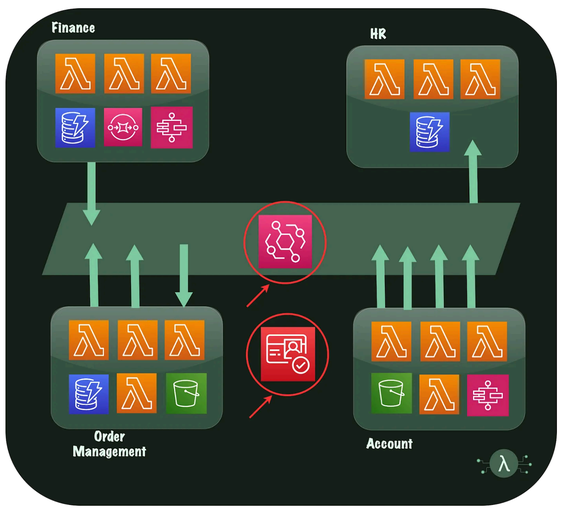
Service Ports: Finding a Loosely Coupled Utopia with Event-Driven Serverless

Ben Ellerby · Follow
Published in Serverless Transformation · 7 min read · Jan 10, 2022



When we build Event-Driven Serverless architectures using Domain-Driven Design, we end up with a set of services clearly split by business function and communicating asynchronously over an event channel (e.g. Amazon EventBridge). These architectures bring many advantages: loose coupling, independent deployability, testability, and reduced complexity, to name a few. Yet, no matter how elegant our modelling of the domain, there are always some cross-cutting infrastructure dependencies (e.g. the event channel itself) that can't be eliminated.

How we deal with them determines if we get a loosely-coupled utopia, or a sprawling tangle of semi-independent services. When you've put in 90% of the effort, don't let this be the final stumbling block.

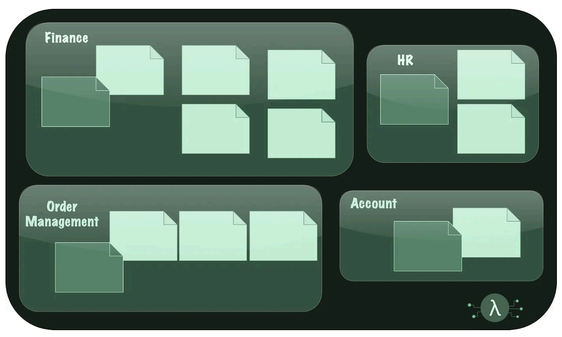
In this article we'll see how clearly defined Service ports resolve the final 10% of infrastructure dependencies and bring loose coupling, independent deployment/testing and increase development velocity.

How the problem arises

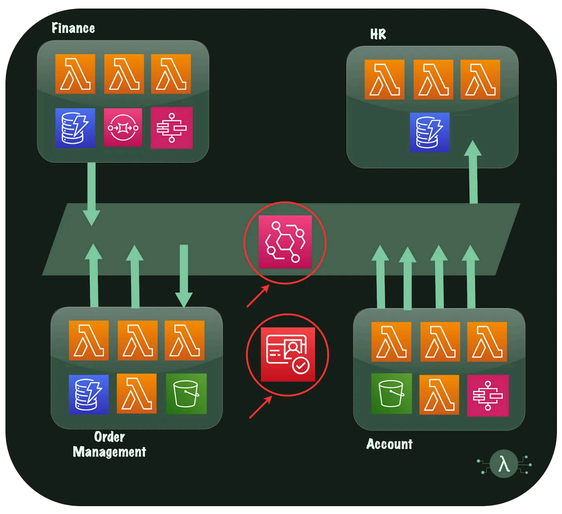
We've discussed previously how Amazon EventBridge can be combined with strong Domain-Driven Design to create such architectures (see EventBridge Storming). When we build a collection of Services (or "microservices") in such a way there are still some cross-cutting global dependencies. These dependencies get in the way of developing services independently, make integration testing difficult and can create complex code in our infrastructure definition. In addition, when such dependencies are not made explicit, the deployment order of Services can become a mystery at best, and a set of race conditions at worst.

To make this more tangible, let's imagine we conducted an EventBridge Storming session to understand an example problem domain of an e-commerce website by gathering all the business events of the system into Bounded Contexts. From this session we would understand the business domain, the events that belong to it, and how they can be grouped together.

From such a session we could derive the following services, grouping together the domain events and creating service boundaries with loose coupling.



A high level view of the architecture would look as follows:



Notice two classic shared infrastructure dependencies are the EventBus itself, and the Cognito User Pool. Reference to these in the infrastructure is simple, it's an ARN. But requiring these dependencies to be deployed (with everything else in the stack they belong to) makes independent development and testing complex. In addition, these references can also be hidden in the weeds of the infrastructure as code files.

Such cross-cutting infrastructure dependencies need to be intentional, and to be designed for and made explicit to the developers. If not, this will lead to several problems:

"The Source of Existence Problem". We could keep this definition of the dependencies in the Finance Service and reference it in the HR Service. When it's 2 Services it's fine, but what about 10 Services all needing this resource.
In addition, we've created some knock-on issues. The Finance Service now needs to be deployed before the HR service. In production this seems manageable, it will happen the first time; but what about Dev, Staging and UAT? Also, with Serverless, one amazing benefit is ephemeral stacks (see Serverless Flow) — how will this be achieved?

"The New Joiner Problem": Independent development of the HR service is now no longer possible. Any new developer to the company will have had to have deployed their own version of the Finance Service before they can develop on the HR service — or they depend on the staging version and

probably not be able to intuitively understand why. We've created a big barrier to development, and increased the cognitive load of what should be a simple and isolated service.

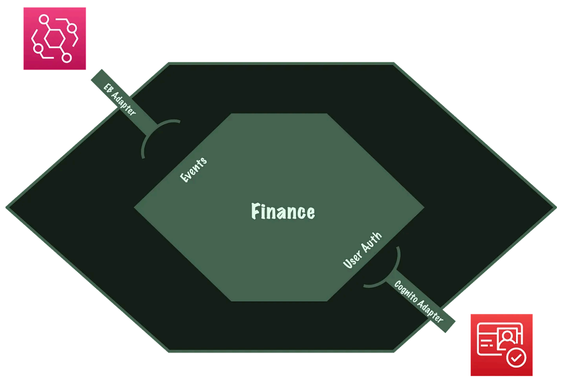
"The Testing Independence Problem": If you can't test independently, you should not deploy independently. Independent testability of the HR service is now not possible; again we'll need to deploy an independent Finance Service, or risk the uncontrolled environment of the shared staging resource — even though these are all Serverless Services with pay-per-use billing and rapid deployment. There is a huge cost to mocking, especially when, with our Serverless approach, we should be able to have an almost unlimited number of independent development/testing environments per service.

Hexagonal Architecture To The Rescue

Ports and Adapters, or hexagonal architecture, is an approach that tries to make dependencies interchangeable. The interchangeable aspect allows development and testing to be simplified. In addition, clear specification of ports ensures that dependencies are explicitly defined. Components list their connections as ports, with an API specification of the protocol they expect. One or more adapters can then implement this protocol.

A classic example is a UI port implemented as either a GUI for users or a CLI for development ease. These are typically at the application code level, but we can bring the concept up to higher level architecture dependencies.

If we look at our example above, the clear dependencies are an EventBus and a UserPool. We could take things further and make these more abstract dependencies, but for now let's just expect an EventBridge EventBus (any EventBridge EventBus) and a Cognito UserPool (any Cognito UserPool).



A simple representation of these ports can be shown in a Serverless Framework IaC yaml file:

```
provider:
  portsStage: ${opt:portsStage}
custom:
  ports:
    eventBridge:
      staging: staging-application-bridge
      production: production-application-bridge
  eventBridge: ${self:custom.ports.eventBridge.${self:provider.portsStage}}
```

Notice here that it's environment dependent, and there is a clear switch case on the environment. This can be easily overwritten to support a temporary CI stack as per Serverless flow, and switched out manually during local development as needed.

Creating this explicit ports section makes the interfaces clearer and should be further explained in a consistent README section for search service of the architecture.

PORT	Type	Environments	Adaptors	Manually Create
eventBridge	EventBridge Event Bus ARN	Staging, production	NA	creating-dev-eventBridge.MD
cognitoPool	Cognito User Pool	Staging, production	NA	creating-dev-cognitoPool.MD

Where Should Cross-Cutting Infrastructure Dependencies Live?

Going back to our example of the Finance Service. This service relies on an EventBridge Bus and a Cognito User Pool. But where should these dependencies live?

If they live with a Service, e.g. the Finance Service, they will be deployed with it — but is that correct? Well no, just because there is a change to the Finance Service, it does not mean there should be a deployment of the EventBridge or Cognito User Pool.

Our Ports have mitigated some of the interdependence that would be created, but that does not mean we should arbitrarily pick a host Service. Instead we should see these as Services, very simple services due to the abstraction our Cloud provider is giving us over Event Bussing and User Authentication.

It's tempting to create a "global" or "central" or "infra" repo and dump any cross cutting dependency into this. **Avoid this temptation**.
It will create a dumping ground for anything needed by > 1 Service and will become a mess quickly. Instead create very small Services for each cross-cutting dependency. These will simply contain the Infrastructure as Code configuration, the CI/CD Pipeline and any integration tests.

Auto-Scaffolding Dev and Test "Adapters"

Any developer asked to build a basic feature on this stack can now deploy an EventBus and UserPool manually or with a script, without requiring all the other stacks.

When it comes to the pipeline, we can make the environment selection and the resulting ports explicit. This can be in a dedicated part of the CI/CD pipeline as a script, or even better as a conditional block in the Infrastructure as Code template (**if dev, if test, if prod**).

Further to this, we can have conditional testing infrastructure on the pipeline to create a test EventBus and UserPool as needed (with easy integration to tools like sls-test-tools for integration testing).

More advanced adapters: Of course, another advantage of using the traditional "ports and adapters" approach is that we don't have to give an EventBus if we build our ports carefully. We've discussed the disadvantages of mocking from an integration testing perspective, and the change in weighting of the traditional testing pyramid of a heavier emphasis on integration testing. There are though situations where flexibility around the actual dependency can be useful — e.g. TDD, testing failure modes, and load testing.

Conclusion: You can't avoid all infrastructure dependencies — but you can choose how you handle them

Strong Domain-Driven Design and an Event-Driven Service-Oriented Architecture go a long way to help ensure a manageable and loose coupled architecture. There are, however, some cross-cutting infrastructure dependencies that occur and blur the services boundaries. If these are not made explicit and isolated, they can have a huge impact on the system design, coupling, cognitive load, and deployment ordering.

Taking a hexagonal architecture approach with clear Service Ports allows us to make these explicit, and remove cross service dependencies from deployment, testing and development.

We should use DDD to avoid dependencies, but not hide the ones we have. If there is a true infrastructure dependency (e.g. event bus, user identity), we should design for it — ensuring it does not open the floodgates to a complex

distributed monolith. In addition, don't hide these inside arbitrary host Services but create small, simple and purposeful Services to contain the configuration, deployment and testing of these dependencies.

Serverless-Transformation is an aleios initiative. aleios helps startups disrupt and large organisations to remain competitive using the best of Cloud-Native, **Serverless**.

Serverless | Domain Driven Design | Hexagonal Architecture | AWS | Eventbridge

Written by Ben Ellerby

663 Followers · Editor for Serverless Transformation

Founder of aleios (https://www.aleios.com/) and AWS Serverless Hero

Follow

More from Ben Ellerby and Serverless Transformation

Ben Ellerby in Serverless Transformation

EventBridge Storming—How to build state-of-the-art Event-Drive...

Serverless Architectures should be Event-Driven and use Amazon EventBridge. Use...

10 min read · Apr 7, 2020

616 5

Xavier Lefèvre in Serverless Transformation

Asynchronous client interaction in AWS Serverless: Polling,...

Event-driven Serverless often requires async update to the client. There are several ways ...

6 min read · Apr 18, 2020

730 7

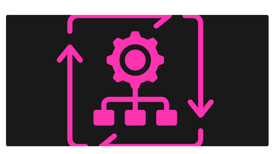
Xavier Lefèvre in Serverless Transformation

What a typical 100% Serverless Architecture looks like in AWS!

If you are new to serverless and looking for a high level web architecture guide, you've...

11 min read · May 19, 2020

2.8K 19



Ben Ellerby in Serverless Transformation

Enabling the Optimal Serverless Platform Team—CDK and Team...

We want our Cloud teams to be secure, fast, stable, reactive, autonomous and of course.....

7 min read · Aug 10, 2022

28 1

See all from Ben Ellerby | See all from Serverless Transformation

Recommended from Medium

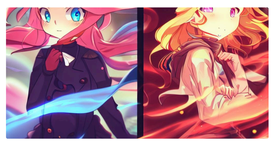
Yesarela Ritonga

Architecting AWS Lambda Functions in Golang: A Clean...

Optimize Development Workflow and Code Quality with Clean Architecture, AWS...

4 min read · Oct 9, 2023

10 1



Thinh Dang

Temporal vs Airflow: A Comparative Analysis

A comparative analysis of two popular workflow orchestration platforms, Temporal...

11 min read · Nov 4, 2023

9 1

Lists

General Coding Knowledge
20 stories · 1017 saves

Leadership
48 stories · 266 saves

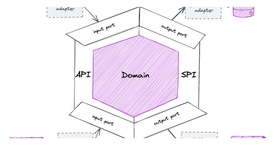
Natural Language Processing
1284 stories · 776 saves

Pubudu Jayawardana

3 ways to catch all the events going through the EventBridge Event Bus

3 min read · Nov 1, 2023

23



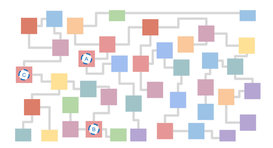
Jeroen Rosenberg in Jeroen Rosenberg

Why Ravioli Is My Favourite Recipe

Hexagonal Architecture

✦ · 14 min read · Feb 24, 2023

275 2



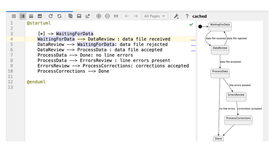
Luke Birdeau

Serverless Temporal

Elevate Redis from an in-memory data store to a game-changing service mesh. Turn you...

5 min read · Nov 3, 2023

4



Konstantin Ignatyev

How to visualize Temporal.io workflows

Hope everyone knows about Temporal.io (https://temporal.io/)—a great workflow...

5 min read · Sep 28, 2023

125 1

See more recommendations