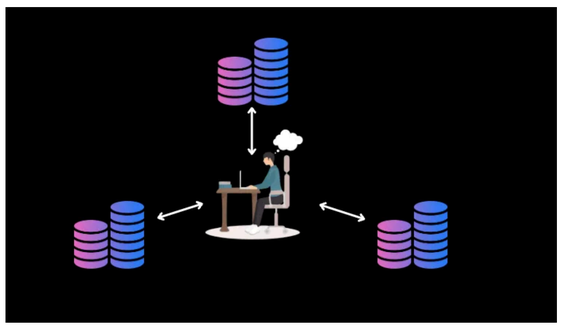
Building a Serverless Anti-Corruption Layer with CDK

Gregory Torrington · Follow
Published in Serverless Transformation · 8 min read · Dec 4, 2023



When building large distributed systems, interacting successfully with external systems can introduce large amounts of complexity into your system design.

Consider a company's software system that interacts with a third-party provider. A plan is made to move to a new provider which is faster and cheaper. An issue is identified that the protocol to interact with the old third-party provider is not compatible with the new one. The company will need to modify multiple sections of its software to interact with this new provider. They want to implement a design principle to account for this and allow for easier transitions in future.

A common solution is the Anti-Corruption Layer (ACL). An ACL implements the principles of Domain Driven Design (DDD). DDD is a software development approach that focuses on modelling software based on the real-world business domain it aims to serve. It emphasizes collaboration between technical and domain experts to ensure the software accurately reflects and evolves with the domain. DDD advocates for a ubiquitous language for all stakeholders and organizes the system into bounded contexts to manage complexity. These bounded contexts are called domains but are often referred to as separate services in the context of a distributed system.

ACL follows the principles of DDD by providing a separation of these services, allowing you to structure your business domains most effectively and ensuring that the design and language of your system are protected from 3rd party influence. In addition, when executed correctly an ACL can also prevent vendor lock-in, and provide easier visibility on your system's architecture.

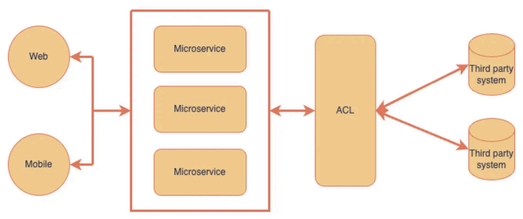
What is an ACL?

An Anti-Corruption Layer is a design pattern that provides the functionality to transform or translate information passing from one system to another. It is not purely a mechanism for communicating messages between two systems, instead, it provides guardrails to ensure that misinterpreted data cannot enter into and corrupt your primary system. An ACL can be designed as a package to be reused across other services, or as a self-standing service.

By acting as the communication layer between services, an ACL separates and increases decoupling allowing for independent scaling and greater portability of each of the services. Due to its low coupling with the rest of the software system, an external system, like a third-party service above, can be torn out when needed without large changes. This brings the added benefit that decoupled services can now be developed independently of each other increasing developer flexibility and build time.

"Whenever possible, you should try to create an Anti-Corruption Layer between your downstream model and an upstream integration model, so that you can produce model concepts on your side of the integration that specifically fit your business needs and that keep you completely isolated from foreign concepts. Yet, just like hiring a translator to act between two teams speaking different languages, the cost could be too high in various ways for some cases."
Vaugh Vernon
Author of Domain-Driven Design Distilled

In the example below, an ACL is positioned in between the internal software infrastructure and the third-party services.



In summary of an ACL:

- The communication between an internal and external system should be done through an ACL. The ACL sits between systems as an independent service or within one of the system's software architecture.
- The ACL should act as a translator of information from one system to present it in a recognisable format to the other system.

ACL as a Service:

An ACL can be created as a package or a fully-fledged service, however, having it as a service will yield the following benefits:

It encourages the abstraction of protocols that are required to communicate with external services. Building a whole new layer to handle the communication between services can become extremely complex. By encapsulating the ACL as a standalone service, an organisation is likely to have a dedicated team of experts building and maintaining this service. This protects developers from other domains needing knowledge of the intricacies of the ACL domain.

Very often, logic used to communicate with external services can result in a large repository of code. Having this within a package can quickly become difficult to maintain, document and use by consumers. A service simplifies this complexity by allowing the creation of standalone features, such as dedicated infrastructure and organisation of logic, in a way that best fits the ACL rather than its consuming services. In addition, the ACL service itself can leverage internal packages to reuse code across the service and the ACL team will have context on these internal development tools.

Ideally, having the ACL as a service, not a module or package within a service, will provide easier construction, understanding, and correct visibility of this tool at the levels required from those in and outside the ACL domain.

There are however a few problems you can run into when trying to implement an ACL as a Service:

- The ACL can increase latency between the two systems if the performance of the ACL is not optimised.

- An ACL will need to be maintained and managed effectively. This will require a dedicated team to become experts in this domain. The company should be ready to make this investment in its tech quality. This team will think about other teams in the company as its clients and build the product that works for them.
- It is important to take careful consideration of how the ACL will scale over time. The architectural design has a greater chance of planning for success after initiating a workshop such as Event Storming.

Building Blocks of an ACL:

When building an Anti-Corruption Layer, we can break down its core functionality into three mechanisms: Facades, Adapters, and Translators.

A Facade provides a simplified interface on top of a complex set of underlying classes. In terms of the ACL, it simplifies access for the client by making the subsystem easier to use. Since we know exactly what functionality is required from the 3rd party system when implementing this layer, it allows the ACL team to create a Facade that ensures only required features are accessible to the client and no more.

An Adapter allows the client to use a different protocol than the one understood by the system they are trying to interact with. When the client sends a message to the Adapter, it converts that message into one that the other system ('adaptee') understands. Another conversion occurs when a response is received from the adaptee and sent back to the client. This allows client teams to use protocols they are familiar with without letting the 3rd party influence their technical decisions.

The final element is the Translator. The Translator's job can be seen as a subset of the Adapter's. The translator's job is to translate the actual logic to be recognisable to the client's system. The job of converting protocols is an extremely complicated task and should be thought of as separate functionality. Each Translator is specific to the Adapter it serves.

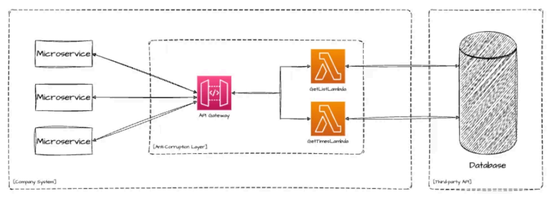
Building an ACL with Serverless:

Serverless boasts numerous advantages, including the ability to scale to zero cost when not in use, on-demand scaling to efficiently handle varying workloads, and the elimination of traditional server maintenance, leading to significant reductions in operational costs and enhanced resource optimization.

This Serverless infrastructure will be configured using the AWS Cloud Development Kit (CDK) and TypeScript for Infrastructure as Code (IaC). AWS CDK offers the flexibility to design cloud applications in an object-oriented manner using familiar programming languages.

AWS CDK streamlines the process of defining cloud resources in code, allowing for a more organized and manageable infrastructure setup. This method ensures a consistent deployment sequence and makes the infrastructure easily reproducible. Additionally, it significantly improves the ability to debug and resolve configuration issues, as the entire cloud environment is defined programmatically.

The example uses an ACL to interact with a third-party API. This API can be any external service, such as a database. See the diagram below for an architecture diagram of the AWS resources used:



The microservices interact with the ACL through an API Gateway that exposes two AWS Lambda functions providing Serverless compute. In Domain-Driven Design terms the ACL can be seen as the core domain, whilst the microservices and the third-party API interacting with the ACL are all separate domains.

In this example, there is a Lambda function, GetListLambda, which makes a request to the 3rd party database. This lambda fetches an array of information from the database.

Firstly we can look at where the Facade is implemented within the GetListLamdba. The client just has to make a valid request to the Lambda. Within the Lambda, the event is deconstructed and the 'authorizationToken' attribute is checked to allow access to the 3rd party database. The parameter 'code', which in this case, is a filter for users within a specific region, is included in the URL to be passed as a query string parameter to API Gateway.

```
export const main = async (
  event: APIGatewayProxyEvent | Partial<APIGatewayProxyEvent>,
): Promise<APIGatewayProxyResult> => {
  const authorizationToken = event.headers?.authorizationToken;
  if (authorizationToken === undefined) {
    throw new Error('Authorization token is not valid.');
  }

  let code = event.pathParameters?.code;
  if (code === undefined) {
    code = 'default-code';
  }
```

The Adapter makes requests to the 3rd party system. Firstly, there is logic to create a URL based on the 'pathParameters' from the event. Next, an HTTP request is made to the 3rd party system using 'fetch'. Error handling is included to confirm a successful response.

```
  const thirdPartyBaseUrl = 'https://third.party.api';
  const url = `${thirdPartyBaseUrl}/list?code=${code}`;

  const response = await fetch(url, {
    method: 'GET',
    headers: {
      ...event.headers,
      Authorization: authorizationToken,
      'Content-Type': 'application/json',
    },
  });

  if (response.status >= 400 || response.status < 200) {
    const err = new FetchError(response.statusText, `${response.status}`);
    err.code = `${response.status}`;
    throw err;
  }
```

The Translator is the last element in the process. By typecasting the response's request from the third-party API, we can work with the data received. However, it is important to note that this typecast is not runtime validation and we must ensure that sufficient research has been done on the structure of the response before creating these types. The data can then be deconstructed into a format suitable to the client.

```
  const listData = (await response.json()) as ListData;

  let totalValues = 0;
  const listLength = listData.items.length;
  for (let i = 0; i < listLength; i++) {
    totalValues += listData.items[i]?.value ?? 0;
  }

  const listResponse = {
    totalValues: totalValues,
    listLength: listLength,
    ...listData.items,
  };

  return {
    statusCode: 200,
    body: JSON.stringify(listResponse),
  };
};
```

Result:

Building an Anti-Corruption layer can be a complex and resource-intensive task. It is important to ensure communication boundaries have been addressed between teams that will use this layer and that sufficient resources have been procured to build and maintain it.

However, if created correctly, an ACL can provide the best protection for your system acting as a guard to prevent influence and leakage of design patterns from external systems. An ACL allows you to follow the principles of Domain Driven Design in your company's system no matter the 3rd party systems you interface with. It also allows you to swap and change external systems more easily without large and fundamental code changes, reducing vendor lock-in. The principles of an ACL such as Facades, Adapters and Translators can be applied to the simplest of systems such as the example given but also to the most complex of enterprise software.

Code and resources:

- Example repo: https://github.com/greg-torrington/serverless-acl
- https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer
- https://www.atelier-solutions.com/the-anti-corruption-pattern-and-legacy-migration-strategies/
- https://codeopinion.com/anti-corruption-layer-for-mapping-between-boundaries/

Domain Driven Design | Serverless | Serverless Architecture | Design Patterns | Ddd

Written by Gregory Torrington

4 Followers · Writer for Serverless Transformation

Follow

More from Gregory Torrington and Serverless Transformation

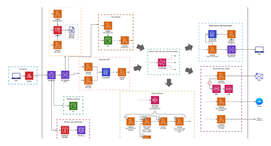
Xavier Lefèvre in Serverless Transformation

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

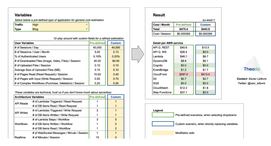
Sarah Hamilton in Serverless Transformation

Building a massively scalable Serverless chat application with...

AWS AppSync allows developers to build quickly. Let's look at how to build a scalable...

9 min read · May 11, 2021

268 4



Xavier Lefèvre in Serverless Transformation

Is serverless cheaper for your use case? Find out with this calculator.

Some fixed architectural opinions simplify the process of estimating a serverless project...

10 min read · Jul 30, 2020

720 2

See all from Gregory Torrington | See all from Serverless Transformation

Recommended from Medium

Amo Moloko

Migrating from the Serverless Framework to AWS CDK

Serverless as a technology is no longer a fad as we venture into 2024. Aleios who have...

12 min read · 4 days ago

41

Błażej Kustra in Software Mansion

Modeling with DynamoDB made easy in Typescript

Dynamode - NodeJS ORM

6 min read · Oct 2, 2023

766 1

Lists

General Coding Knowledge
20 stories · 993 saves

Leadership
48 stories · 262 saves

Natural Language Processing
1265 stories · 752 saves

Haiko van der Schaaf

AWS DynamoDB made easy

Simplify your DynamoDB Typescript code with DynamoDB Entity store

6 min read · Jan 2, 2024

5

The SaaS Enthusiast

Advanced Serverless Techniques V: DynamoDB Streams vs....

Using DynamoDB Streams to trigger AWS Lambda functions directly versus using an...

9 min read · Feb 27, 2024

57

Maksim Zemskov in ITNEXT

Mastering Type-Safe JSON Serialization in TypeScript

How to utilize TypeScript for type-safe serialization and deserialization of data in...

9 min read · Feb 27, 2024

Alae Achhab

Applying Domain-Driven Design (DDD) Principles in Frontend...

Starting with a Next.js Application: A Proposal for Separating API Calls, Business Logic, an...

3 min read · Nov 2, 2023