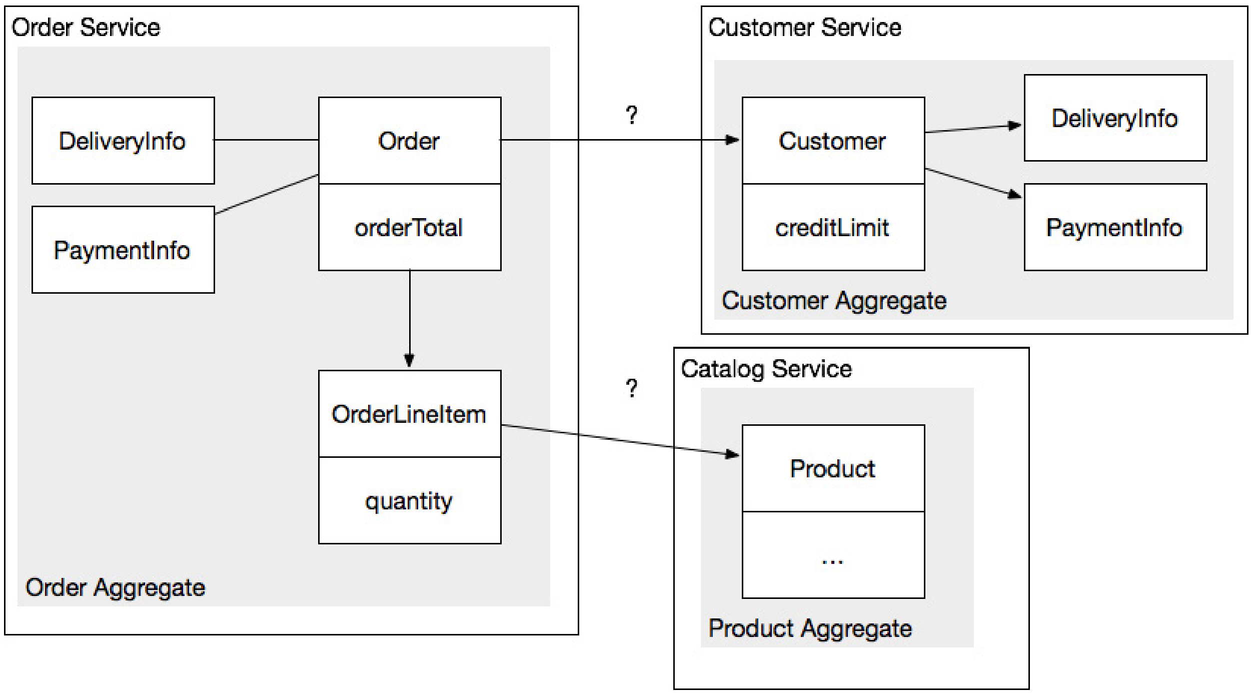


Figure 3 - Aggregates for an online store

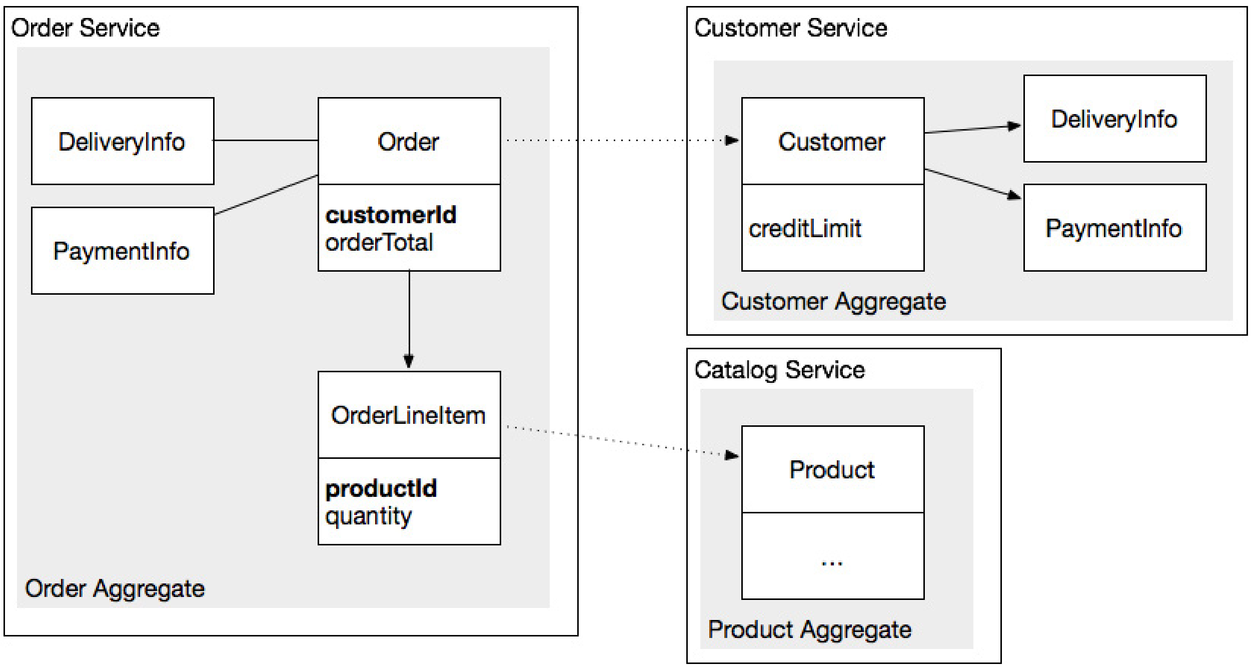


Figure 4 - Inter-aggregate references for an online store

What is an Aggregate?
In Domain-Driven Design, Evans defines several building blocks for domain models. Many have become part of everyday developer language including entity, which is an object with a persistent identity; value object, which is an object that has no identity and is defined by its attributes; service, which contains business logic that doesn't belong in an entity or value object service; and repository, which represents a collection of persistent entities. One building block, the aggregate, has mostly been ignored by developers except by those who are DDD purists. It turns out, however, that aggregates are key to developing microservices.

An aggregate is a cluster of domain objects that can be treated as a unit. It consists of a root entity and possibly one or more other associated entities and value objects. For example, the domain model for the online store contains aggregates such as Order and Customer. An Order aggregate consists of an Order entity (the root), and one or more OrderLineItem value objects, along with other value objects such as a delivery Address and PaymentInformation. A Customer aggregate consists of the Customer root entity along with other value objects such a DeliveryInfo and PaymentInformation.

Using aggregates decomposes a domain model into chunks, which are individually easier to understand. It also clarifies the scope of operations such as load and delete. An aggregate is usually loaded in its entirety from the database. Deleting an aggregate deletes all of the objects. The benefit of aggregates, however, goes far beyond modularizing a domain model. That is because aggregates must obey certain rules.

Inter-Aggregate References Must Use Primary Keys
The first rule is that aggregates reference each other by identity (e.g. primary key) instead of object references. For example, an Order references its Customer ▶

DDD aggregates are key to developing microservices.

using a customerId rather than a reference to the Customer object. Similarly, an OrderLineItem references a Product using a productId.

This approach is quite different than traditional object modeling, which considers foreign keys in the domain model to be a design smell. The use of identity rather than object references means that the aggregates are loosely coupled. You can easily put different aggregates in different services. In fact, a service's business logic consists of a domain model that is a collection of aggregates. For example, the OrderService contains the Order aggregate and the CustomerService contains the Customer aggregate.

One Transaction Creates or Updates One Aggregate

The second rule that aggregates must obey is that a transaction can only create or update a single aggregate. When I first read about this rule many years ago, it made no sense! At the time, I was developing traditional monolithic, RDBMS-based applications and so transactions could update arbitrary data. Today, however, this constraint is perfect for the microservice architecture. It ensures that a transaction is contained within a service. This constraint also matches the limited transaction model of most NoSQL databases.

When developing a domain model, a key decision you must make is how large to make each aggregate. On the one hand, aggregates should ideally be small. It improves modularity by separating concerns. It is more efficient since aggregates are typically loaded in their entirety. Also, because updates to each aggregate happen sequentially, using fine-grained aggregates will increase the number of simultaneous requests that the application can handle and so improve scalability. It will also improve the user experience since it reduces the likelihood of two users attempting to update the same aggregate. On the other hand, because an aggregate is the scope of a transaction, you might need to define a larger aggregate in order to make a particular update atomic.

For example, earlier I described how in the online store's domain model, Order and Customer are separate aggregates. An alternative design is to make Orders part of the Customer aggregate. A benefit of a larger Customer aggregate is that the application can enforce the credit check atomically. A drawback of this approach is that it combines order and customer management functionality into the same service. It also reduces scalability since transactions that update different orders for the same customer would be serialized. Similarly, two users might conflict if they attempted to edit different orders for the same customer. Also, as the number of orders grows it will become increasingly expensive to load a Customer aggregate. Because of these issues, it is best to make aggregates as fine-grained as possible.

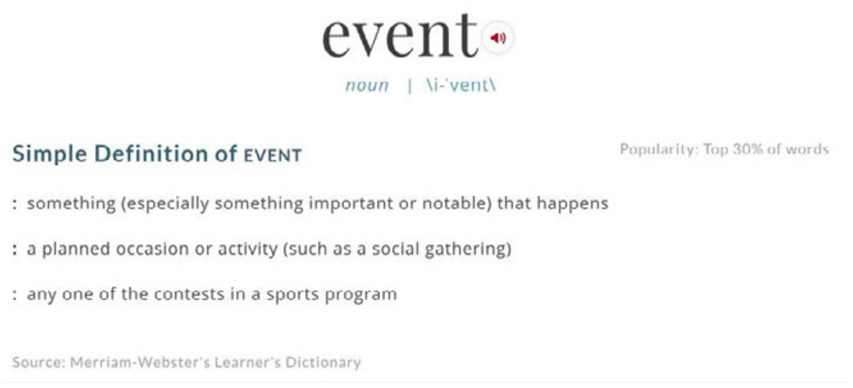
Even though a transaction can only create or update a single aggregate, applications must still maintain consistency between aggregates. The Order Service must, for example, verify that a new Order aggregate will not exceed the Customer aggregate's credit limit. There are a couple of different ways to maintain consistency. One option is to cheat and create and/or update multiple aggregates in a single transaction. This is only possible if all aggregates are owned by the same service and persisted in same RDBMS. The other, more correct option is to maintain consistency between aggregates using an eventually consistent, event-driven approach.

Using Events to Maintain Data Consistency

In a modern application, there are various constraints on transactions that make it challenging to maintain data consistency across services. Each service has its own private data, yet 2PC is not a viable option. Moreover, many applications use NoSQL databases, which don't support local ACID transactions, let alone distributed transactions. Consequently, a modern application must use an event-driven, eventually consistent transaction model.

What is an Event?

According to Merriam-Webster an event is something that happens:



event

noun | \i-ˈvent\

Simple Definition of EVENT

Popularity: Top 30% of words

: something (especially something important or notable) that happens

: a planned occasion or activity (such as a social gathering)

: any one of the contests in a sports program

Source: Merriam-Webster's Learner's Dictionary

In this article, we define a domain event as something that has happened to an aggregate. An event usually represents a state change. Consider, for example, an Order aggregate in the online store. Its state changing events include Order Created, Order Cancelled, and Order Shipped. Events can represent attempts to violate a business rule such as a Customer's credit limit.

Using an Event-Driven Architecture

Services use events to maintain consistency between aggregates as follows: an aggregate publishes an event whenever something notable happens, such as its state changing or there is an attempted violation of a business rule. Other aggregates subscribe to events and respond by updating their own state.

The online store verifies the customer's credit limit when creating an order using a sequence of steps:

1. An Order aggregate, which is created with a NEW status, publishes an OrderCreated event
2. The Customer aggregate consumes the OrderCreated event, reserves credit for the order and publishes an CreditReserved event
3. The Order aggregate consumes the CreditReserved event, and changes its status to APPROVED

If the credit check fails due to insufficient funds, the Customer aggregate publishes a CreditLimitExceeded event. This event does not correspond to a state change but instead represents a failed attempt to violate a business rule. The Order aggregate consumes this event and changes its state to CANCELLED.

Microservice Architecture as a Web of Event-Driven Aggregates

In this architecture, each service's business logic consists of one or more aggregates. Each transaction performed by a service updates or creates a single aggregate. The services maintain data consistency between aggregates by using events.

A distinctive benefit of this approach is that the aggregates are loosely coupled building blocks. They can be deployed as a monolith or as a set of services. At the start of a project you could use a monolithic architecture. Later, as the size of the application and the development team grows, you can then easily switch to a microservices architecture.

Reliably Updating State and Publishing Events

On the surface, using events to maintain consistency between aggregates seems quite straightforward. When a service creates or updates an aggregate in the database it simply publishes an event. But there is a problem: updating the database and publishing an event must be done atomically. Otherwise, if, for example, a service crashed after updating the database but before publishing an event, then the system would remain in an inconsistent state. The traditional solution is a distributed transaction involving the database and the message broker. But, for the reasons described earlier, 2PC is not a viable option.

There are a few different ways to solve this problem without using 2PC. One solution, which is shown in figure 6, is for the application to perform the up- ▶

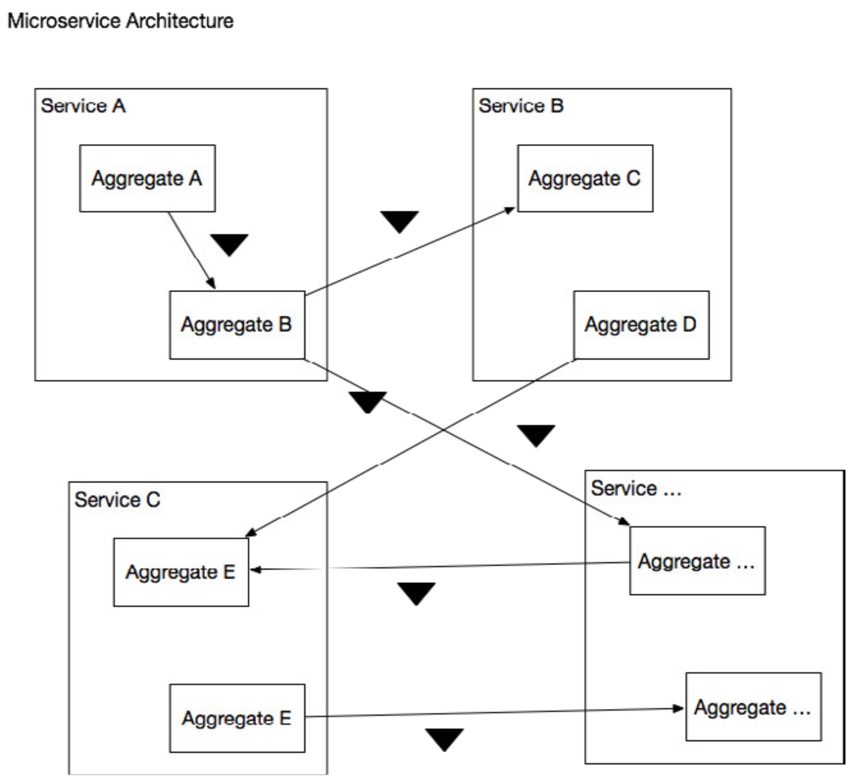


Figure 5 - Events connecting aggregates

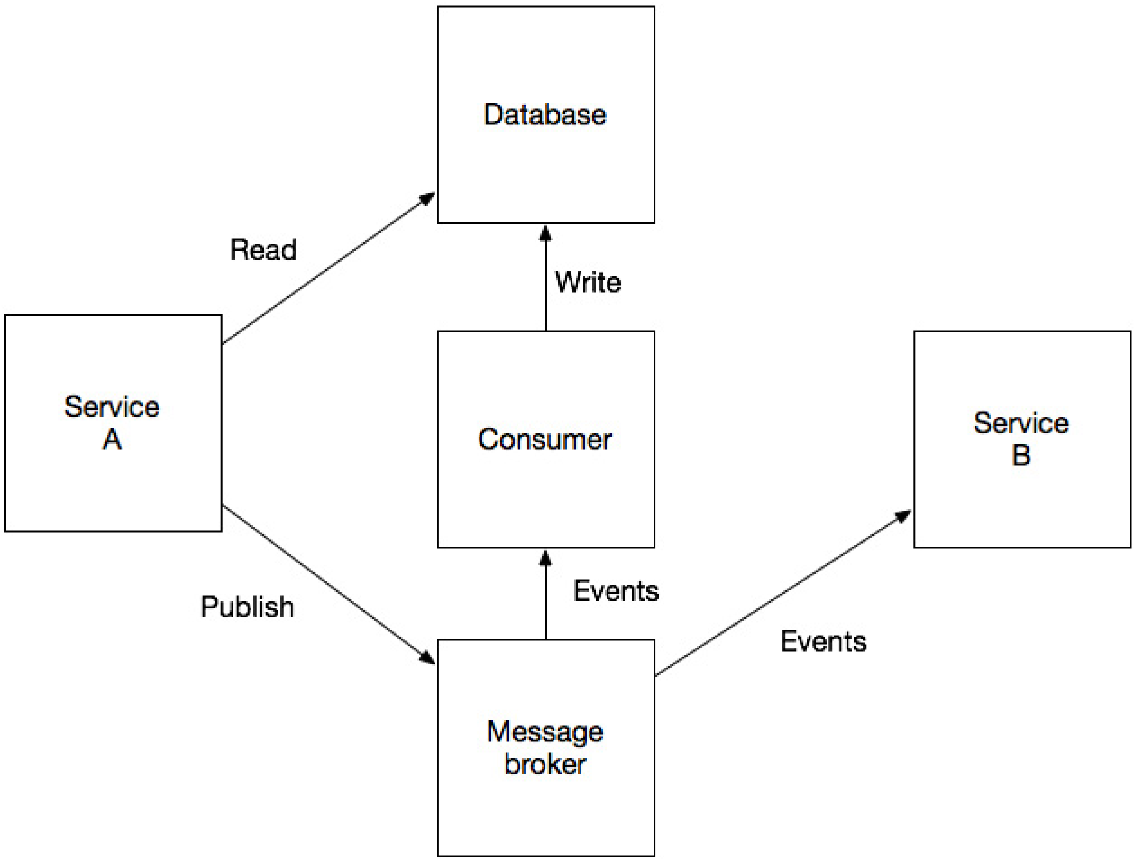


Figure 6 - Updating the database by publishing to a message broker

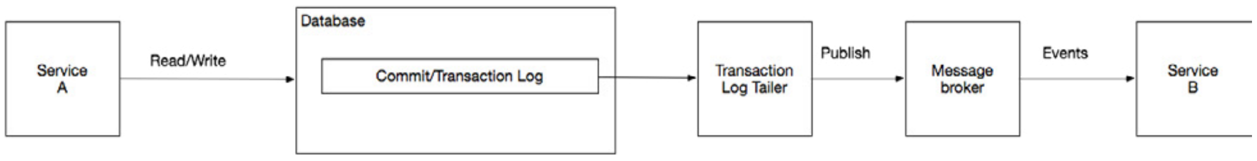
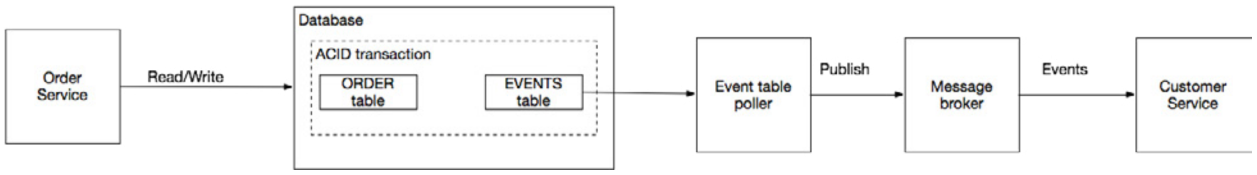


Figure 7 - Tailing the database transaction log

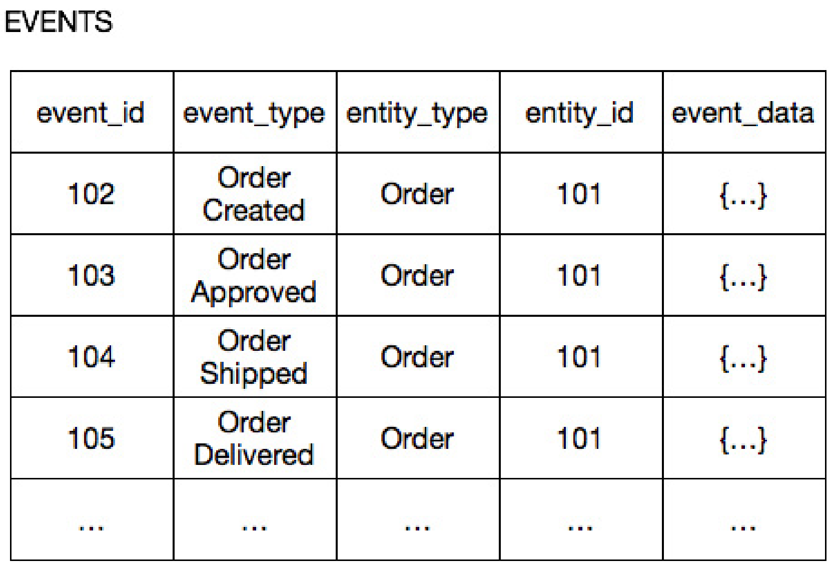


date by publishing an event to a message broker such as Apache Kafka. A message consumer that subscribes to message broker eventually updates the database. This approach guarantees that the database is updated and the event is published. The drawback is that it implements a much more complex consistency model. An application cannot immediately read its own writes.

Another option, which is shown in figure 7, is for the application to tail the database transaction log (a.k.a. commit log), transform each recorded change into an event, and publish that event to the message broker. An important benefit of this approach is that it doesn't require any changes to the application. One drawback, however, is that it can be difficult to reverse engineer the high-level business event - the reason for the database update - from the low-level changes to the rows in the tables.

The third solution, which is shown in figure 8, is to use a database table as a temporary message queue. When a service updates an aggregate, it inserts an event into an EVENTS database table as part of the local ACID transaction. A separate process polls the EVENTS table and publishes events to the message broker. A nice feature of this solution is that the service is able to publish high-level business events. The downside is that it is potentially error-prone since the event publishing code must be synchronized with the business logic.

All three options have significant drawbacks. Publishing to a message broker and updating later doesn't provide read-your-writes consistency. Tailing the transaction log provides consistent reads but can't always publish high-level business events. Using a database table as a message queue provides consistent reads and publishes high-level business events, but it relies on the developer remembering to publish an event when state changes. Fortunately, there is another option. It is an event-centric approach to persistence and business logic known as event sourcing.



EVENTS

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...	...	...	...	...

Figure 9 - Persisting an Order using event sourcing

Developing Microservices with Event Sourcing

Event sourcing is an event-centric approach to persistence. It is not a new idea. I first learned about event sourcing 5+ years ago, but it remained a curiosity until I started developing microservices. That is because, as you will see, event sourcing is a great way to implement an event-driven microservices architecture.

A service that uses event sourcing persists each aggregate as a sequence of events. When it creates or updates an aggregate, the service saves one or more events in the database, which is also known as the event store. It reconstructs the current state of an aggregate by loading the events and replaying them. In functional programming terms, a service reconstructs the state of an aggregate by performing a functional fold/reduce over the events. Because the events are the state, you no longer have the problem of atomically updating state and publishing events.

Consider, for example, the Order Service. Rather than store each Order as a row in an ORDERS table, it persists each Order aggregate as a sequence of events Order Created, Order Approved, Order Shipped, etc.. Figure 9 shows how these events might be stored in an SQL-based event store.

The purpose of each column is as follows:

- `entity_type` and `entity_id` columns - identify the aggregate
- `event_id` - identify the event
- `event_type` - the type of the event
- `event_data` - the serialized JSON representation of the event's attributes

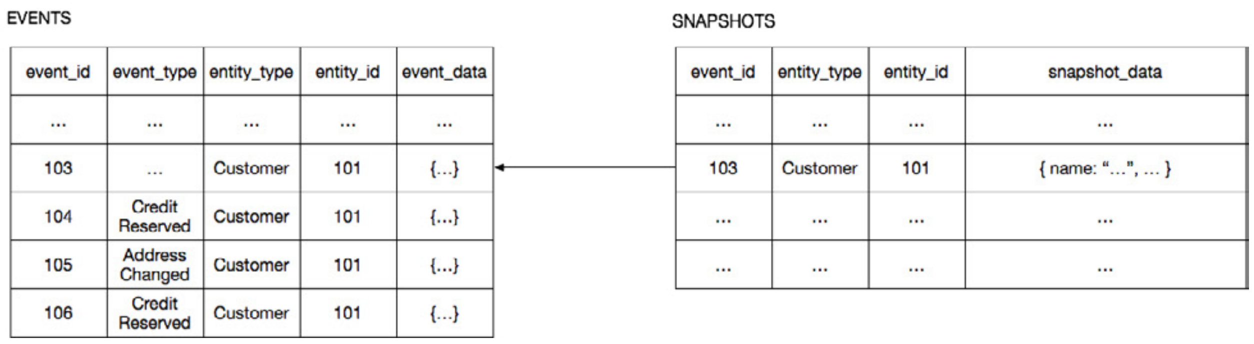
Some events contain a lot of data. The Order Created event, for example, contains the complete order including its line items, payment information and delivery information. Other events, such as the Order Shipped event, contain little or no data and just represent the state transition.

Event Sourcing and Publishing Events

Strictly speaking, event sourcing simply persists aggregates as events. It is straightforward, however, to also use it as a reliable event publishing mechanism. Saving an event is an inherently atomic operation that guarantees that the event store will deliver the event to services that are interested. If, for example, events are stored in the EVENTS table shown above, subscribers can simply poll the table for new events. More sophisticated event stores will use a different approach that has similar guarantees but is more performant and scalable. For example, Eventuate Local uses transaction log tailing. It reads events inserted into the EVENTS table from the MySQL replication stream and publishes them to Apache Kafka.

Using Snapshots to Improve Performance

An Order aggregate has relatively few state transitions and so it only has a small number of events. It is efficient to query the event store for those events and reconstruct an Order aggre- ▶



EVENTS

event_id	event_type	entity_type	entity_id	event_data
...	...	...	...	...
103	...	Customer	101	{...}
104	Credit Reserved	Customer	101	{...}
105	Address Changed	Customer	101	{...}
106	Credit Reserved	Customer	101	{...}

SNAPSHOTS

event_id	entity_type	entity_id	snapshot_data
...	...	...	...
103	Customer	101	{ name: "...", ... }
...	...	...	...
...	...	...	...

Figure 10 - Using snapshots to optimize performance

gate. Some aggregates, however, have a large number of events. For example, a Customer aggregate could potentially have a lot of Credit Reserved events. Over time, it would become increasingly inefficient to load and fold those events.

A common solution is to periodically persist a snapshot of the aggregate's state. The application restores the state of an aggregate by loading the most recent snapshot and only those events that have occurred since the snapshot was created. In functional terms, the snapshot is the initial value of the fold. If an aggregate has a simple, easily serializable structure then the snapshot can simply be, for example, its JSON serialization. More complex aggregates can be snapshotted by using the Memento pattern.

The Customer aggregate in the online store example has a very simple structure : the customer's information, their credit limit and their credit reservations. A snapshot of a Customer is simply the JSON serialization of its state. Figure 10 shows how to recreate a Customer from a snapshot corresponding to the state of a Customer as of event #103. The Customer Service just needs to load the snapshot and the events that have occurred after event #103. The Customer Service recreates the Customer by deserializing the snapshot's JSON and then loading and applying events #104 through #106.

Implementing Event Sourcing

An event store is a hybrid of a database and a message broker. It is a database because it has an API for inserting and retrieving an aggregate's events by primary key. An event store is also a message broker since it has an API for subscribing to events.

There are a few different ways to implement an event store. One option is to write your own event sourcing framework. You can, for example, persist events in an RDBMS. A simple, albeit low performance way to publish events is for subscribers to poll the EVENTS table for events.

Another option is to use a special purpose event store, which typically provides a rich set of features and better performance and scalability. Greg Young, an event sourcing pioneer, has a .NET-based, open-source event store called Event Store. Lightbend, the company formerly known as Typesafe, has a microservices framework called Lagom that is based on event sourcing. My startup, Eventuate, has an event sourcing framework for microservices that is available as a cloud service and a Kafka/RDBMS-based open-source project.

Benefits and Drawbacks of Event Sourcing

Event sourcing has both benefits and drawbacks. A major benefit of event sourcing is that it reliably publishes events whenever the state of an aggregate changes. It is a good foundation for an event-driven microservices architecture. Also, because each event can record the identity of the user who made the change, event sourcing provides an audit log that is guaranteed to be accurate. The stream of events can be used for a variety of other purposes including sending notifications to users, and application integration.

Another benefit of event sourcing is that it stores the entire history of each aggregate. You can easily implement temporal queries that retrieve the past state of an aggregate. To determine the state of an aggregate at a given point in time you simply the fold the events that occurred up until that point. It is straightforward to, for example, calculate the available credit of a customer at some point in the past.

Event sourcing also mostly avoids the O/R impedance mismatch problem. That is because it persists events rather than aggregates. Events typically have a simple, easily serializable, structure. A service can snapshot a complex aggregate by serializing a memento of its state. The Memento pattern adds a level of indirection between an aggregate and its serialized representation.

Event sourcing is, of course, not a silver bullet and it has some drawbacks. It is a different and unfamiliar programming model so there is a learning curve. In order for an existing application to use event sourcing, you must rewrite its business logic. Fortunately, this is a fairly mechanical transformation, which can be done when you migrate your application to microservices.

Another drawback of event sourcing it that message brokers usually guarantee at-least one delivery. Event handlers that are not idempotent must detect and discard duplicate events. The event sourcing framework can help by assigning each event a monotonically increasing id. An event handler can then detect duplicate events by tracking of highest seen event ids.

Another challenge with event sourcing is that the schema of events (and snapshots!) will evolve over time. Since events are stored forever, a service might need to fold events corresponding to multiple schema versions when it reconstructs an aggregate. One way to simplify a service is for the event sourcing framework to transform all events to the latest version of the schema when it loads them from the event store. As a result, a service only needs to fold the latest version of the events.

Another drawback of event sourcing is that querying the event store can be challenging. Let's imagine, for example, that you need to find credit worthy customers who have a low credit limit. You cannot simply write SELECT * FROM CUSTOMER WHERE CREDIT_LIMIT < ? AND c.CREATION_DATE > ?. There isn't a column containing the credit limit. Instead, you must use a more complex and potentially inefficient query that has a nested SELECT to compute the credit limit by folding events that set the initial credit and adjust it. To make matters worse, a NoSQL-based event store will typically only support primary key-based lookup. Consequently, you must implement queries using an approach called Command Query Responsibility Segregation (CQRS).

Implementing Queries Using CQRS

Event sourcing is a major obstacle to implementing efficient queries in a microservice architecture. It isn't the only problem, however. Consider, for example, a SQL query that finds new customers that have placed high value orders.

```
SELECT *
FROM CUSTOMER c, ORDER o
WHERE
   c.id = o.ID
     AND o.ORDER_TOTAL >
100000
     AND o.STATE =
'SHIPPED'
     AND c.CREATION_DATE
> ?
```

In a microservices architecture you cannot join the CUSTOMER and ORDER tables. Each table is owned by a different service and is only accessible via that service's API. You can't write traditional queries that join tables owned by multiple services. Event sourcing makes matters worse preventing you from writing simple, ▶

A distinctive benefit of this approach is that the aggregates are loosely coupled building blocks. They can be deployed as a monolith or as a set of services.

straightforward queries. Let's look at a way to implement queries in a microservice architecture.

Using CQRS

A good way to implement queries is to use an architectural pattern known as Command Query Responsibility Segregation (CQRS). CQRS, as the name suggests, splits the application into two parts. The first part is the command-side, which handles commands (e.g. HTTP POSTs, PUTs, and DELETEs) to create, update and delete aggregates. These aggregates are, of course, implemented using event sourcing. The second part of the application is the query side, which handles queries (e.g. HTTP GETs) by querying one or more materialized views of the aggregates. The query side keeps the views synchronized with the aggregates by subscribing to events published by the command side.

Each query-side view is implemented using whatever kind of database makes sense for the queries that it must support. Depending on the requirements, an application's query side might use one or more of the following databases:

Table 1. Query-side view stores

for example…	**then use….**	**If you need….**
Implement order history by maintaining a MongoDB Document for each customer that contains their orders	a document store such as MongoDB, or a key value store such as Redis.	PK-based lookup of JSON objects
Implement customer view using MongoDB	a document store such as MongoDB	Query-based lookup of JSON objects
Implement text search for orders by maintaining a per-order Elasticsearch document	a text search engine such as Elasticsearch	Text queries
Implement fraud detection by maintaining a graph of customers, orders, and other data	a graph database such as Neo4j	Graph queries
Standard business reports and analytics	an RDBMS	Traditional SQL reporting/BI

In many ways, CQRS is an event-based generalization of the widely used approach of using RDBMS as the system of record and a text search engine, such as Elasticsearch, to handle text queries. CQRS uses a broader range of database types - not just a text search engine. Also, it updates a query-side view in near real-time by subscribing to events.

Figure 11 shows the CQRS pattern applied to the online store example. The Customer Service and the Order Service are command-side services. They provide APIs for creating and updating Customers and orders. The Customer View Service is a query-side service. It provides an API for querying customers.

The Customer View Service subscribes to the Customer and Order events published by the command-side services. It updates a view store that is implemented using MongoDB. The service maintains a MongoDB collection of documents, one per customer. Each document has attributes for the customer details. It also has an attribute that stores the customer's recent orders. This collection supports a variety of queries including those described above.

Benefits and Drawback of CQRS

A major benefit of CQRS is that it makes it possible to implement queries in a microservices architecture, especially one that uses event sourcing. It enables an application to efficiently support a diverse set of queries. Another benefit is that the separation of concerns often simplifies the

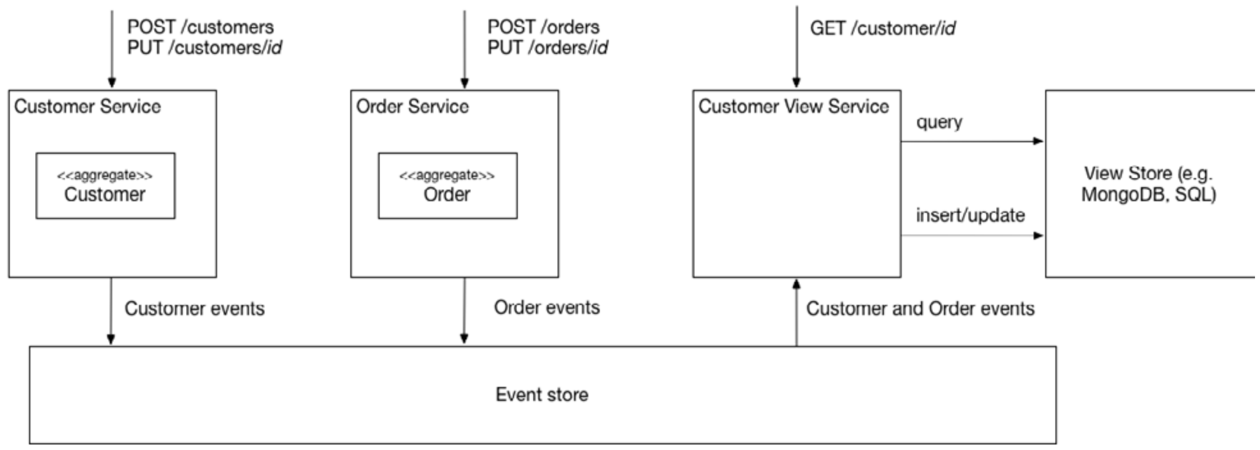


Figure 11 - Using CQRS in the online store

command and query sides of the application.

CQRS also has some drawbacks. One drawback is that it requires extra effort to develop and operate the system. You must develop and deploy the query-side services that update and query the views. You also need to deploy the view stores.

Another drawback of CQRS is dealing with the "lag" between the command side and the query side views. As you would expect, there is delay when the query side is updated to reflect an update to a command-side aggregate. A client application that updates an aggregate and then immediately queries a view might see the previous version of the aggregate. It must often be written in a way that avoids exposing these potential inconsistencies to the user.

Summary

The microservice architecture functionally decomposes an application into services, each of which corresponds to a business capability. A key challenge when developing microservice-based business applications is that transactions, domain models, and queries resist decomposition. You can decompose a domain model by applying the idea of a Domain Driven Design aggregate. Each service's business logic is a domain model consisting of one or more DDD aggregates.

Within each service, a transaction creates or updates a single aggregate. Because 2PC is not a viable technology for modern applications, events are used to maintain consistency between aggregates (and services), following the event sourcing pattern.

Another challenge in the microservice architecture is implementing queries. Queries often need to join data that is owned by multiple services. However, joins are no longer straightforward since data is private to each service. Using event sourcing also makes it even more difficult to efficiently implement queries since the current state is not explicitly stored. The solution is to use Command Query Responsibility Segregation (CQRS) and maintain one or more materialized views of the aggregates that can be easily queried. ■

The first part of CQRS is the command-side to create, update and delete aggregates. The second part handles queries by querying one or more materialized views of the aggregates.

In Defense of the Monolith

Dan Haywood is an independent consultant most known for his work on domain-driven design and the naked objects pattern. He is a committer for Apache Isis, a Java framework for building backend line-of-business applications, which implements the naked objects pattern. Haywood has a 13+ year ongoing involvement as technical advisor for the Irish Government's strategic Naked Objects system on .NET, now used to administer the majority of the department's social welfare benefits. He also has five years ongoing involvement with Eurocommercial Properties co-developing Estatio, an open source estate management application, implemented on Apache Isis. You can follow Haywood on Twitter and on his Github profile.

Originally posted in two parts on Mar 16, 2017 and Mar 24, 2017

Anyone who's worked in the IT industry for a while will have become used to the hype cycle, where the industry seemingly becomes obsessed with one particular pattern or technology or approach.

And for the last couple of years – as a survey of the most recent articles and presentations on InfoQ will show – microservices as an architecture has garnered the most attention. Meanwhile, the term "monolith" seems to have become a dirty word; an application that's difficult to maintain or scale, a veritable "big ball of mud."

This article is a defence of monoliths. But to be clear, when I talk about monoliths, I don't mean an app consisting of one huge lump of code; instead it's a combination of multiple modules. Some of its modules are third-party open source, others are built internally. This article isn't a defence for any old monolith, it's a defence for the "modular monolith". Modules are important, and we discuss them further shortly.

Of course, any architecture is a trade-off between competing forces, and context is all important. In my own case, the two main monoliths I've been involved with are enterprise web apps, which are accessed in-house. For the last 13 years, I've

KEY TAKEAWAYS

Both monoliths and microservices are viable architectures, though a monolith must be modular to be sustainable. Monoliths probably fit better for complex domains (enterprise apps), while microservices are more suited to internet-scale applications with simpler business domains.

Going with a microservices architecture means foregoing both transactions and referential integrity across modules/services. This necessarily comes with an implementation cost.

Both architectures require a platform to support it. For microservices, much of the support relates to the complications that a network introduces (for example circuit breakers). For monoliths, the platform needs to handle cross-cutting technical concerns to allow the developer to focus on the complexity of the domain.

Going "Monolith First" (building the application as a modular monolith initially with the aim of breaking it into microservices later) requires that the modules' boundaries, interfaces and responsibilities be well-defined.

Modules within monoliths (like microservices) should handle their own data, but a naïve mapping of modules to an RDBMS will result in a database that's hard to maintain. A number of patterns can help keep things under control.

worked on a large government benefits administration application running on .NET, and for the last five years I've also worked on an invoicing system running on Java. Both systems are monoliths in the sense that most of the business logic is in a single deployable webapp. I'm sure that many other visitors to the InfoQ website work on similar systems.

I begin this article by exploring some of the key differences between the microservices and monolith approaches; there are pros and cons to both. I then elaborate on some important implementation patterns for modular monoliths and look at the implementation of the Java monolith I work on (its code is available on github).

We start off with a discussion on maintainability (by which you'll see I actually mean modularity).

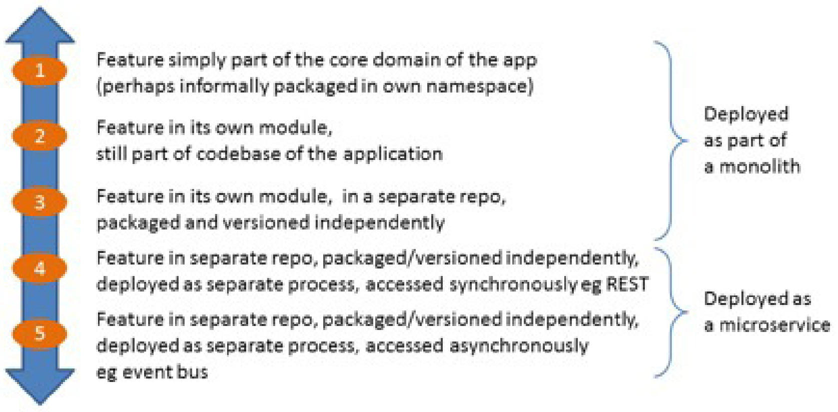
Maintainability (& Modularity)

Whatever its architecture, any non-trivial system represents a substantial investment by the business; the systems I work on are strategic to their respective businesses, and are expected to have a lifetime measured in decades. It's therefore imperative that they be maintainable, that they remain malleable to change. The way to achieve this is through modularity.

Exactly how a module is represented depends on the technology. The source code for a module should be separated out in some way from the rest of the code of the app, and when compiled it may be packaged with additional metadata. A module also has well-defined dependencies, with well-defined interfaces: APIs and possibly SPIs. On the Java system I work on, the modules are JARs built by Maven modules, while on the .NET system they are either C# projects (building a single DLL) or structured as NuGet packages.

Why do modules matter? Ultimately, it's about ensuring that the code is understandable, encapsulating functionality and constraining how different parts of the system interact. If any object can interact with any other object, then it's just too difficult for a developer to fully anticipate all side-effects when code ▶

is changed. Instead, we want to break the app into modules small enough that a developer can understand each module's scope and can reason about its function and responsibility. Moreover, modules must have a stable interface to other modules (even if their implementation changes behind that interface); this will allow those modules to evolve independently of one another. Proper separation of concerns keeps code maintainable over the long term.



1
Feature simply part of the core domain of the app (perhaps informally packaged in own namespace)
2
Feature in its own module, still part of codebase of the application
3
Feature in its own module, in a separate repo, packaged and versioned independently
4
Feature in separate repo, packaged/versioned independently, deployed as separate process, accessed synchronously eg REST
5
Feature in separate repo, packaged/versioned independently, deployed as separate process, accessed asynchronously eg event bus
Deployed as part of a monolith
Deployed as a microservice

Figure 1: Feature packaging/deployment options, monolith-vs-microservices

In breaking up the application into modules, we should also ensure that the dependencies between modules are in one direction only: the acyclic dependencies principle. We'll talk shortly about how to enforce such constraints; whatever the tooling used to enforce these rules, it must be run as part of a CI build pipeline so that any commits that would violate the dependency constraints are rejected.

We should also group code by module so that less stable code depends upon more stable code: the stable dependencies principle. In any given module, all of the classes in that module should have the same rate of change as the other classes in that module. Each module should also only have a single *reason to change*: the single responsibility principle. If we follow these principles, then the module should end up encapsulating some coherent (nameable) business or technical responsibility. And as developers we will know which module holds the code when we need to change it.

It isn't necessary that the source code of all the modules that make up the application be in a single source code repository; after all, the source code for third party open source modules aren't in your repo. On the other hand, it's also not a good idea to create a separate source code repo for every single module, at least, not in the beginning. Chances are that your idea of the responsibilities of a module will change quite a bit, especially in a complex domain. Moving code out too early on is likely to backfire.

So, when should source code for a module move out to its own repo? One reason is when you think you might want to reuse that module within some other application; such a module then has its own release lifecycle and is versioned independently of any application that might be consuming it. Another reason is traceability, so you can easily identify which parts of your monolith have changed (from release to release). Then, any manual user acceptance testing can focus just on the stuff that's changed. A further more pragmatic reason is to reduce contention on the HEAD of a repo, when too many pushes mean that the CI pipeline can't keep pace. If the codebase can't be built and tested in a reasonable timeframe, then enforcing architectural constraints in CI become impossible, and architectural integrity cannot be ensured.

Technical modules are good candidates for moving out into separate repos, for example auditing, authentication/authorization, email, document (PDF) rendering, scanning and so on. You might also have some standardized business sub-domains, such as notes/comments, communication channels, documents, aliases or communications. Figure 1 shows that how we modularize/deploy functionality makes for a spectrum of choices. We can start off with a feature implemented as part of the core domain (option 1), and then gradually modularize (options 2 and 3) as the responsibilities become clearer. Eventually, we can move out the functionality into its own service, deployed as a separate process (options 4 and 5), the difference being whether interactions between the services are synchronous or asynchronous. If this is done for every feature in the application, we have a "pure" microservices architecture.

A key differentiator between monoliths and microservices is therefore that monoliths are more tolerant to changes of modules' responsibilities than a microservices architecture would be:

- If the domain is complex (where a domain-driven design approach makes sense) then you shouldn't try to fix the boundaries around your modules too early; its responsibilities won't be well enough defined. If you are premature then you'll miss the opportunity to have those "knowledge-crunching" insights that are so important to being able to build a rich and powerful domain. Or, if you do have those insights, with a microservices architecture it may be just too expensive/time-consuming to refactor.

- On the other hand, if your domain is well understood, then you can more easily anticipate where those module/service boundaries should be. In such a situation, a microservices architecture is probably viable from the outset.

But opinions on this differ. For example, Martin Fowler's "Monolith First" article is generally in favour of the above approach, but links to some of his colleagues who take an opposing view.

(A)cyclic Dependencies

Building a modular monolith means deciding on how to represent the boundaries of the modules; it means deciding on the direction of the (acyclic) dependencies, and it means deciding on how to enforce those dependency constraints.

Tools such as Structure101 can help with this, allowing you to both map packages/namespaces in your existing codebase to "logical" modules, and optionally enforcing these rules within the CI pipeline. Thus, you can change your module boundaries without moving code about, just by changing the mappings. On the other hand, the boundaries between the modules are not necessarily obvious unless the codebase is looked at through the Structure101 lens, and a developer may not realize that they have broken a dependency constraint until they commit their code causing the CI build to fail.

A more direct approach, requiring no special tooling, is to move code around, for example (on the JVM) creating separate Maven modules (option 2 in figure 1). Then, Maven itself can enforce dependencies, both prior to and within the CI pipeline. In .NET, this option likely means separate C# projects (rather than namespaces within a single C# project), referenced directly rather than wrapped up as NuGet packages.

You may also need to write custom checks to enforce these architectural dependencies. For example, in the .NET application I work on, each module consists of an Api C# project and an Impl C# project. We fail the build if this naming convention isn't followed. We also require that Impl projects only reference other Api projects; we fail the build if an Impl project references another Impl project directly.

So, moving to option 2 is a good pragmatic first step, but you may decide to go further by moving those modules out into their own separate codebases (option 3). However, care is needed. Because each module is built independently, it's possible to end up with cyclic dependencies.

For example, a customers v1.0 module might depend upon addresses v1.0 module, so customers is in a higher "layer" than addresses. However, if a developer creates a new version addresses v1.1 that references customers v1.0, then the layering Is broken, and we seemingly have the cus-

Building a modular monolith means deciding on how to represent the boundaries of the modules tomers and addresses modules mutually dependent upon each other; a cyclic dependency.

Microservice architectures have their own version of this problem. If customers and addresses are microservices, then the exact same scenario can play out, also resulting in a cyclic dependency. It now becomes rather difficult to update either service independently of the other. Net result: the worst of all worlds, a distributed monolith.

At least for monoliths, build tools such as Maven can be used to help flag such issues, which we'll look at this in more detail later. If going with a microservices architecture then you'll have to do more work (with fewer tools to help you) if you are going to even identify the problem, let alone solve it.

Mostly what this tells us is that you shouldn't rush to move to option 3 (separate codebases for modules) for a monolith, and any modules that you do pull out should already have stable interfaces. A microservices architecture, on the other hand, forces every microservice to be independent and in its own codebase. Much more care needs to be taken to get the responsibilities, interfaces and dependencies right early on. That's difficult to do if you don't know the business domain well.

Data

In a microservices architecture, it's generally accepted that each service is responsible for its own data. One of the oft--cited benefits of microservices is that each module can choose the most appropriate persistence technology: RDBMS, NoSQL, key store, event store, text search, and so on.

If a service needs information that is "owned" by some other service, then either (a) the consuming service will need to ask the other service for the data, or alternatively (b) the data will need to be replicated between the owning and the consuming service. Both have drawbacks. In the former, there is temporal coupling between the services (the owning service needs to be "up"), while the latter takes significant effort and infrastructure to implement correctly. One option that should never be contemplated though: services should never share a common database. That's not a microservices architecture, it's another way to accidentally end up with a distributed monolith.

In a modular monolith, each module should also take responsibility for its own persistent data, and of course each module could also use a different persistence technology, if it so wished. On the other hand, many modules will likely use the same persistence technology to store their entities; relational databases still (rightly) rule the roost for many enterprise systems. If a module needs information that is "owned" by some other module, it can just call that module's API; no need to replicate data or to worry if that module is "up".

With multiple modules using the same persistence technology, this offers a "tactical" opportunity to co-locate those tables on a single RDBMS. Don't assume that an RDBMS won't scale well enough for your domain; context is everything, and RDBMS are far more scalable than some might have us believe (we'll revisit the topic of scalability shortly).

The benefits of co-locating data of modules are many. It means we can support business intelligence/reporting requirements (requiring data from multiple modules) simply by using a regular SELECT with joins (probably deployed as a view or stored procedure). It also simplifies the implementation of batch processing, where for efficiency's sake the business functionality itself is deliberately co-located with the data (e.g. as stored procedures). Co-locating data is also going to simplify some operational tasks such as database backups and database integrity checks.

All of these things are more complicated with a microservices architecture. For example, business intelligence/reporting with microservices in effect requires a "client-side" join, with information between services exchanged through some event bus and then merged and persisted as some sort of materialized view. It's all doable, of course, but it's also a lot more work than a simple view or stored proc.

That said, it is possible – in fact, rather easy – when co-locating modules' data to accidentally create a "big ball of mud" in an RDBMS. If we're not careful we can have foreign keys all over the place (structural coupling) and we also run the risk of developers writing a SELECT in one module that queries data directly from another module (behavioural coupling). Next, we'll take a more detailed look at how to address these issues.

There's another major benefit when different modules' data is co-located, and that's to do with transactions. We explore that next.

Transactionality (& Synchronicity)

It's common for a business operation to result in a change of state in two or more modules. For example, if I order a new TV from an online retailer, then all inventory, order management and shipping will be affected (and probably many more modules besides).

In a microservice architecture, because every service has its own data store; these changes must be made independently, with messages used to ensure that a user doesn't end up being charged for a new TV but never receiving it (or indeed, the opposite, getting a new TV without paying for it). If

something goes wrong, then compensating actions are used to "back out" the change. If the retailer has taken the cash but then cannot ship, it will need to refund the cash in some way.

In some domains – such as online retailing – this asynchronous nature of interactions between various subdomains is commonplace. End-users understand and accept that payment of goods versus their shipment are quite separate and decoupled operations, and that if things do go wrong then partially completed operations will be reversed.

However, consider a different domain, where the end-user of an in-house invoicing application might want to perform an invoice run. This will mostly modify the state within the invoicing module. However, if some customers want their invoices to be sent out by email, then it might as a side-effect create documents and communications in their respective modules. So here we have a business operation that could require a state change in several modules.

In a microservices architecture, the documents and communications would need to be created asynchronously. If the end-user wanted to view those outbound communications, then we would require some sort of notification mechanism for when they are ready to be viewed.

In comparison, in a monolith, if the backing data stores for the invoicing, documents and communications modules are all co-located in the same RDBMS, then we can simply rely on the RDBMS transaction to ensure that all the state is changed atomically. Assuming the actual processing is performant enough, the user can simply wait a couple of seconds for all entities in all modules to be created/updated.

In my mind, this is a better user experience, as well as being a simpler design (so cheaper to support/maintain). If the processing does end up taking longer than a couple of seconds, then we can always refactor to a microservices-style approach and move some of the processing into the background, invoked asynchronously.

Synchronous behaviour can improve the user experience in other ways too. Imagine that each customer has a collection of associated email addresses, and that one of these email addresses is nominated as the one to send invoices to. Suppose now that the end-user wants to delete that particular email address. In this case, we want the invoicing module to veto the deletion, because that email address is "in use". In other words, we want to enforce a referential integrity constraint across modules.

Supporting this use case in a microservice requires a different and more complicated approach. One design is for the customer service to call all the other services that use the data to ask if it can be deleted. But to do that it will need to look those services up somehow and query each in turn; and what should happen if one of them is unavailable? Or, the customer service might just "logically" delete the email address, allowing the invoicing service to resurrect the address later on if necessary: a compensating action, in other words. That might suffice in this case but is potentially confusing. In general, any design based solely on asynchronous communication is liable to result in unpleasant race conditions that need to be thought through carefully.

In contrast, a well-designed monolith can easily handle the requirement. Later, we'll look at some designs to handle this, honouring the fact that modules must be decoupled, but exploiting the fact that interactions between modules are in-process.

Complexity (& Asynchronicity)

In a modular monolith, the modules are co-located in the same process space. Thus, to get one module to interact with another is nothing more elaborate than a method call.

The corresponding interaction in a microservices architecture will, however, involve the network:

- If the services interact synchronously, then chances are you'll use REST, in which case there's a plethora of decisions to make and technicalities to navigate: what data format (XML or JSON probably), whether to encode using HAL, Siren, JSON-LD or perhaps roll-your-own, how to map HTTP methods to business logic, whether to do "proper HATEOAS" or simply RPC over HTTP- the list goes ▶

on. You'll also need to document your REST APIs: Swagger, RAML, API Blueprint, or something else.

- Or perhaps you'll go some other way completely, e.g. using GraphQL.
- Also, any synchronous interaction between services must be tolerant to failure, otherwise (again) the system is just a distributed monolith. This means that each connection needs to anticipate this, with some sort of fallback mechanism if the called service is not available.
- If the services interact asynchronously then there are many of the same sorts of decisions, along with some new ones: data format (XML, JSON, or perhaps protobuffers), how to specify the semantics of each message type; how to let message types evolve/version over time; whether interactions will be one-to-one and/or one-to-many; whether the interactions will be one-way or two-way; should events be somehow choreographed; should perhaps sagas be used to orchestrate the state changes; and so on.
- You will also need to decide over which "bus" the services will interact: AMQP/RabbitMQ, ActiveMQ, NSQ, perhaps use Akka actors, or something else? And in some cases, these buses have only limited bindings to programming languages, thereby constraining the language that services can be written in.

Whichever style of network interaction is used, a microservices architecture will also require support for aggregated logging, monitoring, also service discovery (to abstract out the actual physical endpoints that services talk to), load balancing, and routing. The need for this stuff is not to be underestimated: otherwise, when things go wrong you'll have no way of figuring out how n separate processes interact with each other when the end-user tries to checkout their shopping cart, say.

In other words, with a microservices architecture there's an awful lot of technical plumbing, none of which goes towards solving the actual business use case. Granted, it's probably quite enjoyable plumbing, and there are plenty of open source libraries available to help, but even so- it takes a lot of engineering to make it work, and for many applications it is probably over-engineering.

This isn't to say that a monolith doesn't also require a supporting platform. Given that a monolith's sweet spot is to handle more complex domains, it's important that its platform allows the development team to stay focused on the domain, and to not have to worry too much about cross-cutting technical concerns. Frameworks that remove boilerplate for transactions, security and persistence are mature and commonplace.

And monoliths do have issues of their own. Most seriously, it can be rather easy over time for the separation of responsibilities between the presentation, domain and persistence layers to erode over time: a different way to create a big ball of mud. The hexagonal architecture is a pattern that emphasises that the presentation layer and persistence layer should depend on the domain layer, not the other way around. But patterns aren't always followed and so it's also very common with monoliths for business logic to "leak" into adjacent layers, particularly the UI.

Later, we'll see that frameworks do exist to prevent such leakage of concerns – principally by also treating the UI/presentation layer as just another cross-cutting concern (the naked objects pattern). It also means that the developer – tackling a complex domain – can focus just on the bit of the app that really matters: the core business logic.

Scalability (& Efficiency)

One of the main reasons cited for moving to a microservice architecture is improved scalability.

In any given system (microservices or monolith), certain modules/services are likely to see more traffic than other areas. In a microservices architecture, each service runs as a separate operating system process, so it's true that each of those services can be scaled independently of each other. If the bottleneck is in the invoicing service for example, more instances of that service can be deployed. There are a number of solutions to perform the orchestration/load-balancing of Docker containers (e.g. Kubernetes, Docker Swarm, DC/OS and Apache Mesos), and if not fully mature, yet, they are at least getting there; but you will need to invest time learning them and their quirks.

Scaling a monolith requires deploying multiple instances of the entire monolith application, one result being more memory used overall. Even then that may not necessarily solve the issue. For example, the scalability problem might be locking issues in the database and adding more instances of the monolith might actually make things worse. More subtly, you would also need to check that there are no assumptions in the monolith's codebase, that there would only ever be one instance of the monolith running. If there are, that's also a show-stopper to scalability, and will need fixing.

On the other hand, when it comes to compute and network resources, microservices are less efficient than monoliths; if nothing else there is all the extra work handling all those network interactions (in a monolith, just in-process method calls). And, in fact, a microservice system might end up using more memory too, because each and every one of those fine-grained microservices might require its own JVM or .NET runtime to host it.

There is also the notion with a monolith of putting all the eggs in one basket. For the most critical module/service, the architect will select an appropriate (perhaps expensive) technology stack to obtain the required availability. With a monolith, all the code must be deployed on that stack, possibly raising costs. With microservices, the architect at least has the choice to deploy less critical services on less expensive hardware.

That said, high availability solutions are becoming less expensive thanks to the rise of Docker containers and the orchestration tools mentioned above (Kubernetes, et al). These benefits apply equally to both microservices architectures and monoliths.

Flexibility (of Implementation)

With a monolith, all the modules need to be written in the same language, or at least be able to run on the same platform. But that's not all that limiting.

On the JVM there is large number of languages, in a variety of paradigms: Java, Groovy, Kotlin, Clojure, Scala, Ceylon, and JRuby all have significant communities and are actively developed. It's also possible to build one's own DSLs using Eclipse Xtext or JetBrains MPS.

On the .NET platform, the list of commonly used languages is somewhat smaller, but C# is a great (mostly) object-oriented language, while F# is a superb functional language. Meanwhile JetBrains Nitra targets writing DSLs.

In a microservices architecture, there is of course more flexibility in choosing languages, because each service runs in its own process space so can in theory be written in any language: JVM or .NET, but also Haskell, Go, Rust, Erlang, Elixir or something more esoteric. And because services are intentionally fine-grained, the option exists to re-implement a service in possibly a different language, and throw away the old implementation.

However, is it necessarily wise to have a system implemented in a dozen underlying languages? Perhaps it's justifiable for a small number of services to use one of the more specialized languages if their problem domain fits its paradigm particularly well. But using too many different languages is merely going to make the system more difficult to develop and maintain/support.

In any case, there are likely to be some real-world restrictions. If the services interact synchronously then you will need to ensure that they all play nicely with the circuit breakers and so on, that you provide appropriate resilience; you can use Netflix' open source tools for the JVM, but you might be on your own if using some other platform/language. Or, if the services interact asynchronously, then you'll need to ensure there are appropriate language bindings/adapters for those services to send and receive messages over the event bus.

In practical terms, I suspect that for any given application the number of modules that genuinely become easier to reason about when written in a more "esoteric" programming language will be very few, two or three say. For these, go ahead and write them in that language and then link to them either in-memory (if possible) or over the network (otherwise). For the other modules of the application, implement them in a mainstream JVM or .NET language.

(Developer) Productivity

Software is labour-intensive stuff to produce, so the developers writing it need to be productive. Working with microservices should improve productivity, so the thinking goes, because each part of the system is small and light. But that's too much of a simplification.

For a microservice, a developer can indeed load up the code for a microservice in their IDE quickly, and spin up that microservice and run its tests quite quickly. But the developer will need to write substantially more code to make that microservice interact with any other microservice. And, to run up the entire system of microservices (for integration testing purposes) requires a lot of coordination. Tools such as Docker Compose or Kubernetes start to become essential.

For a (modular) monolith, the developer can also work on a single ▶

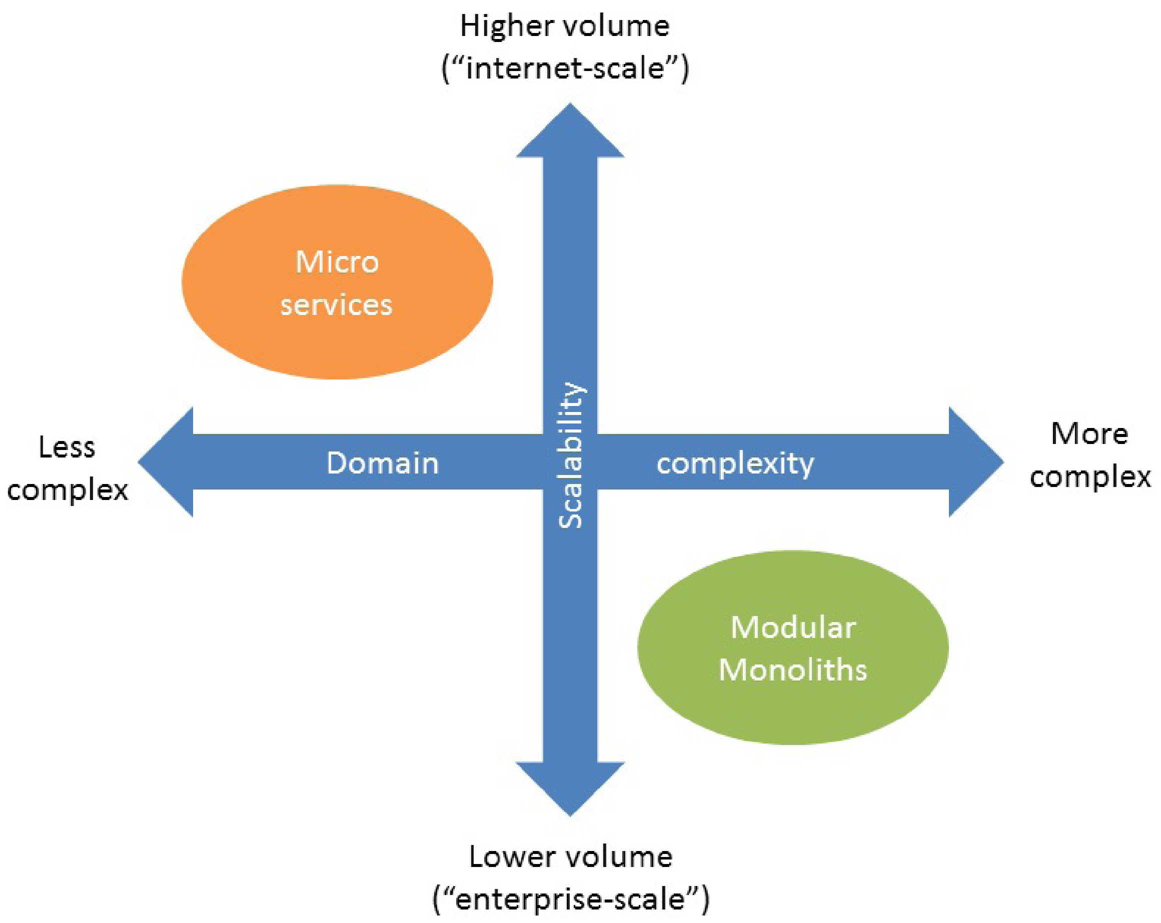


Figure 2: Scalability vs Domain Complexity

module within that monolith. Indeed, if that module has been broken out into its own code repo, then new features can be added and tested entirely separately from the application intended to consume the module. The benefits are similar.

If the module hasn't been broken out into a separate repo, then the monolith's architecture should provide the ability for the application developer to bootstrap only selected subsets of the application required by the feature that they are working on; again, the overall developer experience will be similar to that of working on microservices. On the other hand, if there's no capability to run subsets of the monolith, then this can indeed have a serious impact on productivity. It's not unknown for monoliths to get so large that they take many minutes to restart; a problem that can also affect the time to execute its tests.

Which Architecture Should You Choose?

Thus far, we've been comparing the monolith and microservices architectures, exploring the benefits and weaknesses of both.

In a sense, both a modular monolith and a microservices architecture are similar in that they are both modular at design time. Where they differ is that the former is monolithic at deployment time while microservices take this modularity all the way through to deployment also. And this difference has big implications.

To help decide which architecture to go for, it's worth asking the question: "what is it you are trying to optimise for?" Two of the most important considerations are shown in figure 2.

If your domain is (relatively) simple but you need to achieve "internet-scale" volumes, then a microservices architecture may well suit. But you must be confident enough in the domain to decide up-front the responsibilities and interfaces of each microservice.

If your domain is complex and the expected volumes are bounded (e.g. for use just within an enterprise) then a modular monolith makes more sense. A monolith will let you more easily refactor the responsibilities of the modules as your understanding of the domain deepens over time.

And for the tricky high complexity/high volume quadrant, I would argue that it's wrong to optimize for scalability first. Instead, build a modular monolith to tackle the domain complexity, then refactor to a microservices architecture as and when higher volumes are achieved. This approach also lets you defer the higher implementation costs of a microservices architecture until such time that your volumes (and presumably revenue) justify the business case to spend the extra money. It also lets you adopt a hybrid approach if you wanted: mostly a monolith, with microservices extracted only as and when it makes sense.

If you do want to adopt a "Monolith First" approach, then you should exploit the similarities between the two architectures:

- Both modules in a monolith and microservice are responsible for their own persistent data. The difference is that the co-located modules can also leverage transactions and referential integrity provided by the (probably relational) data store.

- Both monoliths and microservices should interact only through well-defined interfaces. The difference is that with a monolith the interactions are in-process, whereas with microservices they are over the network.

Bear these points in mind and it will be that much easier to convert a modular monolith to an microservices architecture if you find you need to.

Even so, building a modular monolith needs to be tackled thoughtfully. Next, we'll look at some of the implementation patterns for building a modular monolith, and look at a platform and an example monolith that runs on the JVM.

Implementation Concerns

Implementing a microservices architecture correctly can be challenging, but building a modular monolith also needs to be tackled thoughtfully. We've identified a number of potential issues:

- A modular monolith must consist of, well, modules. However, this can result in accidental cyclic dependencies. It can also give rise to JAR hell, which we'll explore next.

- While every module should be responsible for its own data, monoliths can "tactically" exploit the fact that many modules may persist to the same, single, transactional data store. Care is needed though to ensure the resultant database doesn't become a "big ball of mud".

- Guaranteed synchronous calls between modules can provide a better user experience. However, these modules must be decoupled to allow them to evolve independently. Slowly evolving modules should not depend on modules that are often changed.

- In order to allow the development team to stay focused on the domain, a platform/framework is required to handle as many cross-cutting concerns as possible. Even so, it's still rather common for business logic to "leak" from the domain layer into the adjacent presentation or persistence layers.

We're now going to explore how to tackle these issues, and we'll look at an example of a real-world modular monolith on the JVM that leverages a powerful open source framework to manage cross-cutting concerns.

Acyclic Dependencies and JAR hell

With a modular monolith, we need some way to delineate the boundaries of each module.

Our first option is to use language features – such as packages (Java) or namespaces (.NET) – to group together the module's functionality, but it isn't otherwise distinguished from the rest of the application. There are however no guarantees that there won't be cycles between those packages/namespaces; if you only use this option, you're very likely to end up with a non-modular monolith, a big ball of mud.

Instead, we need a bit more structure, allowing build tools to enforce the acyclic dependencies we require between those modules. Implementing this on the Java platform could be done using a Maven multi-module project; for .NET it would be a single Visual Studio solution with multiple C# or F# projects within. All this code is recompiled together, but the build tooling (Maven or Visual Studio) will ensure that there are no cyclic dependencies between those modules.

One downside with this second option is that, because all the code is held in a single code repo and is all (re)compiled together, it also must all be (re)tested and it all gets the same version num- ▶

ber. This option doesn't exploit the fact that, in reality, different modules evolve at different speeds. Why continually rebuild/retest code that changes only slowly over time?

A third option is therefore to move modules out into their own code repos, and version each separately. On the .NET platform, we can package each module up as a NuGet package, while on Java we can package as Maven modules. From the context of the main application that consumes them, these modules are indistinguishable from a third-party dependency.

However, this is also where we need to take care because it's possible to end up with cyclic dependencies. For example, suppose that a customers v1.0 module depends upon an addresses v1.0 module. If a developer creates a new version addresses v1.1 that references customers v1.0, then we seemingly have the customers and addresses modules mutually dependent upon each other; a cyclic dependency. This is, of course, a Bad Thing™.

To solve this, we need to decide which direction the dependencies are meant to flow in: is customers module meant to depend on the addresses, or vice versa? The heuristic here is the stable dependencies principle: unstable (frequently changing) modules should depend on stable (infrequently changing) modules. In our example, the question becomes: which concepts are more volatile: customers or addresses? If the direction of the dependency is incorrect, then the dependency inversion principle can be used to refactor.

Figuring this out can be quite straightforward. Some modules may just hold reference data, for example tax rate tables or currency. Other modules that are almost but not quite reference data include counterparties, and fixed assets, or maybe (financial) instruments. Another good example is "filing-cabinets" which just store stuff, for example, documents or communications. In all these cases, other modules will depend on these modules, not the other way around.

We could also take a more scientific approach and turn to our version control history, measuring the relative amount of churn in each module.

Modules that are stable are good candidates to move out of the application's code repository and into their own repositories. And once you have moved out modules into their own repo, then they can start being reused in other applications too.

Actually, all we require is that the interface defined by a module be stable. Whether or not the implementation behind the interface is stable is unimportant. In fact, it can be a good move to also move modules out whose implementation is still in flux, because it removes some of the code churn from the main repo. Exploiting this fact does though require that the module's interface be formally, and not implicitly, defined.

The above is all well and good, but what we also need is an early warning when a cyclic dependency does accidentally get introduced, ideally within our build or CI. This is achievable.

Let's go back to the example above: customers v1.0 à addresses v1.0 while addresses v1.1 à customers v1.0. The application itself will link to the latest version of each module, which gives us customers v1.0 and addresses v1.1 in a cyclic dependency.

This is a dependency convergence problem, more commonly

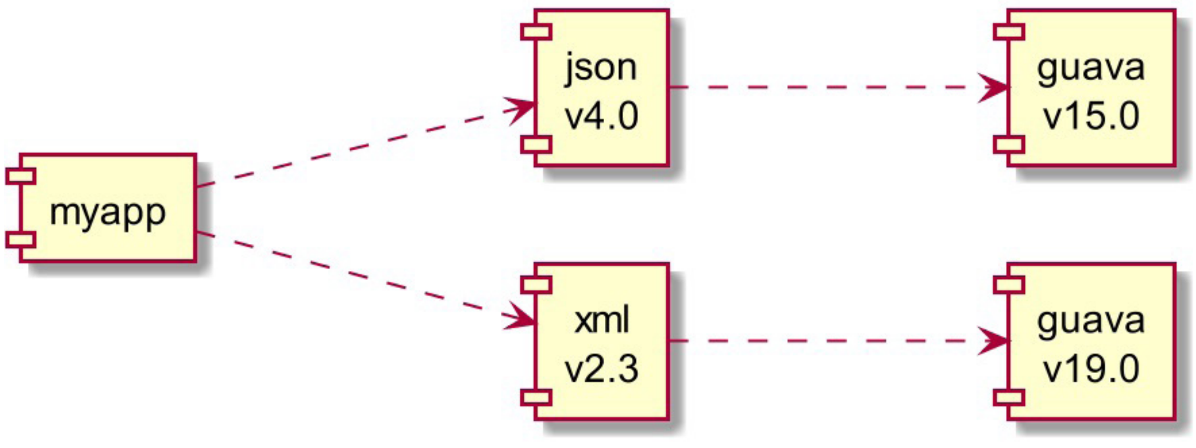


Figure 3: Dependency Convergence Conflicts

called "JAR (or DLL) hell". Figure 3 shows a more common example, where an application uses two libraries that in turn use conflicting versions of some common base library.

If running on the JVM, then this would manifest at runtime with linkage errors; under normal circumstances the JVM only loads one version of a class at a time.

To fix this, Maven's Enforcer plugin can be configured to flag any dependency convergence issues, if necessary failing the build. The developer can then use <dependencyManagement> section within the pom.xml (or sometimes dependency <exclusions>) to decide which version of any given common library to run with. The use of semantic versioning by open source libraries is increasingly common, so if the version difference is only minor (v2.3 vs v2.4) then most likely the higher version can be used without issue.

If using NuGet 3.x, then a similar effect can be achieved by virtue of the "Nearest wins" dependency resolution rule.

That said, some projects, such as Guava, release major versions quite regularly and do delete deprecated API; there's a chance that it might not even be possible to run the monolith shown in figure 3. In such a case, you must look to fix that dependency conflict by updating it. If that's not an option, you might be able to shade (repackage) the dependency. If those aren't options for you, you'll just have to rework your code somehow to remove the conflict or maybe even the dependency.

For the sake of completeness, we should note that OSGi applications (on the JVM) avoid this problem because each module chain (bundle in OSGi parlance) can be arranged to load in a different classloader. However, while OSGi has its fans, it's the exception rather than the rule, and may well lose ground when Java 9 ships with the Jigsaw module loading system. Jigsaw is no silver bullet though;it very deliberately does not attempt to tackle the dependency convergence issue, instead leaving it as a problem for build tools such as Maven to handle.

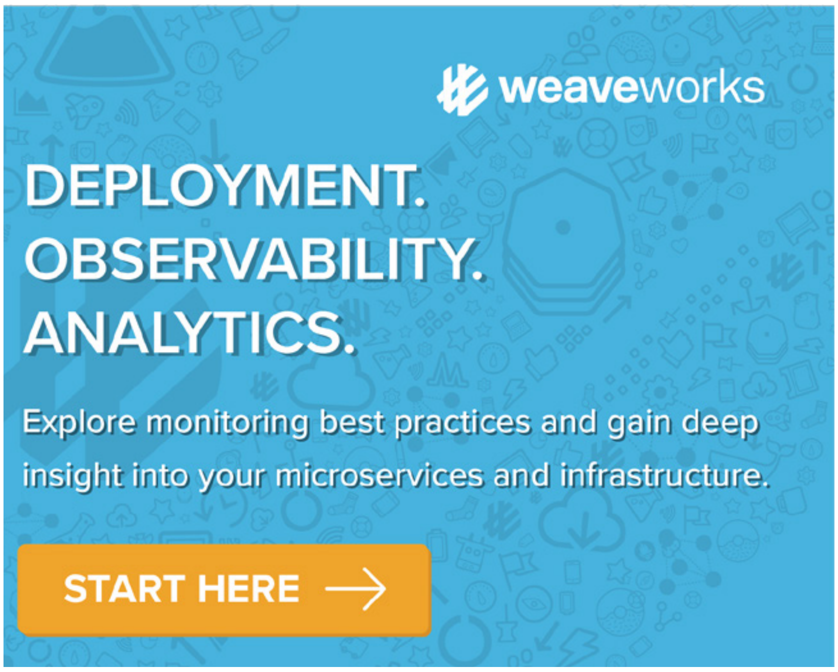
To summarize: (on the JVM at least) use Maven's Enforcer plugin to enforce dependency convergence issues, and if there are conflicts, then clearly handle them with <dependencyManagement> sections and if necessary <exclusions>. Keep these under close review – I've started putting mine into an always-active <profile> called "resolving-conflicts" so they are more obvious – and always be looking to reduce these exceptions over time.

Data

Just as in a microservices architecture, in a modular monolith each module is responsible for persisting its own data. In most cases, these modules will all be using a relational database to store their entities: relational databases still (rightly) rule the roost for many enterprise webapps. This then provides the "tactical" opportunity to co-locate those tables on a single RDBMS, and thus take advantage of transactions.

In terms of mapping entities in a module to an RDBMS, since each module will have its own namespace/package, this should be reflected in terms of the schema names of the tables (to which the entities within those modules are mapped). The module/schema should also be used as the value of any discriminator columns for super-type tables (i.e. mapping inheritance hierarchies).

One of the key differences between a domain object model and a relational database is the


weaveworks
DEPLOYMENT.
OBSERVABILITY.
ANALYTICS.
Explore monitoring best practices and gain deep insight into your microservices and infrastructure.
START HERE

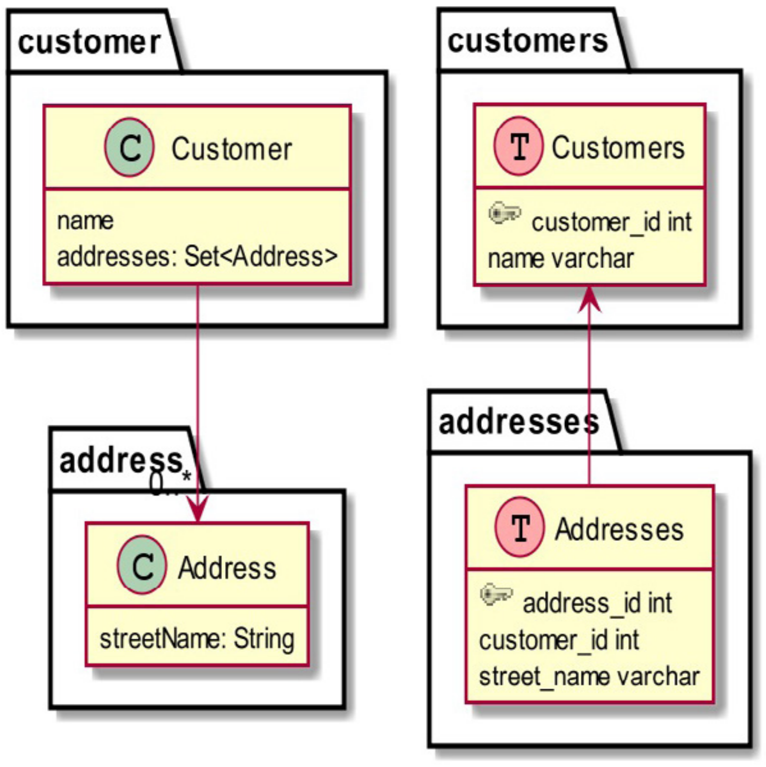


Figure 4: Class vs Table Relationships

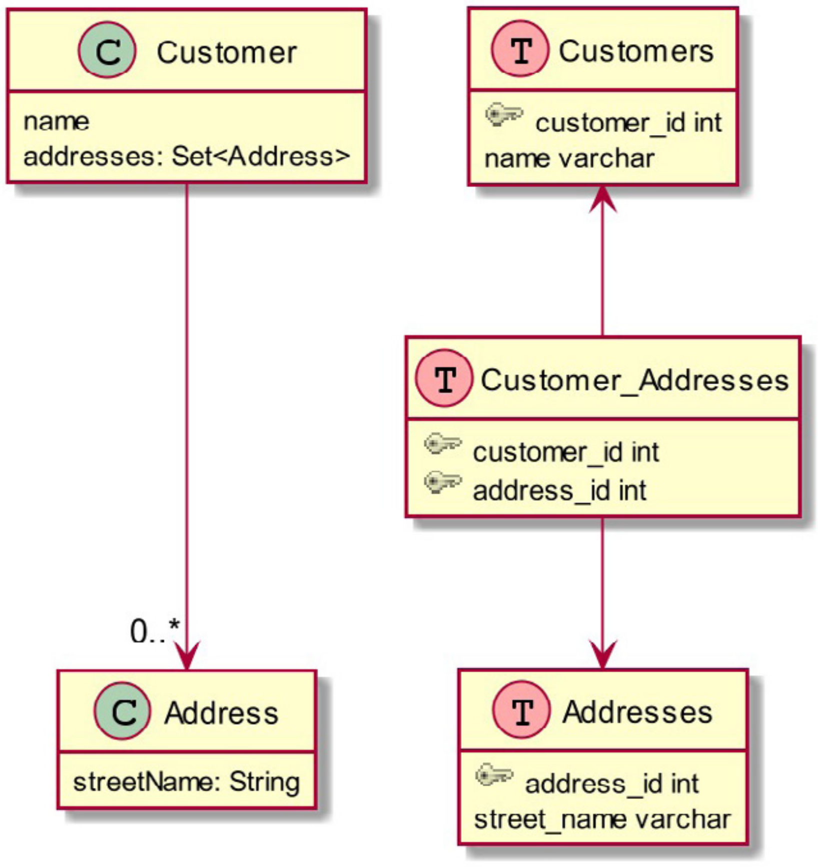


Figure 5: Link table

means by which relationships between entities are represented; in memory, there's an object pointer, whereas in the database there's a foreign key attribute. As figure 4 shows, a naïve mapping of the classes (on the left) to the tables (on the right) can result in the direction of dependencies in effect being the opposite in the database to that of the code.

The places that hold the Customer entity are both the Customers table, and also the Addresses. customer_id column (because that foreign key corresponds to the Customer.addresses field). Even if the codebase is nicely organized as a set of layered modules with acyclic dependencies, when we look at the RDBMS we have our big ball of mud.

The problem can be fixed though. To keep all the Customer information in the same schema, we should move the foreign key out of the Addresses table and into a link table, as shown in figure 5. The performance hit will be negligible.

I would argue that relationships for the tables of entities within the same module don't need this treatment... but I also wouldn't argue too hard against you if you wanted to always introduce a link table for all associations.

More involved are polymorphic associations between objects. For example, we might want to be able to attach Documents to all domain objects. As shown in figure 5, we can introduce the concept of Paperclip (an interface) and use concrete implementations to act as the link table.

Each individual Paperclip will be mapped to two tables, one in the documents schema, and one in the schema specific to its

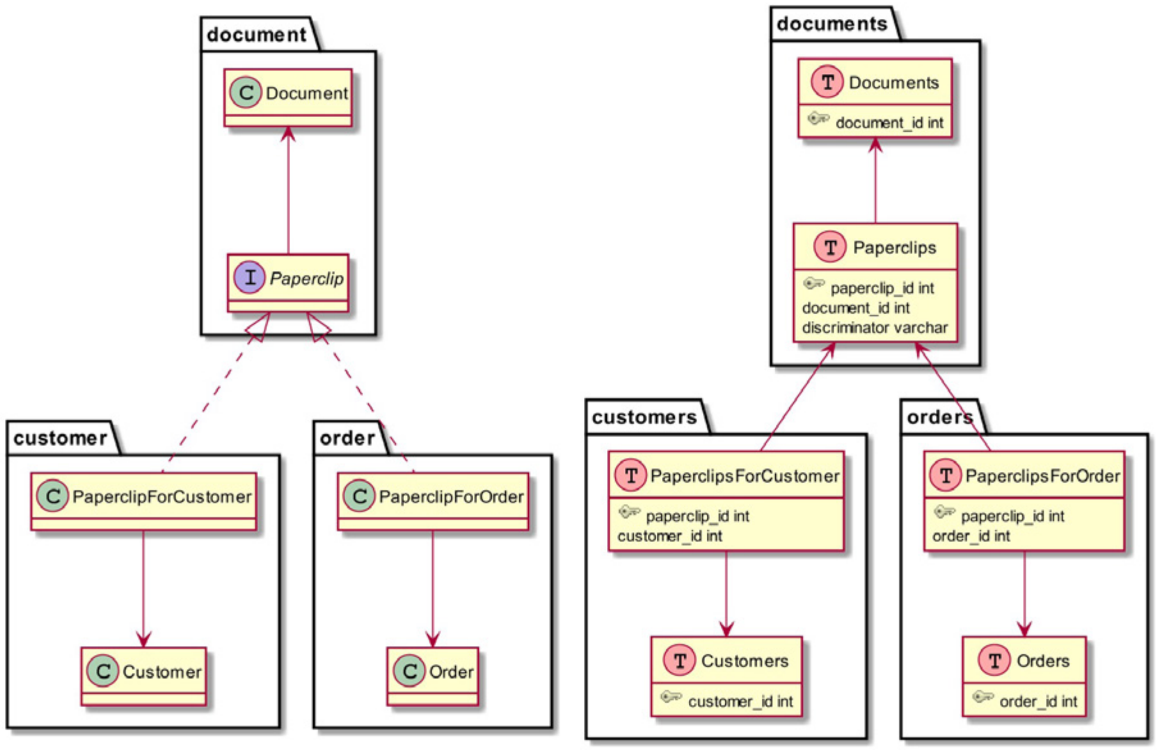


Figure 6: Polymorphic associations

implementation, for example PaperclipsForCustomer. The Paperclips.discriminator column indicates the concrete subtype.

What's nice about this mapping is we can still leverage referential integrity between all the tables in the database, while in the code we have a natural use of the Paperclip interface.

The patterns described above show that there are techniques to tackle structural decoupling of the database, but this doesn't necessarily address behavioural coupling. Earlier, we identified the problem that a developer working in module A could write a SELECT statement directly querying the tables owned by module B. How should this be tackled?

The solution used on the monoliths I work on is to make the ORM the way in which database interactions are performed; ad-hoc SELECT statements are verboten. On the .NET monolith I work on, we use Entity Framework, and each module corresponds to a separate DB Context. This also handles structural issues; EF only manages foreign keys within the module/DB Context, and we use the polymorphic link pattern described above to handle relationships between modules. For the Java monolith, we use DataNucleus (which implements JDO and JPA APIs); again, each module has its own persistence context.

You may well ask, what of those use cases where an ORM doesn't work? The glib answer is that it's worth investing the time learning to use the ORM effectively; chances are that it does work, actually. That said, in both monoliths, we handle special cases – typically where large volumes of data are required from two or more modules - using views which JOIN the tables from the relevant modules. The ORM neither knows nor cares that the entity is mapped to a view rather than a table. This is a performance optimization; the view effectively co-locates the business processing with the data. The view definitions are also trackable as code artefacts in their own right; we can see where we've deliberately chosen to subvert module boundaries in order to meet some user goal.

Transactionality (& Synchronicity)

It's common for a business operation to result in a change of state in two or more modules. For example, consider an invoicing application where we want to perform an invoice run. This will mostly modify state only in the invoicing module, creating new Invoice and InvoiceItem objects. However, if some custom- ▶

ers want their invoices to be sent out by email, then it might as a side-effect create Document objects (in the documents module), and Communication objects (in the communications module).

In a microservice architecture we have no transactions across services, which in general means we must use messages to coordinate such changes. The system therefore has only eventual consistency, and compensating actions are used to "back out" the change if something goes wrong. In some systems, this eventually-consistent behaviour can be confusing to the end-user, and to the developer too. For example, in the CQRS pattern that separates out writes from reads, a change written against one service will not immediately be available to read from another.

For a monolith though, if the backing data stores for the invoicing, documents and communications modules are all co-located in the same RDBMS, then we can simply rely on the RDBMS transaction to ensure that all the state is changed atomically. From an end-user perspective, everything remains consistent; there are no potentially confusing interim states or compensating actions to worry about. For the developer, they can expect that writes written to the database will be there to read immediately.

Synchronous behaviour can improve the user experience in other ways too. Imagine that each Customer has a collection of associated EmailAddresses, and that one of these EmailAddresses is nominated as the one to send invoices to. Suppose now that the end-user wants to delete that particular EmailAddress. In this case, we want the invoicing module to veto the deletion, because that email address is "in use". Basically, we want to enforce a referential integrity constraint across modules.

While supporting this use case in a microservice can be complicated, in a monolith we can easily handle the requirement. One design is to use an internal event bus, whereby the customer module broadcasts the intention to delete the EmailAddress, and allows subscribers in other co-located modules to veto the change:

```
public class Customer {
    ...
    @Action(domainEvent = EmailAddressDeletedEvent.class)
    public void delete(EmailAddress ea) {
        ...
    }
}
```

Listing 1: Customer action to delete email address, emitting an event

with a subscriber:

```
public class InvoicingSubscriptions {
    @Subscribe
    public void on(Customer.EmailAddressDeletedEvent ev) {
        EmailAddress ea = (EmailAddress)ev.getArg(0);
        if(inUse(ea)) {
            ev.veto("Email address in use by invoicing");
        }
    }
    ...
}
```

Listing 2: Invoicing subscriber of the delete email address event

The underlying technical platform would automatically emit the EmailAddressDeletedEvent onto the internal event bus, prior to invoking the delete. The subscriber can, if required, veto this interaction for the provided email address, if it is in use.

A different, more explicit design is for the customer module to declare a service provider interface (SPI) and then allow other modules to implement that SPI:

```
public class Customer {
    ...
    public void delete(EmailAddress ea) {
        ...
    }
    public String validateDelete(EmailAddress ea) {
        return advisors.stream()
                       .map(advisor -> advisor.cannotDelete(ea))
                       .filter(reason -> reason != null)
                       .findFirst().orElse(null);
    }

    public interface DeleteEmailAddressAdvisor {
        String cannotDelete(EmailAddress ea);
    }

    @Inject
    List<DeleteEmailAddressAdvisor> deleteAdvisors;
}
```

Listing 3: Customer action to delete email address, with validation and an "advisor" SPI

with an advisor class implementing the SPI:

```
public class Invoicing implements DeleteEmailAddressAdvisor {
    public void cannotDelete(EmailAddress ea) {
        if(inUse(ea)) {
            return "Email address in use by invoicing";
        }
        return null;
    }
    ...
}
```

Listing 4: Invoicing module implementation of the "advisor" SPI

Here the validateDelete method is a guard called before the delete method; it is used to determine if the delete may be performed for this particular email address. Its implementation iterates over all injected advisors; a non-null return value is interpreted as the reason that the EmailAddress cannot be deleted.

Here's another use case. In figure 6 we saw how different modules might provide the ability to attach Documents to their respective entities by way of Paperclip implementations. One can imagine that the documents module might contribute an "attach" action that would allow Documents to be attached, but this action should only be made available in the UI for those entities for which a Paperclip implementation exists. Again, the documents module could discover which entities expose the "attach" action either by emitting events on an internal event bus, or through an SPI service. ▶

For example:

```
@Mixin
public class Object_attach {
    private final Object context;
    public Object_uploadDocument(Object ctx) { this.context = ctx; }

    public Object attach(Blob blob) {
        Document doc = asDocument(blob)
        paperclipFactory().attach(context, doc);
    }
    public boolean hideAttach() {
        return paperclipFactory() == null;
    }

    public interface PaperclipFactory {
        boolean canAttachTo(Object o)
        void attach(Object o, Document d);
    }
    PaperclipFactory paperclipFactory() {
        return paperclipFactories.stream()
                                 .filter(pf -> pf.canAttach(context))
                                 .findFirst().orElse(null);
    }

    @Inject
    List<PaperclipFactory> paperclipFactories;
}
```

Listing 5: Mixin to attach Documents to arbitrary objects

The idea here is that the Object_attach class acts like a mixin or trait, contributing the attach action to all objects. However, (via the hide method) this action is not shown in the UI if there is no PaperclipFactory able to actually attach a document to the particular domain object acting as the context to the mixin.

Platform Choices

Whether you build yourself a monolith or a microservices system, you'll need some sort of platform or framework on which to run it.

For microservice architectures the platform is mostly focused on the network; it needs to allow services to interact with each other (protocols, message encodings, sync/async, service discovery, circuit breakers, routers, etc.) and to be able to run up the system in its entirety (Docker Compose, etc.). The language to implement any given individual service is less important, so long as it can be packaged, e.g. as a Docker container (of course, the project team must have the appropriate skills in that language for initial development and ongoing maintenance/support).

For monoliths, too, a common platform is required, but here the focus is more on the language and supporting ecosystem. At a very minimum this will be the technology platform such as Java or .NET. On top of this you'll probably also adopt some framework, JEE and Spring being common choices.

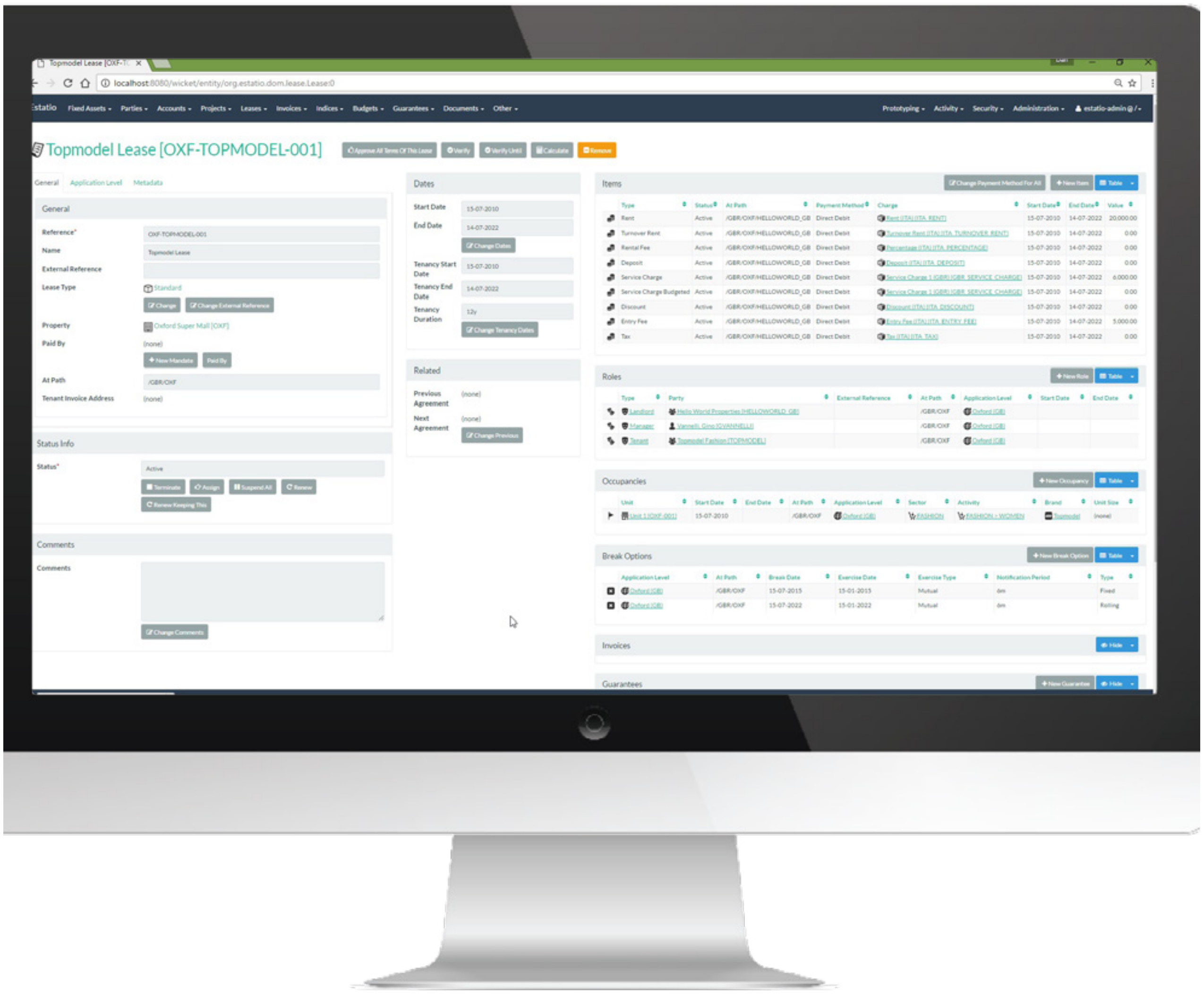
Because a monolith's strength is dealing with complex domains, the underlying platform should pick up as many technical/cross-cutting concerns as possible: security, transactionality and persistence are the obvious ones (there are others, as we'll see). Moreover, business modules should not depend on the technical modules; we want to get as close to the hexagonal architecture as possible.

It's also important for a monolith's platform to provide tools allowing business modules to be decoupled from each other. A solution to this for a monolith is remarkably similar to that of a microservice: use an event bus. The difference is that with a monolith, this event bus is intra-process and is also transactional.

A (Modular) Monolith Example

To help make the case for a modular monolith, we end with a real-world example.

The application in question is called Estatio, an invoicing sys-



Topmodel Lease [OXF-TOPMODEL-001]

General

Reference*	OXF-TOPMODEL-001
Name	Topmodel Lease
External Reference	
Lease Type	Standard
Property	Oxford Super Mall [OXF]
Paid By	(none)
At Path	/GBR/OXF
Tenant Invoice Address	(none)

Status Info

Status*	Active

Comments

Comments	

Dates

Start Date	15-07-2010
End Date	14-07-2022
Tenancy Start Date	15-07-2010
Tenancy End Date	14-07-2022
Tenancy Duration	12y

Related

Previous Agreement	(none)
Next Agreement	(none)

Items

Type	Status	At Path	Payment Method	Charge	Start Date	End Date	Value
Rent	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Rent (ITA) (ITA_RENT)	15-07-2010	14-07-2022	20,000.00
Turnover Rent	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Turnover Rent (ITA) (ITA_TURNOVER_RENT)	15-07-2010	14-07-2022	0.00
Rental Fee	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Percentage (ITA) (ITA_PERCENTAGE)	15-07-2010	14-07-2022	0.00
Deposit	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Deposit (ITA) (ITA_DEPOSIT)	15-07-2010	14-07-2022	0.00
Service Charge	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Service Charge 1 (GBR) (GBR_SERVICE_CHARGE)	15-07-2010	14-07-2022	6,000.00
Service Charge Budgeted	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Service Charge 1 (GBR) (GBR_SERVICE_CHARGE)	15-07-2010	14-07-2022	0.00
Discount	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Discount (ITA) (ITA_DISCOUNT)	15-07-2010	14-07-2022	0.00
Entry Fee	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Entry Fee (ITA) (ITA_ENTRY_FEE)	15-07-2010	14-07-2022	5,000.00
Tax	Active	/GBR/OXF/HELLOWORLD_GB	Direct Debit	Tax (ITA) (ITA_TAX)	15-07-2010	14-07-2022	0.00

Roles

Type	Party	External Reference	At Path	Application Level	Start Date	End Date
Landlord	Hello World Properties (HELLOWORLD_GB)		/GBR/OXF	Oxford (GB)		
Manager	Vannelli, Gino (GVANNELLI)		/GBR/OXF	Oxford (GB)		
Tenant	Topmodel Fashion (TOPMODEL)		/GBR/OXF	Oxford (GB)		

Occupancies

Unit	Start Date	End Date	At Path	Application Level	Sector	Activity	Brand	Unit Size
Unit 1 (OXF-001)	15-07-2010		/GBR/OXF	Oxford (GB)	FASHION	FASHION > WOMEN	Topmodel	(none)

Break Options

Application Level	At Path	Break Date	Exercise Date	Exercise Type	Notification Period	Type
Oxford (GB)	/GBR/OXF	15-07-2015	15-01-2015	Mutual	6m	Fixed
Oxford (GB)	/GBR/OXF	15-07-2022	15-01-2022	Mutual	6m	Rolling

Invoices

Guarantees

Figure 7: Estatio Screenshot

tem for Eurocommercial Properties, a real-estate company that owns and operates (at the time of writing) 34 shopping centres in three European countries. The source code for Estatio can be found on GitHub.

The underlying technology platform/framework for Estatio is Apache Isis, a full-stack framework for the JVM that handles all the usual cross-cutting concerns such as security, transactionality and persistence. However, it goes further than this in also automatically rendering domain objects either through a web UI or through a REST API, following the naked objects pattern. In the same way that an ORM automatically maps/marshals a domain object into a persistence layer, you can think of Apache Isis as mapping that domain object into the presentation layer.

Because the UI is generic, it can be steadily improved/enhanced with no changes to the domain object model. For example, in a previous release, the Apache Isis viewer was improved to use Bootstrap for styling. Every application that updated to this release was then "magically upgraded" with the improved viewer. When capabilities such as maps, calendars or Excel exports have been added, they too are rendered automatically in the UI everywhere that the framework can infer that they apply.

Because interactions to the business domain objects go "through" the generic UI provided by Apache Isis, then a whole bunch of other cross-cutting concerns can also be tackled. For example, Apache Isis automatically creates a command memento (serializable to XML) for every action invocation or property edit, and this can then be published to an event bus such as Apache Camel as the transaction completes. It also correlates this command with an audit trail, providing full cause-and-effect traceability of every change made to every domain object.

The framework works by building an internal metamodel (similar to how ORMs work), and this ▶

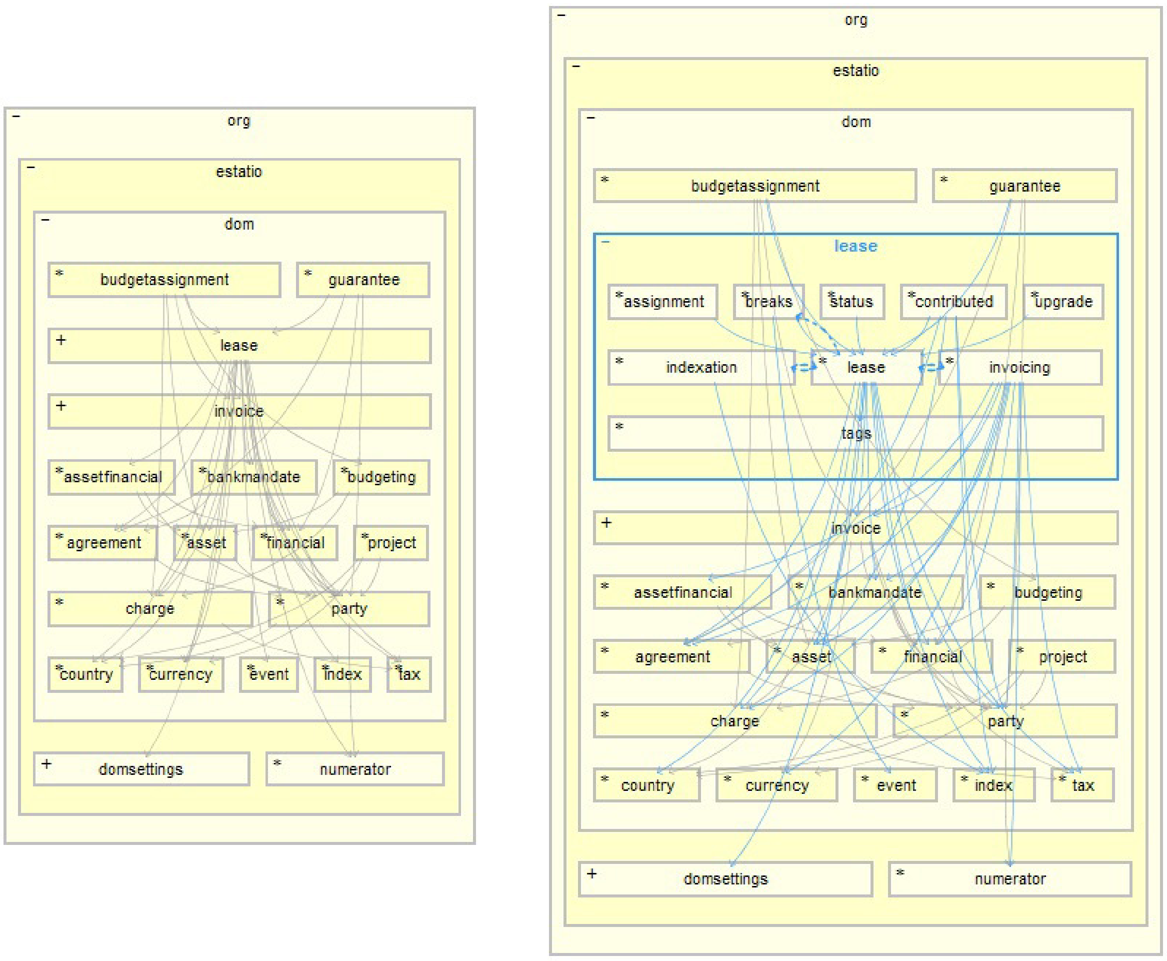


Figure 8: Estatio Modules

metamodel can be exploited for other purposes than just the generic UI and REST API. For example, a Swagger interface file can be exported to allow custom UIs to be built against the REST API, while the powerful security module defines roles and permissions with respect to the properties and actions of the domain object types. The metamodel is also used to generate gettext ".po" files to be translated for i18n. It's also possible to define metamodel validators to enforce architectural standards, for example, that every entity in a given module is mapped to the correct database schema.

With the framework handling so many of the technical concerns, the developer is able to focus on the domain, ensuring that it is properly modularized for long-term maintainability. To help modules stay fully decoupled, the framework supports the concept of mixins, whereby the rendering of a given domain object can include state and behaviour from several modules without there actually being any coupling of the business modules themselves. The ability to attach Documents to arbitrary objects is a good example; the code in listing 5 above is very similar to the Apache Isis programming model.

Equally important is the provision of an internal event bus. Rather than having one module directly call another, it can just emit an event which other modules can then subscribe to. The code listings 1 and 2 are once again examples of how Apache Isis supports this.

Persistence patterns such as support for polymorphic associations (figure 6) are also important. These are implemented by various open source modules in the Incode Catalog to support generic subdomains such as documents, notes, aliases, classifications, and communications.

A further extensive set of modules can be found at Isis Add-ons. These tackle technical concerns such as security, auditing, and event publishing. The extensions to the Apache Isis viewer (maps, calendars, PDF, etc.) are also to be found here.

To make both the generic business subdomains and technical add-ons easy to reuse, each is supported by its own demo app and integration tests. The would-be consumer of these apps can therefore check them out easily to see if they fit requirements.

So much for Apache Isis and its supporting ecosystem; the proof of the pudding is in the eating. What the technical platform should enable is the ability for the development team to concentrate on the core domain, with that domain broken up into modules. And so, if you inspect the Estatio codebase you will indeed see that it consists of a number of separate modules. Figure 8 shows how these depend on each other (diagram generated using Structure101).

In the diagram on the left-hand side of figure 8, each box represents a separate Maven module, and the lines represent dependencies between the modules.

Towards the bottom are utility modules (domsettings, numerator) or modules that contain strictly reference data (country, currency, index, tax, charge).

Moving into the middle we see the agreement, party, financial, asset, assetfinancial and bankmandate modules; neither the structure of these modules nor the data within them changes that often. By the time we get to budgeting, invoice and in particular lease, we are at the heart of the system; these are the modules that depend most on the other submodules.

The diagram on the right-hand side of figure 8 is almost the same, however the lease module has been expanded into its sub-packages. Here we can start to see some bidirectional dependencies, suggesting that this code could perhaps be improved. There are certainly a lot of outbound dependencies, so the module is probably doing too much. No software is perfect. Then again, while lease is the largest module in the system, it's still conceptually small enough for us to work on ("a lease is an agreement between two parties – a tenant and landlord – that calculates invoices").

Estatio is now almost five years old as an application, with its scope set to continue to expand to support further use cases. But its code base may shrink even as its scope expands; the majority of the modules in Isis Add-ons and Incode Catalog were factored out of Estatio, and we expect to factor out further modules in the future. And if you cloned its repo today to take a look, you might find it has moved on from the above diagrams. That's to be expected; this software is intended to have a long-shelf life, and will continue to evolve.

Conclusions

Initially, we compared the modular monolith with the microservices architectures, exploring the benefits and weaknesses of both.

We also asked the question: "which architecture should you go for, microservices or monoliths?" And we answered by asking a different question: "what is it you are trying to optimise for?" If on balance you've decided that the risk of domain complexity outweighs the risk of not being able to scale, then you should have decided to implement a modular monolith. Hopefully the various techniques and patterns we've described here will assist.

Technical platforms are important whatever the architecture; there's no point in reinventing the wheel. A framework such as Apache Isis will allow you to channel your energies into tackling the complexities of the domain, helping you explore the module boundaries, while mopping up almost all of the technical cross-cutting concerns (including the presentation layer).

We also looked at a substantial open source application, Estatio, that uses Apache Isis as its underlying platform, showing what a modular monolith looks like "in the flesh".

Neither monoliths nor microservices is a silver bullet; the answer to "which should I go for?" is always "it depends", and anyone who tells you otherwise is selling you snake oil. Consider where your system fits with respect to scalability vs. domain complexity, and take it from there. ■

The Journey from Monolith to Microservices: A Guided Adventure

Mike Gehard is a senior software engineer at Pivotal Labs. He works with clients to migrate legacy, monolith applications onto the Spring IO platform and eventually to microservices. He's also worked on the Spring Cloud services team.

Adapted from a presentation by Mike Gehard at SpringOne Platform in August 2016, and originally published on InfoQ on Jan 20, 2017.

This is a story of a recent migration from a monolith to microservices. It should provide good information to enable you to make smart decisions, rather than receiving strict guidance that needs to be followed exactly.

Nearly every developer falls into one of three categories when it comes to monoliths and micro-services: supporting a monolith that needs migrating; actively migrating a monolith; or building net new microservices. The information provided here has something for all of these groups.

Where did these ideas come from?

This whole discussion started with Mike Barinek at Pivotal, who came up with the idea of an App Continuum. Depending on how much knowledge you have about your system and how much code you have, you are somewhere on the continuum; it is not an "either/or", it's a "yes, and". If you've got ten lines of code, you're probably on the left-hand side (see Figure 1), with an unstructured system, possibly just a single folder with some class files. Adding more code leads to namespaces. One application with many libraries can progress to multiple applications sharing some libraries. Eventually, you add some services, shown

KEY TAKEAWAYS

A project's location on the App Continuum depends on how much knowledge you have about your system and how much structure exists in the codebase.

Before building a microservices architecture, start with a well-structured monolith.

Bounded contexts, a concept from Domain-Driven Design, are necessary building blocks for adding structure to your codebase. They encapsulate the business logic which can be extracted into a single microservice.

Focusing on activities which help you identify and visualize your domain adds structure, making it easier to add developers to a project and aiding testing.

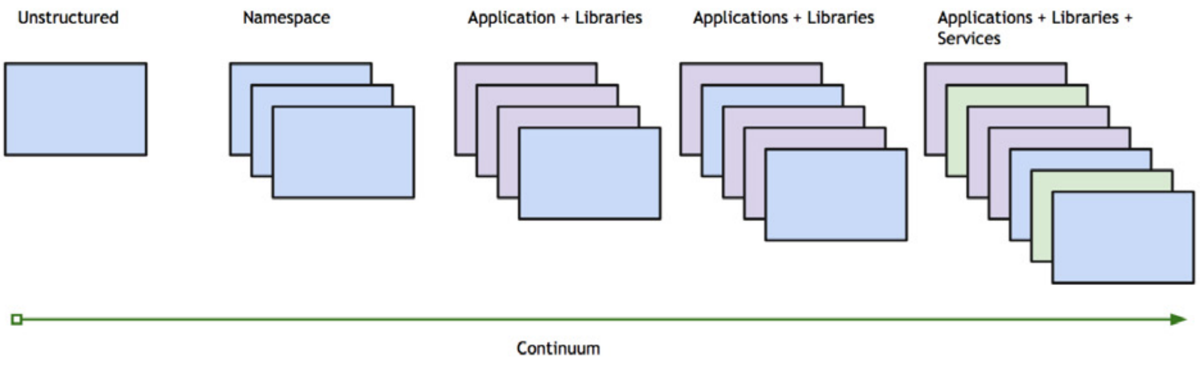
Microservices require supporting applications, such as service discovery and circuit breakers. The additional support these apps require make your first microservice the most expensive, but once they exist, standing up additional services is more economical.

as the green boxes on the right-hand side, and congratulations, you have a distributed system.

Simon Brown came up with a similar idea, called a Modular Monolith. If I build a codebase that is well structured, when I go to microservices all I do is take it apart. A well-structured monolith provides many benefits, including high cohesion & low coupling, is focused on a business capability, encapsulates data, and is composable. Microservices provide all the benefits of a modular monolith, as well as having individually deployable, upgradeable, replaceable and scalable services in a heterogeneous technology stack. If you are looking for the first set of features, a modular monolith may be a good solution. While a monolith may be long-lived, modularity can facilitate moving to microservices later.

What do these two ideas have in common? Looking back at the app continuum, moving to the right adds structure to the codebase. My hypothesis is that the more structure I have in my application, the better off I'm going to be in the long-term. ▶

Well-defined, in-process components are a stepping stone to out-of-process components.



If you can't build a well-structured monolith, what makes you think you can build a well-structured set of microservices?

- Unknown

These ideas may not sound entirely new. We've been talking about ideas like the Single Responsibility Principle for fifteen years. What we haven't been doing for very long is implementing those ideas. We've also been doing it subtly wrong in a number of ways, which we'll discuss later.

Which comes first?

Like any interesting question, the answer to "Monolith or microservices, which comes first?" is, of course, "It depends."

Again looking at the app continuum, it represents the level of understanding about a system. If I'm a startup with no idea what my business model is, and I begin to build microservices, that's probably not a good idea. But, if I'm a bank, and I've been doing banking for 25 years, and can forecast doing roughly the same for the next 25 years, then it's probably okay to be further along on the continuum, since I have a general idea of what my app is doing.

My favorite quote is, "If you can't build a well-structured monolith, what makes you think you can build a well-structured set of microservices?" Let's be honest with ourselves. If we can't take one codebase and make it look good so we can work in it for a long period of time, what makes us think we can just magically birth microservices into the world and have those things be well-structured?

The worst thing you can do for yourself is build what I call a distributed monolith. If you have a microservices architecture that's really chatty and you're doing distributed transactions across different microservices, you now have a distributed monolith. Congratulations, you now have the worst of all worlds.

My hypothesis is that a well-structured monolith is the right starting point. The main reason is because I don't trust myself to do this right the first time. I want a codebase I can experiment in very easily. If I have a monolith, it's very easy to push files around inside of one codebase. If I have twelve microservices, and I need to move some files around, I'm lucky if they're all in the same repository (I probably have twelve different repositories). The more I know, and the more stable my codebase is, the more likely I am to build a good set of microservices. If I'm still moving boundaries around, and I'm trying to find where the user boundary lives, then I probably want to stay in a monolith. That's why I do it; so I can make a bunch of mistakes, and those mistakes are reversible in a monolith.

I've talked about boundaries and structure, and the key is an idea called Bounded Contexts, discussed in two books, Domain-Driven Design, by Eric Evans in 2003, and Implementing Domain-Driven Design by Vaughn Vernon in 2013. The problem with Eric Evans' book is he puts bounded contexts at the back of the book, and by the time you get there your head is spinning with all these new ideas. I like Vaughn Vernon's book because he talks about bounded contexts right up front.

A bounded context is a business concept in my app. So if I'm building a shipping piece of software, I probably have shipments, and users, and packages. Those might be my bounded contexts. These are things in the domain, not architectural things.

This is where SOA went wrong. We put a lot of this logic into the enterprise service bus and into the infrastructure, which makes it really hard to separate things because we now have coupling to the infrastructure. The idea of a bounded context is to put all the stuff in a box, and to make the pipes super dumb. HTTP is a pretty "dumb" protocol.

This has been talked about for years; Eric Evans' book was from 2003. It's not a new concept, it's just really hard to get right because it depends on your bounded contexts, and it takes a lot of experimentation to get the boundaries correct.

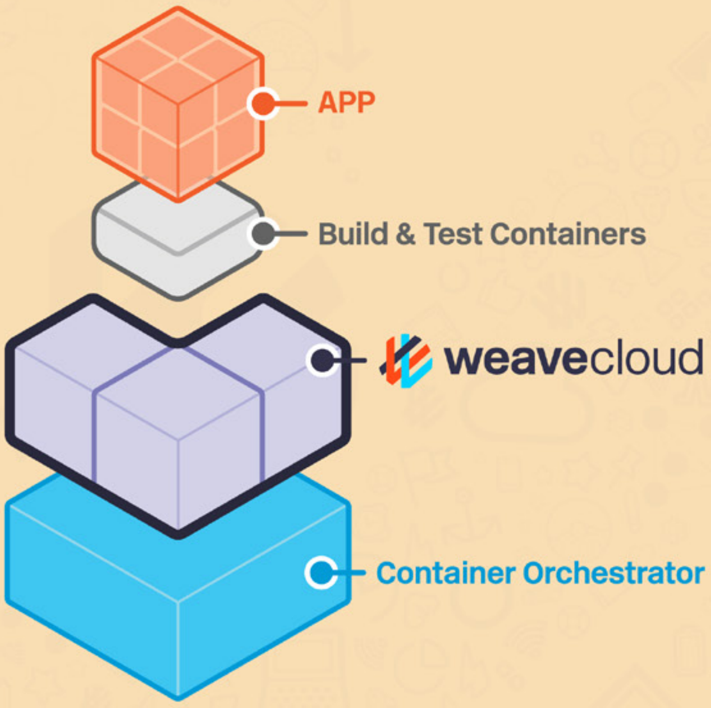
The project

We did a proof-of-concept for a big telecommunication company feeling pressure from Netflix and Hulu. When they came to us, they had a Rails app, which wasn't scalable. The project had several goals. First, to migrate a monolith that was in production and making money, to a more sustainable solution for the future. They weren't concerned about their business model at this point, because Netflix and Hulu had proven that using the internet to get TV shows was a valid business model. But, it had to be sustainable because they wanted to make money off of this.

Second, they knew they needed scalability. As Netflix has proven, these apps need to be able to scale. You have lots of users, and the way to make more money is to scale the app. They also knew they needed multiple teams. They had a POC running in production, and they knew how big it was. They wanted to have multiple teams working in the codebase.

Finally, they also wanted to scale the resources up and down. They wanted to add people for a couple of weeks to get a bump in velocity, then they wanted to take those people off. And they wanted enough room in the swimming pool for everybody, without affecting the other teams.

The current state was a set of API servers, shipping JSON all over the world. There were multiple clients, including XBox, set-top boxes, iPhones, iPads, plus JavaScript front-ends. A fun challenge was dealing with the dead code, which was somewhere between zero and 100%, but in reality was somewhere around 35-30%. The codebase was inherited from a company they bought. They had not taken the dead code out, so we didn't know what they were using at the time. ▶



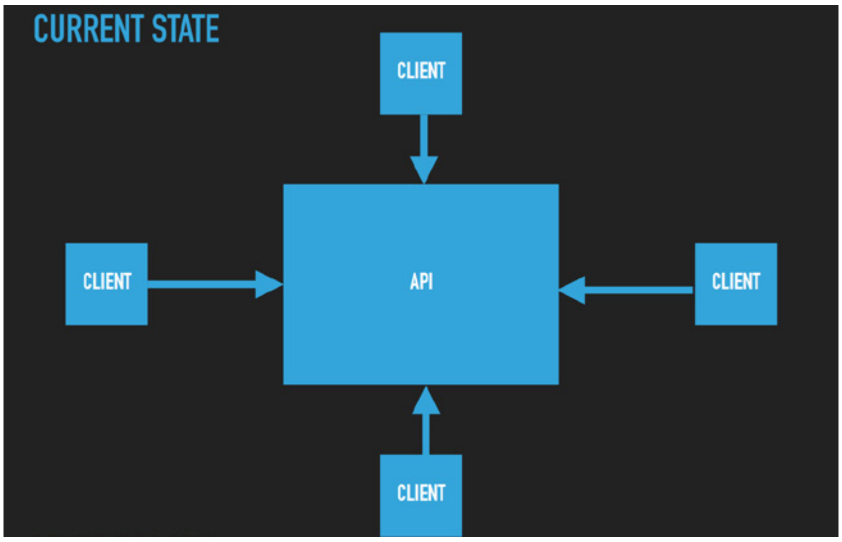


Figure 2: Current State

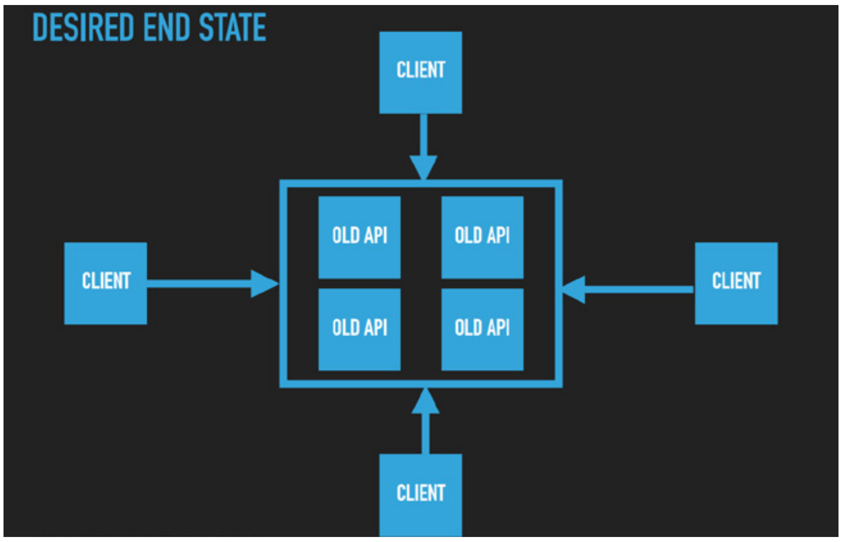


Figure 3: Desired End State

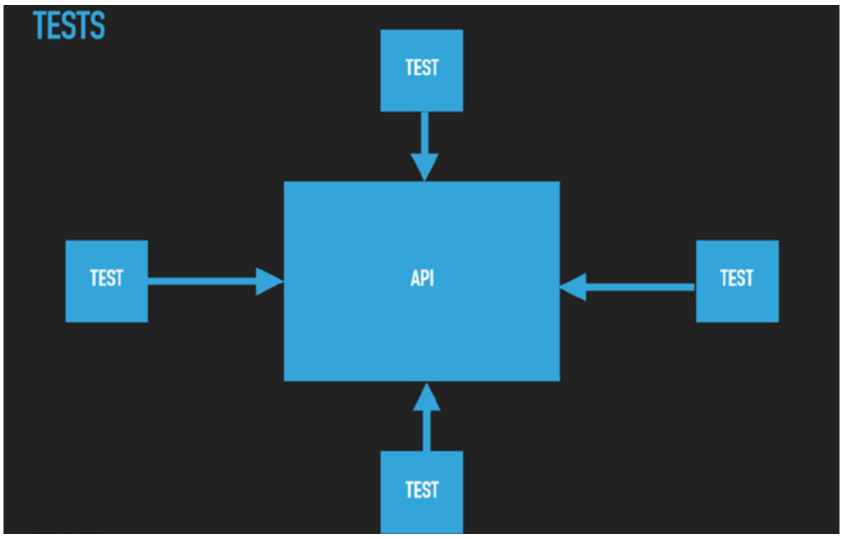


Figure 4: API Tests

This was coupled by a lack of comprehensive, current API tests, which made us pretty sad. They were running in production, without any way to test the system. They had already faced one expensive outage, so they were pretty risk-averse to breaking changes.

Step 1: API tests

At Pivotal, one of the things we do is Test-Driven Development. We felt we were walking on a tightrope, and we wanted to write some tests. Our first step was writing API-level tests. We didn't care about unit testing at this point, since we knew most of that code was going to be replaced. Comparing the current state in Figure 2 with the desired future state in Figure 3, what stays the same is the interface. All we're doing is smashing that thing into lots of pieces.

The goal of writing API-level tests was to validate that no client-facing behavior broke during the migration. The tests also made sure issues were found before we went to production.

For the API tests, we used a framework called Pact to write consumer-driven contract tests. Simply put, these tests are formed with a JSON doc that says, "When I give you this JSON, you will respond with this stuff."

When doing TDD and writing a Javascript front-end, the first step is to write a dummy server that just serves up canned JSON. Similarly, when writing a backend service, you write tests that pass in JSON and expect a 200 response with corresponding JSON. Pact tests allow us to specify that contract in the middle, then auto-generate both sides of that. This means the client code can run the tests in its test suite, and the server side can run the tests in its test suite. As long as the contract hasn't changed, everyone knows that they're adhering to the contract. If you change the contract, and the server side tests fail, then you know the server is no longer satisfying the contract, and the client will break if you don't fix it.

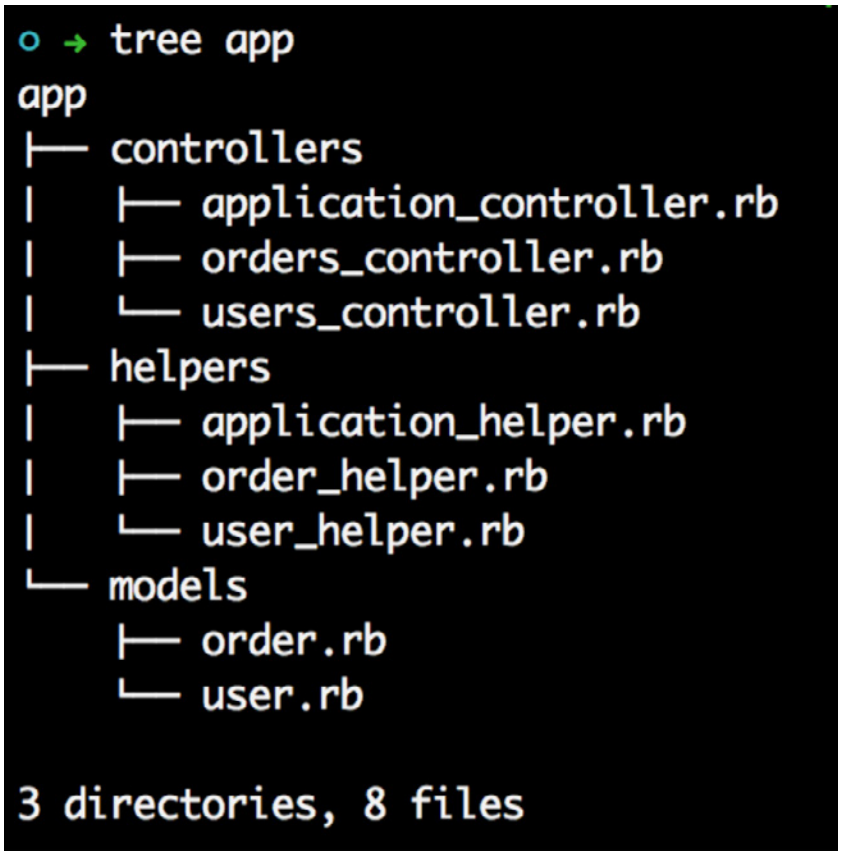
The result was we had tests that would tell us, before we went to production, if we had broken anything. Again, we had a system running in production, making money. It's like flying an airplane, then pulling up another airplane alongside, and moving people between them. If you don't do that with a safety net, you're going to drop a couple, and that's always bad for business.

Step 2: Arrange application so you can see your domain

Step two was to move code around so we could see our domain and begin to understand what bounded contexts exist. The reason we have trouble breaking up monoliths is because everything is tangled together. If I have clearly defined bounded contexts, then I can just ship them in different directions.

Let's compare the two sample application project structures shown in Figures 4 and 5. In the first, making a change to Users will require modifications in up to three directories. In the second, modifying Users will be constrained to a single directory. Figure 5 also makes it more obvious that the app has two bounded contexts, Orders and Users.

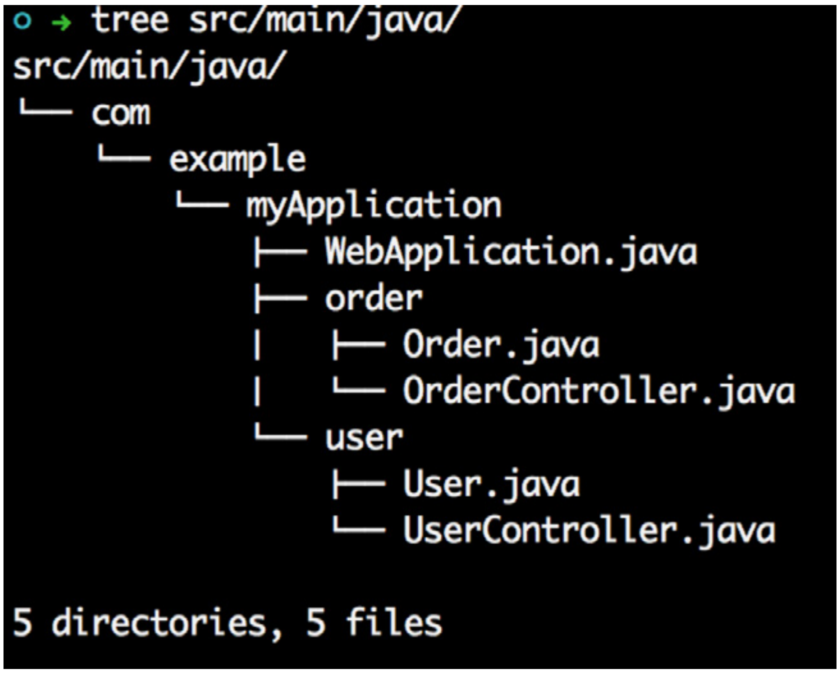
The first structure is the old way of using horizontal layers of architecture, with layers for models, views and controllers. The alternative is to have vertical slices, in this case a Users slice and an ▶



```
o → tree app
app
├── controllers
|   ├── application_controller.rb
|   ├── orders_controller.rb
|   └── users_controller.rb
├── helpers
|   ├── application_helper.rb
|   ├── order_helper.rb
|   └── user_helper.rb
└── models
    ├── order.rb
    └── user.rb

3 directories, 8 files
```

Figure 4: Sample app structure #1



```
o → tree src/main/java/
src/main/java/
└── com
    └── example
        └── myApplication
            ├── WebApplication.java
            ├── order
            |   ├── Order.java
            |   └── OrderController.java
            └── user
                ├── User.java
                └── UserController.java

5 directories, 5 files
```

Figure 5: Sample app structure #2

Orders slice. Each vertical slice is a bounded context. This begins to add structure to the project, but the structure is in service to bounded contexts, not in service to a layer of the architecture.

These two examples, although trivial, show how arranging your application so you can see the domain provides several benefits. In addition to minimizing the number of directories where changes happen, it also becomes less costly to experiment with and evolve bounded contexts. This arrangement also allows you to delay architectural decisions. As "Uncle Bob" Martin advocates, good architecture allows you to delay decisions until you have more information.

Step 3 - Break out components

With our bounded context defined and structured in our codebase, next we start to separate those further apart. If we previously had a rice paper wall between bounded contexts, we're now going to be adding some drywall.

In our Java app, we have an `applications` directory that contains a `Controller` and the `build.gradle` file. Next to the `applications` directory is a `components` directory, and this is where the domain lives, with subdirectories for `billing` and `email`. This is the important separation between domain code and architecture framework code.

What about databases?

One question that always comes up is, "Where does the database live?" The answer is, of course, it depends. In some cases, the application can manage the database. In others, it makes sense for components to manage the portion of the database they are concerned with.

Regardless of where the database is managed, one fundamental rule is migrations only touch one table. If I have database Table A and database Table B, I'm lucky if a migration takes them both in the same direction. More likely, Table A will go one direction, while Table B goes another. If I have the code for both of those changes in a single migration file, then I have to split that migration file up. If I go to microservices, then Table A will go to Microservice A, while Table B goes to Microservice B. I should therefore treat each of those as separate migrations, with only one table affected at a time.

Benefits of breaking out components

As components are broken out in the codebase, it creates more room for multiple teams to work on the app. I can have a Users team who only work in the User component directory, and maybe one directory in a controller. One team doesn't have to worry about their changes breaking another team's changes.

Clear boundaries now exist to separate domain layer code from framework code. I'm also moving closer to microservices. A microservice is simply a bounded context with an HTTP or a messaging interface. At this point, I have a microservice; it's just not being server via HTTP. Stopping here will provide a lot of the benefits, without the overhead of microservices.

This is hard. It can take eight months to a year to get to this point. If your monolith is a big mess, without a lot of structure, you have to work to add it back into the codebase.

Step 4: Promote your first microservice

Congratulations, you've now gotten to a point where you can create your first microservice. There are a few reasons when this makes sense.

First, you may want to scale a certain bounded context, independent of all the other bounded contexts. In a monolith, I have to ship out more of all the bounded contexts. But with a microservice, I can just ship out more of that one bounded context, and it can scale much more quickly.

I may want to deploy one bounded context more frequently. If there's a part of my business that is iterating faster than the rest of the business, a microservice allows me to deploy at their pace. On a monolith, this will be a slower pace, as there will usually be more code and business units involved. Deploying a microservice is also less risky because there is less code; the less code moving to production, the less risky a deployment is.

There are many other reasons to move to microservices. Sam Newman's book, Building Microservices, covers twelve of thirteen of them, and is my favorite book on microservices.

Why not extract a microservice?

Sometimes, it doesn't make sense to extract microservices. If the costs of managing a microservice outweigh the benefits, then it probably makes sense to stay with a well-structured monolith.

Dysfunctional organizational patterns are also a warning sign when considering microservices. This ties in to Conway's Law, which says the system will reflect the communication patterns of the organization. If you have dysfunctional organizational patterns, and you go to microservices, you will have bad communication patterns between your microservices. This forces you to change your org at this time, because you cannot go any further if you stay in the old org structure.

Moving the code is now the easy part

After creating a well-structured monolith, with clearly defined bounded contexts within components, separating the application is fairly straightforward. If I have components, all I have to do is spin up another application. I take the controllers and other stuff and put them into a new Spring Boot application. I add my .jar file and I'm off and running. That's it. It's literally just moving a .jar file into another application.

Service discovery

Congratulations, you now have a distributed system, along with all the pain and suffering that comes with that. If you have two microservices, for Billing and Email, where does the Billing service live, and where does the Email service live? You also have network communications. The network will fail at some point, so you need to deal with that.

Microservices don't come for free. That's what people don't tell you.

We want to use service discovery to solve the problem of location. If you're using Spring Cloud Services, Eureka can be used for service discovery. This creates a third party responsible for managing the location and quantity of the available services. The billing ser- ▶

Microservices don't come for free. That's what people don't tell you.

vice first calls the service discovery service, and then is able to call the email service. This can replace any hard-coded references to the email service, which would have to change every time a new instance of the service is stood up.

Service discovery allows service calls to benefit from this loose coupling between location and number of instances. It also enables client-side load balancing, which can reduce the number of network calls. In Netflix OSS, this is called Ribbon.

However, service discovery adds another application to monitor. In contrast to the two services which add business value, this third application adds no business value. Said another way, 33% of my system is not adding business value.

Circuit breaker

Now that a network exists between services, that network will fail at some point. We need to protect against cascading failures that cause system downtime. The solution to this problem is to use circuit breakers.

Similar to adding service discovery, the circuit breaker service is stood up, and called to check the availability of a dependent service. This does add another hop in the call stack. Before the billing service can call the email service, it calls the circuit breaker to make sure the email service is up. If I can tell the call to the email service will fail, then I shouldn't bother calling it. It can also tell if the email service is just slow, because the last thing you want to do to a suffering service is to hammer it with more load.

We've increased the resiliency of the system, because when the network fails it won't bring down the entire system. We've also increased visibility of the health of the system. Typically, with a circuit breaker you get a dashboard that shows you all the circuits that are open or closed. This can provide a very handy tool to monitor the system, know what's down, and what needs to be repaired.

At this point, we have four applications, only two of which are adding business value. Cloud Foundry comes in handy, because the cost of running those apps is a little less, but it still exists.

Next steps

You are now the proud owner of a set of microservices. Next, start breaking out more microservices. The overhead costs have already been paid. Adding a third microservice brings the total number of applications to five, since it won't require new service discovery or circuit breaker services. The first microservice is the most expensive.

Because you've created a boundary between your framework code and your domain, you can now easily switch out the communication patterns. If you want to go to RabbitMQ, the only thing you have to touch is something in the application folder. The domain doesn't care where it's getting its information from. That interface is stable, so you can just change to using a message queue, if you want to.

Or, you can do nothing. You can keep the system in this state, and continue to iterate on your business model. The decision is up to you.

This model is great because it allows you to delay those engineering decisions and do engineering analysis on what the next step should be. It's not emotion driven. There are costs and there are benefits. When the benefits outweigh the costs, then it's time to ship the next microservice.

Source Code

There are two examples of this. The first is based on the client project, and I rewrote it in my spare time. https://github.com/mikegehard/journeyFromMonolithToMicroservices.

I've also been writing Kotlin in my spare time. If you want to check out the Kotlin solution, I highly recommend this repo: https://github.com/mikegehard/user-management-evolution-kotlin ■

Git Ops - the fastest way from code to production

A decade of best practices says that config is code, and that code should always be stored in version control. Git has moved the state of the art forward in development and now it is paying that benefit forward to Ops.

The adoption of microservices means that developers are not only responsible for writing the code, but also for its deployment. With monoliths, changes to applications were large, infrequent, and required a lot of coordination. But now with microservices, small and frequent code changes can be deployed by independent teams at any time to a running app.

A "you build it, you own it" development process requires tools that developers know and understand. "GitOps" is our name for how we use developer tooling to drive operations.

What is GitOps?

GitOps is a way to do Continuous Delivery. It works by using Git as a source of truth for declarative infrastructure and applications. Automated delivery pipelines roll out changes to your infrastructure when changes are made to Git. But the idea goes further than that – it uses tools to compare the actual production state with what's under source control and tells you when it doesn't match the real world.

Git enables declarative tools

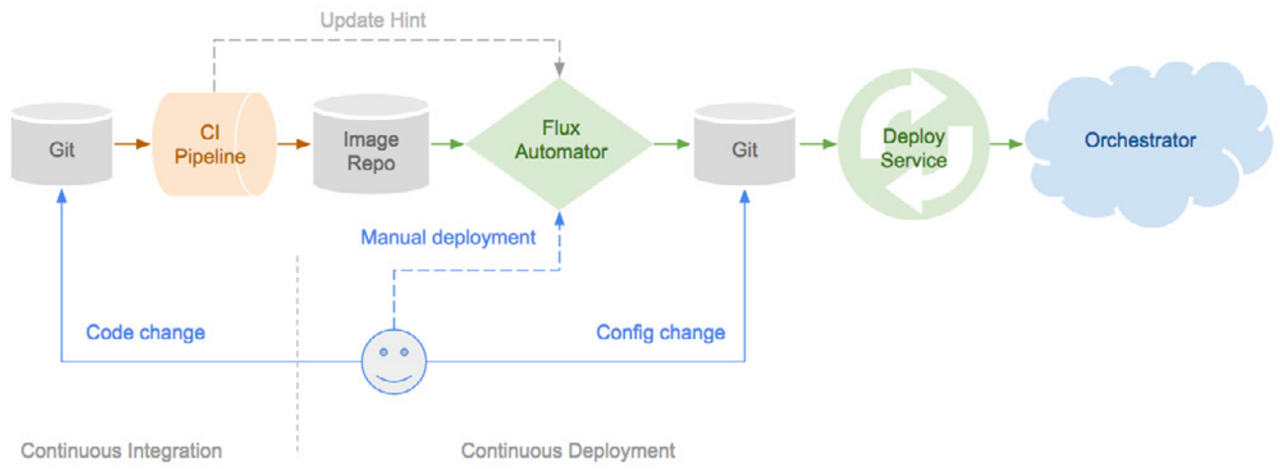
Kubernetes is just one example of many modern tools that are "declarative". Declarative means that configuration is guaranteed by a set of facts instead of by a set of instructions, for example, "there are ten redis servers", rather than "start ten redis servers, and tell me if it worked or not".

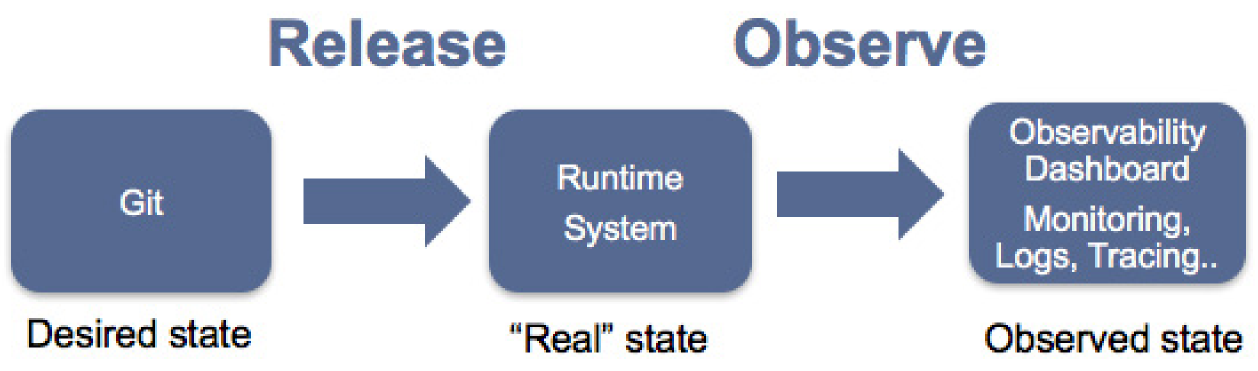
By using declarative tools, the entire set of configuration files can be version controlled in Git. This means that Git is the source of truth and that an entire infrastructure can be reproduced from Git.

GitOps empowers developers to embrace operations

The GitOps core machinery in Weave Cloud is in the CI/CD tooling and the critical piece is continuous deployment (CD) and release management which supports Git-cluster synchronization. Weave Cloud deploy is designed specifically for version controlled systems and declarative application stacks. Every developer can use Git and make pull requests and now they can use Git to accelerate and simplify operational tasks for Kubernetes as well.

A developer adds a new feature to his app and pushes it to GitHub as a pull request which triggers the GitOps pipeline to deploy to production: ▶





Observability is a pipeline catalyst

Observability can be seen as one of the principal drivers of the Continuous Delivery cycle for Kubernetes since it describes the actual running state of the system at any given time. The running system is observed in order to understand and control it, and new features and fixes are pushed to git and feeds the pipeline:

Git-centric tools accelerate delivery

Our goal is to help teams accelerate delivery. We provide Git-centric tools that unify pipelines with observability in ways that make developers love operations.

The role of a GitOps dashboard in Weave Cloud is to enable observation and to speed up both the understanding and validation of the system, and to suggest mitigating actions. This accelerates the operations cycle.

Final Thoughts

In the GitOps pipeline model, Git is the design centre. It plays the central role of "source of truth for everything in the system" - code, config and the full stack. CI, build and test services are necessary for constructing deployable artefacts. But in the GitOps pipeline, the overall orchestration of delivery is coordinated by the deployment and release automation system - triggered by updates to repos.

At Weaveworks, these principles are built into Weave Cloud. This not only helps customers ship apps faster, it also helps run a cloud native stack.

For further reading we recommend our blog series on GitOps:

- Operations by Pull Request
- The GitOps Pipeline
- GitOps Observability ■

Virtual Panel: Microservices in Practice

Originally posted by Mark Little on Feb 10, 2017

Microservices have gone from internal development practices for the select few so-called "unicorns," to something many developers in a wider range of organisations are embracing, or considering for their next project.

Some people believe that in order to deliver on the benefits of DevOps, microservices are a necessary requirement.

In the last few years we have seen new technologies and experiences shape microservices, often reinforcing their ties to Service Oriented Architectures at the same time as expanding on their differences. Some believe that technologies and methodologies which can assist in developing and adopting microservices are ineffective without associated changes within the organisations that wish to use them.

InfoQ spoke with five panelists to get different perspectives on the current state of the art with microservices, how they are likely to evolve, and to share their experiences, both good and bad, when developing with them. ▶

KEY TAKEAWAYS

Understand some of the lessons learned in the past few years, and real-world development with microservices.

Understand whether the principles for using microservices for brownfield development are the same as when using them in greenfield development.

Hear from experienced practitioners about some of the latest open source technologies, problems and approaches shaping microservices.

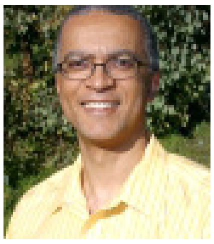
Learn some of the best practices (do's and don'ts) for using microservices effectively.

Understand some important considerations before using microservices, such as how they tie into (classic) distributed systems theory and practice.

Learn whether specific programming languages or technologies are recommended for developing with microservices.

Understand whether REST/HTTP should continue as the de facto standard for communication with and between microservices.

THE PANELISTS

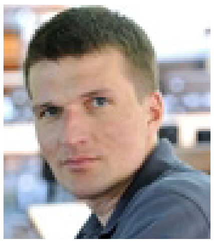


Chris Richardson is a developer and architect. He is a Java Champion and the author of POJOs in Action, which describes how to build enterprise Java applications with frameworks such as Spring and Hibernate. Richardson was also the founder of the original CloudFoundry.com. He consults with organizations to improve how they develop and deploy applications, and is working on his third startup. You can find Richardson on Twitter @crichardson and on Eventuate.

James Lewis studied Astrophysics in the 90's but got sick of programming in Fortran. As a member of the ThoughtWorks Technical Advisory Board, the group that creates the ThoughtWorks Technology Radar, he contributes to the industry adoption of open source and other tools, techniques, platforms and languages. For the last few years he has been working as a coding architect on projects built using microservices; exploring new patterns and ways of working as he goes.

Martijn Verburg is the CEO and co-founder of jClarity, a Machine Learning based Java/JVM performance analysis company. He is the co-leader of the London Java User Group (LJC), and leads the global Adopt a JSR and Adopt OpenJDK efforts to enable the community to contribute to Java standards and OpenJDK. He is a popular speaker at major conferences (JavaOne, JFokus, OSCON, Devoxx etc) where he is known for challenging the industry status quo as "the Diabolical Developer." Verburg was recently made a Java Champion in recognition for his contribution to the Java ecosystem.

Christian Posta (@christianposta) is a principal architect at Red Hat and well-known for being an author (Microservices for Java Developers, O'Reilly 2016), frequent blogger, speaker, open-source enthusiast and committer on Apache ActiveMQ, Apache Camel, Fabric8 and others. Posta has spent time at web-scale companies and now helps companies creating and deploying large-scale distributed architectures - many of what are now called Microservices based. He enjoys mentoring, training and leading teams to be successful with distributed systems concepts, microservices, devops, and cloud-native application design.



Adam Bien is a consultant and Java (SE/EE/FX) enthusiast who uses Java since JDK 1.0 and still enjoys writing Java code. Bien occasionally organizes Java EE / HTML5 / JavaScript workshops at Munich's airport.

InfoQ: We are a couple of years into the popularity of microservices; what important lessons have we learned in that time that perhaps weren't apparent at the start?

Chris Richardson: 'Microservices' is a terrible term. It places excessive emphasis on size and leads developers to create services that are too fine-grained, e.g. single REST endpoint per service. Not only that, but the term suggests that it makes sense to have **a** microservice. For example, I've heard, "we can do that with a microservice." I've also seen an increasing number of "Xyz microservice frameworks," which in reality have very little to do with the microservice architecture, e.g. they are simply web frameworks.

It is important to remember that the proper term is the "microservice architecture." It is an architectural style that structures a system as a set of collaborating services that are organized around business capabilities.

James Lewis: I think the first thing is that widespread adoption has led to a certain amount of semantic diffusion. Martin Fowler and I were fairly clear when we wrote the definition that all of the characteristics we mentioned contributed to the success of the companies using the style. I speak to a lot of big organisations who want to adopt the first characteristic, componentization via services, but who aren't at all keen on the organisational changes implied by the other characteristics. Specifically, Products not Projects, Organised around Business Capabilities and Decentralised Governance. Personally, I think that the organisational aspects of microservices are a key success factor for adoption.

Martijn Verburg: Oh there are so many! But I'll pick out some of my favourites:

- Service discovery - Solving both at development time and at runtime is much harder than people have realized. I've seen many cases where a team of developers is arguing about "where the message went next."
- Distributed tracing - of business logic / transactions is also very difficult to do in a light weight and unified manner. For example, how do you insert a trace that may follow a piece of business logic that travels through 10+ services, all of which are built with different technologies?
- Distributed architectures - Microservices based applications tend to lend themselves to distributed architectures, horizontal scaling and load balancing. This is a skill set that traditional monolith developers and sysadmins do not have and must learn. For example, how do you load balance Websocket connections which by their nature are two-way and 'permanent'?

Christian Posta: With respect to the microservices hype, I'm hoping we've learned that there are no utopian architectures; simply adopting buzzword technology doesn't equate to microservices, and the communication structures of your organization have more to do with the limitations or advantages or your services architecture than previously acknowledged.

I think it's also key that with a microservices-like architecture we've stretched deeper into a distributed systems theory and practices that we've studied, implemented and from which we've learned over the last 40 ►

years, and that very little is new. What's new is bringing this body of study and implementation to mainstream to solve new business problems.

Adam Bien: I spend most of my time in Java EE projects. The availability of Java EE 6 in 2009 became the main driver towards microservice-like architectures. We were able to package the pure business logic in a WAR and ship it [Java EE 6 Kills The WAR Bloat]. Back in 2009 we called our projects "shared nothing architecture." The term "microservices" had not arrived yet.

Monitoring and performing stress tests was a hard sell back then. With microservices, it is more acceptable now to focus on stress tests, system tests and monitoring of the essential use cases.

Prior to the microservice hype, most of the projects focused on the implementation of unit tests only to achieve high (but meaningless) code coverage. This begins slowly to change.

The biggest difference between the early Java EE 6 projects and the current development is the nature of communication protocols. In the 2009 timeframe we mostly relied on binary RPC protocols; right now JAX-RS (REST/HTTP) and WebSockets are the new default.

InfoQ: At the start, the so-called "unicorns" were popularising microservices; do you think this is still the case and if not, then who are the poster children at the moment?

Richardson: Yes. It feels like the majority of the conference talks on the microservice architecture are "cool microservice architecture topic at Netflix/Uber/Slack/Twitter…." On the one hand, these talks are incredibly useful and have helped evangelize the architecture. On the other hand, that has led some developers to think that the microservice architecture is a way to address application scaling issues, where in reality it is a way to tackle complexity. In general, it would be great to hear from mainstream companies about how they are using microservices.

Lewis: I think they are still at the cutting edge, yes. It's where a lot of advances are being made in the infrastructure supporting microservices. It is not a "101" architectural style, and some of the benefits manifest most obviously when you are operating at extremes of scale.

Verburg: They still are. Companies like the BBC, Netflix, Twitter, Amazon et al are all microservice based because they had the horizontal scalability requirement thumped firmly on their desk. But *this* is the major question that most IT organisations fail to address when they blindly jump on the bandwagon. "Do we actually need microservices? Does our scale require it? Does our business logic require it?" The answer for many organisations should actually be a resounding "no."

Posta: To paraphrase Dr. Branden Williams, "there are no unicorns or horses anymore, just thoroughbreds and horses heading to the glue factory." The internet companies may have been the vanguard showing the way, but I think there are good examples in the traditional enterprise space (FSI, Manufacturing, Retail, etc.) demonstrating the ability to move fast and innovate using technology.

Bien: At the start no one knew what "micro" actually meant. There was a pointless debate about a typical size of service. In Java EE a microservice is a Thin WAR created by a one-pizza team -- two-pizza teams are already too large :-)

In my eyes, the poster children are many enterprise projects based on pragmatic microservices. Unicorns come and go. It is really hard to estimate their success, if they barely survive the first year. Enterprise projects have to last longer.

InfoQ: Some vendors push microservices for greenfield development, whereas others tend to focus on brownfield and decomposing monolithic applications; do you think the same principles apply to architects and developers for each approach?

Richardson: The microservice architecture is potentially applicable to both greenfield and brownfield applications that are (or will be) large and complex. What matters is the context within which you are developing the application. For example, if you are a startup still trying to figure out your business model, then it is possible that you will be able to pivot more rapidly with a monolithic architecture.

I think that many (perhaps the majority) of business critical applications are large, complex monoliths. The business is in monolithic hell and unable to innovate rapidly. The solution is to incrementally refactor to a microservice architecture.

Lewis: Personally I've been involved with teams who have done both. For brownfield it's often a nice approach since it gives you many more options or seams to start containing a previous system. For greenfield, the approach I favour is to consider the functional and cross-functional requirements, and the context within which the system should run. Sometimes that means a microservice architecture, sometimes not.

Verburg: Same principles but a lot of different compromises :-). Decomposing a monolith is an admirable and satisfying goal to complete, but for a good chunk of that application's life, it's going to be a microservice and monolith hybrid (some of us call this the software incarnation of Cthulhu). For example, microservice purists working on a database centric monolith would have put aside their principles and "do that message passing through the stored procedure in the monolith database" for a period of time until they refactored everything out.

Integration test writing and maintenance becomes very important here.

Posta: The principles are similar insofar as both approaches try to find the right boundaries to affect the speed of development cycles. You may have pockets of greenfield development but the harder part is finding the right seams for existing brownfield systems to expand, speed up, innovate, etc and do so safely.

Bien: The vast majority of all client inquiries in 2016 were about the introduction of microservices with the big hope of increased maintainability and cost savings. The problem was never the lack of distribution, rather cargo cult practices and unnecessary pattern implementation.

Splitting a bloated monolith into smaller overcomplicated monoliths will only make the situation worse. In brownfield projects it is crucial to remove the cargo cult based patterns first, and re-learn the domain concepts. Splitting a lean monolith into independent units becomes a fully optional task.

In greenfield projects you should completely focus on business logic and stay with a monolith in the first iterations. Introduce a microservice only if you can clearly explain the benefits. Shipping a lean monolith is still the easiest possible approach.

Both greenfield as well as brownfield share the laser focus on business logic.

InfoQ: What are your top five do's and don'ts where microservices are concerned?

Richardson: The most important thing to remember is that the microservice architecture is not a silver bullet. You need to carefully evaluate the trade-offs to determine whether it is appropriate for your application.

Lewis:

Do:

1. Monitor, monitor, monitor.

2. Get good at deploying services independently.

3. Prefer rapid remediation and canary deploys over integration testing.

4. Prefer choreography over orchestration.

5. Limit your call tree. The more services in the graph, the more difficult it is to stay available.

Don't:

1. Don't suddenly build 500 services - start with a reasonable number that is supportable with your current infrastructure.

2. Don't think they are a magic bullet. You need to understand some non-trivial distributed computing concepts to get good at building them.

3. Don't fall for vendor snake-oil - that's why SOA originally died a death.

4. Don't forget the bit about replaceability. They should be small enough to be thrown away.

5. DON'T DO DISTRIBUTED TRANSACTIONS.

6. Did I mention don't do distributed transactions?

Verburg:

Do:

1. Make sure your team is working in an 'a' Agile manner.

2. Make sure your team has a DevOps culture.

3. Build three prototype services that communicate with each other and figure out how to do all of the non functional requirements like security, service discovery, health monitoring, back pressure, ▶

failover etc., *before* you go and build the rest.

4. Let the engineers pick the right technology for each service; this is a major advantage.

5. Care more about your Integration tests.

Don't:

1. Start using them because Netflix is.

2. Forget about data consistency. "Oh yeah, our microservice architecture doesn't do ACID transactions, sorry we lost your money" is not acceptable.

3. Ignore the infrastructure requirements, even for the 'developer desktop.' Get developers mimicking the real PRD architecture as soon as possible.

4. Forget about naming - once you've released a public API you are stuck with it. Don't forget to version your APIs as well for that matter.

5. Throw away the years of developer experience and business logic already written. It's an evolution, not a new paradigm.

Posta:

Do:

1. Measure your adoption of a microservices architecture and use that as a guide post. Microservices is about speed, so measure how quickly your teams can make changes and deploy without impacting other services. Things like #s of builds, #s of deployments, # of bugs introduced, time it takes to approve a deployment, mean time to recovery, etc.

2. Do establish proper feedback loops for your feature teams. It does no good to make changes to your systems/services without knowing what the effect of that change will be. Put developers and feature teams as close to their customer (or even in their customer's shoes) so they see the pain directly from the systems the teams build.

3. Do pay attention to data. Data is the lifeblood of a company. When building services, pay attention to use case boundaries, transaction boundaries, consistency issues, and data processing (stream, storage, etc.).

4. Build microservices/feature teams with autonomy, responsibility, and freedom built in. Build the tooling, APIs, and infrastructure for them to self-service.

5. Build your services with instrumentation, metric collection, debuggability, and testing as a first class citizen, not as an afterthought.

Don't:

1. Don't just copy the parts of X unicorn company you see just because they seem successful; figure out the principles that drove that company and use that as a guide. Case in point. Simply adopting Netflix OSS technology will not make you Netflix.

2. Don't approach microservices as a way to cut costs; Microservices is about enabling innovation and business outcomes through

the microservice architecture is not a silver bullet. You need to carefully evaluate the trade-offs to determine whether it is appropriate for your application.

- Chris Richardson

technology, not as a way to minimize operating costs, like traditional IT has been treated for decades.

3. Don't break down systems arbitrarily small just for the sake of breaking them down. You run the risk of creating a non-scalable, highly inefficient distributed monolith and strangle yourself with transactions.

4. Don't ignore the fallacies of distributed systems and the challenges of integration.

5. If you have trouble with CI/CD, APIs, DevOps, self-service platforms, or self-service teams, then don't force a microservices architecture; get the principles, practices, and organizational learning systems down first. Doing microservices isn't the goal; fast moving, innovative teams is the goal. Get the foundational pieces in place first.

Bien:

Do:

1. Evaluate container technology for deployment. The advantages are too big to be ignored.

2. Focus on business logic.

3. Monitor the essential use cases / key performance indicators.

4. Focus on system tests, not unit tests.

5. Automate everything, CI / CD are a no brainer.

Don't:

1. Don't copy the practices of Netflix, Twitter, Facebook or Google unless you have their scale / requirements.

2. Do not ignore slow turnaround cycles. Deploying a Fat-WAR takes longer than a thin one. Productivity really matters.

3. Do not even attempt to coordinate (XA) transactions between microservices.

4. Don't start your project with downloading the internet. Less is more. Each dependency makes your deployment slower and requires security audits and bug-fix maintenance. There are no "free" dependencies.

5. Don't distribute (or only distribute with obvious advantages). A lean monolith could become the best possible choice.

InfoQ: HTTP or REST/HTTP is often seen as the de facto standard for communication between microservices, yet we've recently seen a lot of groups talking about asynchronous, message-oriented approaches instead; what do you think?

Richardson: Yes. The de facto IPC mechanism these days is HTTP/REST. It is familiar and easy to use. The drawback is that it introduces a temporal coupling between the client and service. Whether or not that is a problem depends on the context. For example, when writing code that handles a query request that aggregates data from multiple services, then it might be ok. If on the other hand, you are handling a command request that updates data, you should use asynchronous messaging to implement eventually consistent transactions, a.k.a sagas.

Lewis: Smarts in the endpoints does not strictly imply REST/HTTP; for me it was about these groups of small collaborating services that encapsulated their own logic, communicating via a *uniform interface*. I've been involved in teams that have used lightweight messaging and RESTful approaches to supplying that uniform interface and both have been successful. Still, the most important thing is to choose the appropriate patterns for the problem at hand. If you have a business process that lends itself to asynchronicity then you should use an asynchronous integration technique; conversely if your problem is more amenable to map-reduce aggregation over a number of discrete services then you should probably use some form of reactive approach. Once again I'm reminded that we often try and take a reductive approach when actually it's really about thinking for yourself.

Verburg: I think both approaches will be used with asynchronous messaging becoming more popular over time. At jClarity for example, we have an asynchronous message-oriented approach but also offer a REST/HTTP(S) API for easier public consumption.

Posta: I think as you scale out systems like we talk about with microservices, they tend to exhibit characteristics we see in other Complex Adaptive Systems (stock markets, ant colonies, communities), to wit: autonomous agents, independent decision making, learning/adaptation driven by feedback, nonlinear interaction, etc. In systems like that, events, message passing, and time are all critical ►

enablers that tend to look like the "async" model. IMHO making time the focal point between these systems (as well as the fact our communication channels may not be reliable) force us to deal with reality up front and make for a model we know scales in other applications.

Bien: In my projects HTTP / REST (always JAX-RS) was good enough for the realization of the vast majority of all use cases. Sometimes we also introduced WebSockets as an asynchronous, peer-2-peer, messaging protocol. These cases were more of an exception than a rule.

InfoQ: Given that microservices architectures are more distributed systems-oriented than some developers have been used to in the past, where should a developer new to microservices, and perhaps distributed systems, start?

Richardson: A developer will use many of the familiar frameworks and libraries to develop an individual service, so not much has changed there. However, a consequence of applying the microservice architecture is that some things such as transaction management and querying need to be done differently. A good starting point to learn about those issues and how to address them, are my recent Infoq articles (re-published in this eMag) as well as my website.

The essence of the microservice architecture is the idea of organizing services around business capabilities or subdomains (or bounded contexts). I'd recommending reading Eric Evans' book "Domain Driven Design", since both aggregates and his ideas around strategic design are central to the microservice architecture.

Lewis: I think Sam Newman did an excellent job with his book "Building Microservices", so I would start there. For background reading, I would recommend "Domain Driven Design" by Eric Evans, "REST in Practice" by Webber, Robinson and Parastatidis, "Enterprise Integration Patterns" by Hohpe and Woolf, and "Release It!" by Michael Nygard. For a peek at the philosophy behind building systems composed of small things, I heartily recommend "The Art of UNIX Programming" by Eric Raymond. Further resources can be found on Martin Fowler's site too.

Verburg: If they're a traditional Java enterprise developer then the new Microprofile.io community is a great place to start. Regardless of where they start, they *have* to understand what it takes to set up the infrastructure. Start by renting a few Linux boxes (and/or taking the plunge into Docker) and building a "hello world" service that talks to a "Hi Back!" service on the other 'machine.' You should experiment with HTTP(S), certificates, load balancing, IP tables, having a distributed data store (like MongoDB) and so forth.

The application code is now truly the easy part of this new world. The hard part is the plumbing.

Posta: This is such a great question. The last 40 years of distributed systems computing research and practice is the core backbone of implementing a microservices architecture. Understanding why your ACID database is so good for you and the challenges you'll need to overcome when you distribute things is paramount. There are lots of good papers on this. Ones by Jim Gray, Peter Bailis, Alan Fekete, Pat Helland, Leslie Lamport etc are my favorites. My background is in integration and messaging and I've also found those to be a hugely valuable body of knowledge to help set the foundational concepts.

Bien: Just focus on domain concepts, target domain and the users. Keep the signal to noise ratio as high as possible. E.g. the best Java EE projects only contain business logic with a few annotations, without any additional cruft, patterns or indirections.

Forget about all modularization attempts from the past. Keep your code simple. Thin WARs are the ultimate module.

Be paranoid and assume that the whole infrastructure around your service can and will fail. Test the behaviour in failure case. Provide the simplest possible solution (should be classes, not frameworks).

InfoQ: Are there particular languages or technologies you'd recommend for developing with microservices? If so, why? Any that you'd avoid? If so, why?

Richardson: The short answer is that the microservice architecture is independent of languages and frameworks. The longer answer is that some languages might have more microservice chassis frameworks that help with building distributed applications than others. For example, Java developers can use Eureka/Ribbon, possibly via Spring Cloud, for client-side service discovery, and the Hystrix circuit breaker library. On the other

Just focus on domain concepts, target domain and the users. Keep the signal to noise ratio as high as possible.

- Adam Bien

hand, there are good arguments for using a deployment platform that provides server-side discovery so that the developer doesn't have to worry about it.

Lewis: I think one of the good things to come out of the last few years has been a refocus on small and simple. I'm most comfortable talking about the Java ecosystem so I would call out Dropwizard and Spring Boot as fairly good places to start. I like the philosophy of Dropwizard in particular. I seem to remember the tagline was "a bunch of libraries that don't suck much" and that is pretty much what you want, over heavy frameworks certainly. Outside that ecosystem, I know a fair few teams that are having a lot of success with Go Lang and with Elixir. I can't comment on the JavaScript ecosystem since, and to misquote Dave Thomas of Pragmatic Programmers fame, "I haven't checked my twitter feed this morning."

Verburg:

- Microprofile.io, Vert.x, Spring Boot, JHipster for Java developers. At jClarity we use Vert.x which is an amazing (JVM based) polyglot language library for developing Microservices applications. Can't recommend it enough.
- Akka for Scala developers.
- NodeJS for JavaScript developers.

Posta: Use whatever you're comfortable with that will help you go fast. In my mind, Go, Java/.NET, and NodeJS are the most often used languages for these types of services.

In the enterprise, if you're trying to modernize your Java services, technologies like linux containers and "micro frameworks" like Dropwizard, WildFly Swarm and Spring Boot are helpful. If using Domain Driven Design, event frameworks and the reactive frameworks like Vert.x are awesome. Other cloud scale technologies for both the platform and the application layer include Kubernetes, Hystrix, and Envoy to help solve difficult distributed systems issues.

Bien: Java is 20 years old, mature, and comes with unbeatable tooling and monitoring capabilities. At the very beginning, Java already incorporated microservice concepts with the Jini / JXTA frameworks mixed with no-SQL databases like e.g. JavaSpaces. As often -- Java was just 15 years too early. The market was not ready for the technology back then. However, all the design principles from 1999 still do apply today. We don't have re-invent the wheel.

I frequently suggested Java EE (the full profile) application servers, (Payara, TomEE and Wildfly) as a microservice platform for startups / greenfield projects tipi.camp, artem, dreamit, next farming (...). We started with the realization of business logic in the very first hour without wasting any time for discussion. We were productive from day one. The developers were positively surprised about the development efficiency, memory footprint and built-in features of modern application servers. Developers were stunned at how much you can achieve with stock Java SE / EE without any external library. I got only positive feedback so far.

Docker is another key ingredient to success. Coupled with Java EE it is a dream team. ►

InfoQ: Where do you think we'll be two years from now with microservices?

Richardson: Who knows!? I first gave a talk about what is now known as the microservice architecture in April 2012. The core idea of the microservice architecture has remained unchanged. Since then I've since seismic shifts in technology: the rise of Docker and AWS Lambda for example. Consequently, it is difficult to make predictions. Having said that, I expect (or hope) that the microservice architecture will traverse the Gartner hype curve and will reach the plateau of productivity.

Lewis: My prediction is that some companies will have made a lot of money by adopting them and there will be a number of organisations that have tried, but have not understood the implied organisational changes and who will have gotten into a terrible mess. Also, I hope we get some more really cool tooling around service visualisation, request tracing and more intelligent failure detection. That would be nice.

Verburg: In terms of the Hype Cycle curve, we'll have crested the Hype Wave and fallen into the Trough of Disillusionment; some organisations will be heading towards the slope of enlightenment :-)

Posta: I'd like to reframe the question if you don't mind: given we're about two years into the microservices hype, do you think in two more years we'll be at the same point we were with SOA four years into? Yes :)

The difference this time is that the internet companies and startups are significantly disrupting traditional enterprises with technology as the main weapon (the game has changed). So with regards to the technology hype, it'll be no different than SOA, but enterprises that don't adopt the many principles that make up DevOps, Agile, and Microservices will lose to those that do.

Bien: During a Java User Group meeting a developer proudly stated: "My 4-devs team ships 35 microservices." Another day a consultant approached me to present 70 JVM instances running on his notebook.

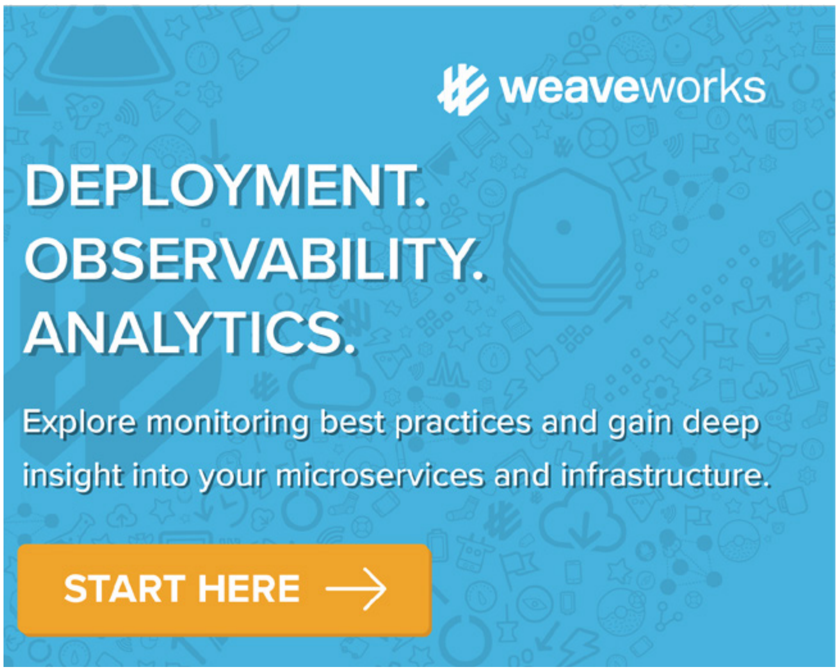
Both were surprised by my question: "Why are you doing this? What is the added value of your services?"

I expect the first exaggerated microservice projects to be more expensive as estimated. Such projects may cause the first articles / conference talks about microservice bloat or low developer productivity to appear on the horizon.

Bad press usually leads to another extreme. I'm not sure whether we get Macroservices or Nanoservices. I'm pretty sure we get another old concept "sold" as new with another funky name.

Conclusion

In this virtual panel article, we learned about the current state of the art with microservices, experience-driven best practices and some predictions for where things may be heading in the next few years. We were given various recommendations from the panelists about technologies and approaches that can help with using microservices successfully, but also that you should never lose sight of the fact that a microservices architecture is inherently a distributed system and therefore decades of theory and practice may be hiding beneath the surface waiting to pounce on the unsuspecting developer. We also heard their thoughts on REST/HTTP as a means of communication with and between microservices as compared to other mechanisms such as asynchronous, message-oriented implementations.


weaveworks
DEPLOYMENT.
OBSERVABILITY.
ANALYTICS.
Explore monitoring best practices and gain deep insight into your microservices and infrastructure.
START HERE

Evolution of Business Logic from Monoliths through Microservices, to Functions

Adrian Cockcroft has had a long career working at the leading edge of technology. He's always been fascinated by what comes next, and he writes and speaks extensively on a range of subjects. At Battery, he advises the firm and its portfolio companies about technology issues and also assists with deal sourcing and due diligence. Before joining Battery, Adrian helped lead Netflix's migration to a large scale, highly available public-cloud architecture and the open sourcing of the cloud-native NetflixOSS platform.

Originally posted on A Cloud Guru on Feb 16, 2017. Reprinted with permission.

Underlying technology advancements are creating a shift to event driven functions and radical improvements in time to value

The whole point of running application software is to deliver business value of some sort. That business value is delivered by creating business logic and operating it so it can provide a service to some users. The time between creating business logic and providing service to users with that logic is the time to value. The cost of providing that value is the cost of creation plus the cost of delivery.

In the past, costs were high and efficiency concerns dominated, with high time to value regarded as the normal state of affairs. Today, when organizations measure and optimize their activities, time to value is becoming a dominant metric, driven by competitive pressures, enabled by advances in technology, and by reductions in cost. Put another way, to increase return on investment you need to find ways to increase the return, start returning value earlier, or reduce the investment. When costs dominate, that's where the focus is, but as costs reduce and software impact increases, the focus flips towards getting the return earlier.

As technology has progressed over the last decade, we've seen an evolution from monolithic applications to microservices and are now seeing the rise of serverless event driven functions, led by AWS Lambda. What factors have driven this evolution? Low latency messaging enabled the ▶

KEY TAKEAWAYS

Business software is valued based on the timeframe for a return on investment. Modern software development processes are altering the equation by reducing costs and increasing the impact of software.

Over the past ten years, we've seen the best architecture shift from monoliths, to microservices, to event-driven functions.

Costs associated with monoliths were dealt with through process automation, starting with server provisioning, leading to containers, and eventually to serverless solutions.

Radically faster networks and NoSQL databases are key enablers in the move away from monoliths.

Changes in "people and processes," including DevOps and cellular "two-pizza teams", have also contributed to more efficient delivery.

move from monoliths to microservices, and low latency provisioning enabled the move to Lambda.

To start with, ten years ago a monolithic application was the best way to deliver business logic, for the time constraints. Those constraints changed, and about five years ago the best option shifted to microservices. New applications began to be built on a microservices architecture, and over the last few years, tooling and development practices changed to support microservices. Today, another shift is taking place, to event driven functions, as the underlying constraints have changed, costs have reduced, and radical improvements in time to value are possible.

In what follows, we'll look at different dimensions of change in detail: delivery technology, hardware capabilities, and organizational practices, and see how they have combined to drive this evolution.

At the start of this journey, the cost of delivery dominated. It took a long time to procure, configure and deploy hardware, and software installations were hand crafted projects in their own right. To optimize delivery the best practice was to amortize this high cost over a large amount of business logic in each release, and to release relatively infrequently, with a time to value measured in months for many organizations. Given long lead times for infrastructure changes, it was necessary to pre-provision extra capacity in advance and this led to very low average utilization.

The first steps to reduce cost of delivery focused on process automation. Many organizations developed custom scripts to deploy new hardware, and to install and update applications. Eventually common frameworks like Puppet and Chef became popular, and "infrastructure as code" sped up delivery of updates. The DevOps movement began when operations teams adopted agile software development practices and worked closely with developers to reduce time to value from months to days.

Scripts can change what's already there, but fast growing businesses or those with unpredictable workloads struggled to provision new capacity quickly. The introduction of self service API calls to automatically provision cloud capacity using Amazon EC2 solved this problem. When developers got the ability to directly automate many operations tasks using web services, a second wave of DevOps occurred. Operations teams built and ran highly automated API driven platforms on top of cloud services, providing self service deployments and autoscaled capacity to development teams. The ability to deploy capacity just-in-time, and pay by the hour for what was actually needed, allowed far higher average utilization, and automatically handled unexpected spikes in workloads.

Another wave of optimization arrived when docker made containers easy enough for everyone to use. Docker containers provide a convenient bundled package format that includes a fixed set of dependencies, a runtime that gives more isolation than processes, but less than a virtual machine instance, startup times measured in seconds, and a substantial saving in memory footprint. By packing many containers onto an instance, and rounding off run times to minutes or seconds instead of hours, even higher utilization is possible. Container based continuous delivery tooling also sped up the work of developers and reduced time to value.

When there's a reasonably predictable amount of work coming in, containers can be run at high utilization levels, however many workloads are spiky or drop to zero for extended periods. For example, applications used in the workplace may only be active for 40 of the 168 hours in a week. To maintain high availability, it's usual to spread application instances over three availability zones, and even to require more than one instance per zone. The minimum footprint for a service is thus six instances. If we want to scale down to zero, we need a way to fire up part of an application when an event happens, and shut it down when it's done. This is a key part of the AWS Lambda functionality, and it transforms spiky and low usage workloads to effectively 100% utilization by only charging for the capacity that is being used, in 0.1 second increments, and scales from zero to very high capacity as needed. There's no need to think about or provision servers, and that's why this is often called the serverless pattern.

Advances in delivery technology provide stepping stones for improvements in time to value, but there are other underlying changes that have caused a series of transitions in best practices over the last decade.

The optimal size for a bundle of business logic depends upon the relative costs in both dollars and access time of CPU, network, memory and disk resources, combined with the latency goal for the service.

For the common case of human end users waiting for some business logic to provide a service, the total service time requirement hasn't changed much. Perception and expectations haven't changed as much as the underlying technology has over the last decade or so.

CPU speed has increased fairly slowly over the last decade, as the clock rate hit a wall at a few GHz, however on chip caches are much larger, and the number of cores increased instead. Memory speed and size have also made relatively slow progress.

Networks are now radically faster, common deployments have moved from 1GBit to 10GBit and now 25GBit (as explained by James Hamilton in his AWS re:Invent 2016 keynote), and software protocols are far more efficient. When common practice was sending XML payloads over 1GBit networks, the communication overhead constrained business logic to be co-located in large monolithic services, directly connected to databases. A decade later, encodings that are at least an order of magnitude more efficient over 25Gbit networks mean that the cost of communication is reduced by more than two orders ▶

of magnitude. In other words, it's possible to send 100 to 1000 messages between services in the same amount of time as communicating and processing one message would take a decade ago. This is a key enabler for the move away from monolithic applications.

Storage and databases have also gone through a revolution over the last decade. Monolithic applications map their business logic to transactions against complex relational database (RDBMS) schemas, that link together all the tables, and allow coordinated atomic updates. A decade ago the best practice was to implement a small number of large centralized relational databases connected via storage area networks to expensive disk arrays using magnetic disk, fronted by large caches.

Today, cached magnetic disks have been replaced by solid state disks. The difference is that reads move from slow, expensive and unpredictable—as cache hit rate varies, to consistently fast and almost unlimited. Writes and updates move from being fast for cached disks to unpredictable for solid state disks, due to wear leveling algorithms and other effects.

New "NoSQL" database architectures have become popular for several reasons, but the differences that concern us here are that they have simple schema models and take advantage of the characteristics of solid state storage. Simple schemas force separation of the tables of data that would be linked together in the same relational database, into multiple independent NoSQL databases, driving decentralization of the business logic. The Amazon DynamoDB datastore service was designed from the beginning to run only on solid state disk, providing extremely consistent low latency for requests. Apache Cassandra's storage model generates a large number of random reads, and does infrequent large writes with no updates, which is ideally suited to solid state disks. Compared to relational databases, NoSQL databases provide simple but extremely cost effective, highly available and scalable databases with very low latency. The growth in popularity of NoSQL databases is another key enabler for the move away from monolithic schemas and monolithic applications. The remaining relational core schemas are cleaned up, easier to scale and are being migrated to services such as Amazon's RDS and Aurora.

It's common to talk about "people, process and technology" when we look at changes in IT. We've just seen how technology has taken utilization and speed of deployment to the limit with AWS Lambda, effectively 100% utilization for deployments in a fraction of a second. It's also made it efficient to break the monolithic code base into hundreds of microservices and functions, and denormalized the monolithic RDBMS into many simple scalable and highly available NoSQL and relational data stores.

There have also been huge changes in "people and process" over the last decade. Let's consider a hypothetical monolith built by 100 developers working together. To coordinate, manage test and deliver updates to this monolith every few months, it's common to have more people running the process than writing ▶

Lambda based applications are constructed from individual event driven functions that are almost entirely business logic, and there's much less boilerplate and platform code to manage.

the code; twice as many project managers, testers, DBA's, operators etc. organized in silos, driven by tickets, and a management hierarchy demanding that everyone write weekly reports and attend lots of status meetings, as well as find time to code the actual business logic!

The combination of DevOps practices, microservices architectures, and cloud deployments went hand in hand with continuous delivery processes, cellular based "two pizza team" organizations, and a big reduction in tickets, meetings and management overhead. Small groups of developers and product managers independently code, test and deploy their own microservices whenever they need to. The ratio of developers to overhead reverses, with 100 developers to 50 managers. Each developer is spending less time in meetings and waiting for tickets, getting twice as much done with a hundred times better time to value. A common shorthand for this change is a move from project to product. A large number of project managers are replaced with far fewer product managers. In my somewhat contrived example, 150 people are producing twice the output that 300 people used to. Double the return a hundred times sooner, on half the investment. Many organizations have been making this kind of transition, and there are real examples of similar improvements.

Lambda based applications are constructed from individual event driven functions that are almost entirely business logic, and there's much less boilerplate and platform code to manage. It's early days, but this appears to be driving another radical change. Small teams of developers are building production ready applications from scratch in just a few days. They are using short simple functions and events to glue together robust API driven data stores and services. The finished applications are already highly available and scalable, high utilization, low cost and fast to deploy.

As an analogy, think how long it would take to make a model house starting with a ball of clay, compared to a pile of Lego bricks. Given enough time you could make almost anything from the clay; it's expressive, creative, and there's even an anti-pattern for monolithic applications called the "big ball of mud". The Lego bricks fit together to make a constrained, blocky model house, that is also very easy to extend and modify, in a tiny fraction of the time. In addition, there are other bricks somewhat like Lego bricks, but they aren't popular enough to matter, and any kind of standard brick based system will be much faster than custom formed clay.

If an order of magnitude increase in developer productivity is possible, then my example of 100 developer monolith could be rewritten from scratch and replaced by a team of ten developers in a few weeks. Even if you doubt that this would work, it's a cheap experiment to try it out. The invocation latency for event driven functions is one of the key limitations that constrains complex applications, but over time those latencies are reducing.

The real point I'm making is that the ROI threshold for whether existing monolithic applications should be moved unchanged into the cloud or rewritten depends a lot on how much work it is to rewrite them. A typical datacenter to cloud migration would pick out the highly scaled and high rate of change applications to re-write from monoliths to microservices, and forklift the small or frozen applications intact. I think that AWS Lambda changes the equation, is likely to be the default way new and experimental applications are built, and also makes it worth looking at doing a lot more re-writes.

I'm very interested in your experiences, so please let me know how you see time to value evolving in your environments. ■

PREVIOUS ISSUES