thoughtworks

# #Defaults
## Back-end

A set of sensible defaults for engineering

Front-end web | Front-end mobile | Data | Emerging Technologies

# Table of contents

## Code and code-level design

- Code structure and design validation as automated Architecture fitness function
- Monorepos for back-end repos as a sweetspot between cohesiveness and flexibility
- Observability as code
- Rich Domain models over anemic domain models
- Ports and Adapters (Hexagonal) Architecture for backend

## Testing

- Leverage testcontainers for tests that involve databases, other services or even cloud components
- Bi-Directional contract testing for frictionless testing of distributed systems
- Test transaction headers
- Shareable API collections to ease testing with multiple scenarios
- Mutation tests to assess the quality of tests

Developer experience

- Service templates and starter kits for consistency and quicker bootstrapping
- Component stubs and verified component stubs for faster, predictable development and testing

Resilience and stability

- Resilience and fault-tolerance for service providers and service consumers
- Reliable retries for improving resilience
- Service health-checks integrated with automated restarts

Security

- Static Application Security Testing tools for DevSecOps
- DAST for automated penetration testing
- Lightweight and secure base images for application containers

# Introduction

The #defaults report consists of a set of sane engineering decisions and choices that our teams have made. It complements some of the existing artifacts of Thoughtworks - such as the Tech Radar and the Looking glass report. Currently produced by the Engineering Directors (formerly Office Tech Principals) and Heads of Technology of India, preparing the #defaults report involved connecting with various project teams, identifying patterns, synthesizing them and eventually coming up with a repository of architectural and engineering choices. #defaults reports would be released for each **technology domain:** the web front-end report is already here, and the one you're reading currently is the back-end #defaults report. The Data and AI, mobile and XR reports would be released in the upcoming months. More importantly, the report contents will be iterated upon.

A note on how the report is structured: for a given technology domain, there are several aspects or considerations called **facets**. For example, the back-end domain consists of facets such as architecture, code and resilience. Under each of these facets, we have what we call as **signals** - which are the defaults themselves!

Every signal has a narrative that outlines the context, the reason behind the decision, some of the accounts where the signal has worked well, a peek into the How-to part, and a set of learnings and challenges.

As this report contains a curated list of engineering choices firmly rooted in our experience, we hope it elevates the game for both our clients as well as the industry at large. The report is by no means exhaustive in terms of coverage or details, yet attempts to cover relevant themes and provide enough details to pique your interest. You could validate or critique some of your own choices with the ones presented here. We have tried to add as much context as possible and there is bound to be an element of subjectivity. Needless to say, the “How” part contains a few possible ways to realize the signal - but not the only ones. We encourage you to reach out to the practitioner accounts mentioned in each signal in order to delve deeper into them.

This version of the report is meant for internal consumption: you will find the names of clients, specifics etc., so that the readers can approach the respective project teams to get more details. In due course of time, we would be releasing a sanitised version of the report for the industry.

This report has been made possible by the amazing support of our communities - who contributed to the making of this report through experience-sharing and reviews. A big thanks to all of them.

For any feedback on the report - say if you want to add your point of view on a signal, you can either directly comment on this document or write to the hash-defaults team.

# Back-end

The context of back-end domain has grown from server-side code to encompass considerations such as architecture, builds and CI/CD, testing, system qualities such as resilience and security and so on. An explosion of tools and technologies have not only excited but also bewildered us to an extent. However as the wave of excitement subsides, we notice a greater emphasis on stability and best practices. In this report, we highlight stories on key trends as well as the top recommendations that have stood Thoughtworks teams in good stead. Building robust and extensible APIs seem to be a key focus area for many of our teams using microservices - evidenced by techniques like API-first design and API orchestration. So is the admiration for events and asynchronicity. Like all things in engineering, this is a trade-off: decoupled systems bring their own set of challenges and complexities. Thankfully, some of our signals on developer experience (see service templates and verified component stubs), testing (see bi-directional contract testing and test containers) and resilience (see fault tolerance patterns and reliable retries) address some of these pain-points. Beneath the proliferation of tools and technologies, our teams swear by time-tested practices such as hexagonal architecture, security testing and architecture decision records.

# Facets

## Architecture

### Back-end-Arch-001 - Architecture Decision Records (ADRs)

**[What is it]**

Whenever we make an important choice that'll influence our software design and the subsequent direction of the team, we should make sure it is recorded and traceable. This can be achieved by having ADRs (architecture decision records) in the components of our system. An ADR is a small document which is used for documenting design decisions.

**[Where have we used it]**

- In Mercedes-Benz OTR we have used it in the Custamigos Team
- It is also used in Axis Bank for Maximus Platform

**[Why did we take this choice]**

Previously, each service in the Mercedes-Benz account handled its documentation differently. Some used confluence and some used Readme files - each having a different structure and format. Developers found it difficult to look into multiple sources. By capturing design decisions in a consistent manner in source control, there was a single place for developers to look at design decisions. This is also helpful for onboarding new people to the team: they can get context on previous design decisions made by the team.

In Axis, Architecture Decision Records were needed for similar reasons. People continue getting rolled off from the account and new people join over the years. Eventually very few people (or worse, no one) would know the real reasons behind certain architectural decisions. ADRs help answer those questions even after years of the decision being made. It also helps make new decisions based on the context of the old ones.

# Back-end

**[How did we do it]**

## ADRs segregated by repository

The ADRs in the Mercedes Benz OTR team are stored as markdown files at the root level in each service repository. ADRs within a service repository signify the design decisions related to that service. This is applicable for design change across backend or frontend. Over a period of 5 years, one service repository accumulated around 17 ADRs.

The ADR document at Mercedes-Benz follows the following structure:

| |
|---|
| **Context and Problem Statement**<br><br>*What is the issue that we're seeing that is motivating this decision or change?* |
| **Considered Options**<br><br>*What is the change that we're proposing and/or doing?* |
| **Decision Outcome**<br><br>*What is the outcome and what becomes easier or difficult after this change?* |

The ADR generation was also part of the project kickstarter.

# Back-end

## One account-level ADR repository

**Axis** has a single repository called “adr” in bitbucket, dedicated for the entire platform and all teams. It contains an mkdocs project and uses mkdocs-material plugin. This essentially means that one could write the ADRs in markdown and publish them in a static website. The website is hosted in the internal network and visible to all account members.

This is the template used in Axis:

```
---
tags:
  - Tag 1
  - Tag 2
---

# [Title]
Date: [Date]

## Status
[InProgress | Proposed | Accepted | Superseded]

## Context

[Business Context + Possible solutions]

## Decision
[Final decision taken]

## Consequences
[Additional things]
```

Using mkdocs-material gives us benefits of

- In-browser searchable ADRs (using lunr.js)
- Organization using tags and categories
- It supports images, diagrams (using mermaid syntax) and code snippets
- Since it is git-backed, it automatically captures the history of changes to ADRs with its authors

The repo has an associated CI pipeline which builds and publishes the static website on every commit. As of August 2023, Axis Maximus has 61 ADRs which span across 5 years and 4 Tech Principals.

# Back-end

**[What were the learnings and challenges]**

1. In Axis, as the account rapidly grew in size, one challenge was to ensure that everyone was aware of the ADRs and that everyone was making an effort to keep the ADRs up to date. The biweekly Tech Lead catch up was used to discuss platform architecture designs and assign ownership to add an entry in the ADR repository at the end of call when required. This ensures that the direction taken on the platform is validated with a larger group and that everyone is onboard with the decisions.
2. There was a light-weight approval process at Axis that was put in place for ADRs: 1 Tech Principal and 2 Tech Leads had to vet the ADR.
3. At Mercedes Benz, whenever a story that involves the implementation of an architectural change is played, the task to document it as an ADR is mentioned in the user story (in JIRA). The newly added ADR in the service is then reviewed during desk-check by the TL and QAs.

**[References]**


1. ADR process - AWS Prescriptive Guidance
2. Material for MkDocs

# Back-end-Arch-002 API-first design and API standards with linting assistance

**[What is it]**

API first design is an approach where people collaborate and work on designing the API construct first before fully working on an implementation. The construct must include but not limited to the request and response format, url with http action, possible errors with status codes, customized error message. To guide evolution, API designs are often complemented with API linting. API linting is a process of analyzing and enforcing a set of rules, best practices and standards for the APIs. Factors like naming conventions, versioning, documentation, schema validation, query parameter validation can be standardized through API linting even before we start the implementation.

Design/API first approach along with API linting is a good choice of enforcing standardization.

**[Where have we used it]**

We have used the API-first design approach in GXS account - a digital bank operated out of Singapore for all the backend APIs. This approach is proposed to be used at Pepsico - API Management.
Kroger also has been using this technique, especially when creating new APIs or migrating over old APIs to incorporate new API guidelines.

**[Why did we take this choice]**

Microservices are becoming the norm, and API-first design solves some of the challenges associated with microservices development:

- Bringing understanding and clarity: Sharing the API design with other teams in a microservices-based development environment helps bring clarity to the expected behavior of an API.
- Third party integration: Well defined API interfaces can be easily shared to external systems.

- Documentation: Automated document-generation tools are available for API-first design frameworks like OpenAPI and protobuf, and these documents can be always kept in sync with the API specifications. At Kroger, we publish all our contracts to a central catalog called the Developer Marketplace, thus aiding in easy discovery of services and avoiding duplication.
- Parallel development: Although an anti-pattern, we're seeing many clients structuring their teams as front-end vs back-end. By sharing the API structure to frontend teams beforehand, we make sure that experience (front-end) and backend can start working independently. As long as front-end teams know about what the API has to return, they could freely mock a response and work on UI separately without waiting for the backend API to be fully developed.
- Code generation: With the construct designed through data definition languages like protobuf, we could use a compiler to generate code base in the language of our choice. This can serve as models for backend logic. For example in GXS, we generated entity code from proto files using a proto compiler for the GO programming language.
- Specifications and Standardization: API Linter tools ( spectral,  protolint) will help us in standardizing the API specifications across the projects in the organization.

**[How did we do it]**

1. **Protobufs (in GXS account)**

Protocol Buffers is a language-agnostic serialization format developed by Google that is commonly used for defining structured data and messages. While protobufs are often associated with efficient data serialization and deserialization, they can also be used to define API contracts in a clear and structured manner. Protobuf offers:

- strong typing, reducing the risk of data related issues.
- The .proto files provide a structured way to define your API contracts, making them clear and well-documented.
- Backward Compatibility: Protocol Buffers support versioning and backward compatibility, which can help with evolving your APIs over time.
- Efficiency: Protocol Buffers provide efficient serialization and deserialization, which can result in faster communication between clients and servers.

In a .proto file we declare the request ,response formats, api method as below:

```
syntax = "proto3";
package myapi;
service WeatherService {
    rpc GetWeatherInfo (WeatherRequest) returns (WeatherResponse);
}
message WeatherRequest {
    string location = 1;
}
message WeatherResponse {
    string temperature = 1;
    string conditions = 2;
}
```

We then use a protobuf compiler to compile the above message and convert to our desired programming language, say GO.

For API linting, protolint can be used. Protolint is a pluggable linting/fixing utility for Protocol Buffer files.

## 2. OpenAPI specification (Pepsi account)

Spectral is an open-source API description linter that can be used to enforce API style guides, identify errors and warnings, and improve the quality of API descriptions. It supports linting for Swagger 2.0, OpenAPI v3.1, v3.0, v2.0, and AsyncAPI v2.x.

Spectral comes with a set of built-in rules that can be used to lint API descriptions, but it also allows users to create custom rules to meet their specific needs. Spectral can be used as a command-line tool so it can be easily integrated with CI Pipelines.

**[What were the learnings and challenges]**

Some of the challenges in this approach are

- API Changes over time: When there are additional requirements, some APIs tend to change in terms of request and response parameters. When changing request parameters, we make sure that the changes are backward compatible.
- Consistent Documentation: Some accounts maintain a confluence page for each service. Few of them were not consistently updated to the latest API specifications. Other accounts solve this problem by automating the generation of documentation through CI pipelines that publish the enterprise-level API documentation to a static website.

**[References]**


1. Understanding the API-First Approach to Building Products
2. API Linting - Spectral
3. API/Design First Approach
4. https://github.com/yoheimuta/protolint

# Back-end-Arch-003 -API Orchestration using a workflow engine

**[What is it]**

An API that satisfies a use-case can have several tasks that need to be accomplished. Workflow engines are powerful abstractions that can weave together these different tasks: in short, they are state machines which have a set of predefined steps that need to be performed in response to an API trigger - some of these could be synchronous, some could be asynchronous.

For example, an attempt to take a loan in a banking system can involve several high-level steps/tasks like authorizing input, transfer process within internal accounts, creating a new account for the loan, creating the EMI schedule for the account, publishing the transactions involved to a separate service and finally sending notifications to the end user.

**[Where have we used it]**

We have used the workflow engines in

- GXS account, a digital bank based out of Singapore
- Axis Bank India, Maximus Platform - a Loan Origination System

**[Why did we take this choice]**

The use of workflow engine has the following advantages:

- Asynchronous processing - Non blocking communications. The API's response might be immediate, but internally the workflow engine could be orchestrating complex task dependencies, i.e. some tasks may have to be performed asynchronously and some may need to wait for the completion of others.
- Event driven continuation - Though asynchronous, the workflow can also wait for certain events to happen and then continue execution.

- Easier to define and understand - The workflow steps can be easily defined, with each step having a handler logic.
- Success and Failure Steps - With complex engineering, failures are unavoidable, which means proper exception handling needs to be in place for systems like banking. With the workflow engine, with each step we define a state for success as well as failure.
- States Recording - In the workflow engine, every step is a state. A state uniquely identifies the step the workflow currently is in. Also once the workflow successfully executes a step, the outcome is recorded in DB. The record also contains additional payload that the workflow was using.
- Failure and Retry Mechanisms - With workflow, we also define the retry options, like the number of possible retry attempts for every step.

**[How did we do it]**

In GXS, an in-house library (written in Golang) was developed for the workflow engine. However there are standard third party libraries available in the market like Temporal to facilitate complex orchestration processes.

Defining a simple workflow for a backend application involves the following:

1. Define the workflow, this involves creating the steps involved for a process.
2. For each step, include the handler logic which gets executed when the step is encountered.
3. Assign states for each step. Once the handler logic completes execution, define the state for success and failure based on the outcome of handler logic.
4. If the steps involve waiting for a particular event to happen, define the event and assign that to the workflow step. By this way we make the workflow to wait for an event to happen.
5. Register the workflow defined in the main part of the application.
6. Invoke the workflow with respective payload if any.
7. Update the workflow with any state changes during handler executions.
8. Make sure to interfere with the event hooks and to record every step in the database.

# Back-end

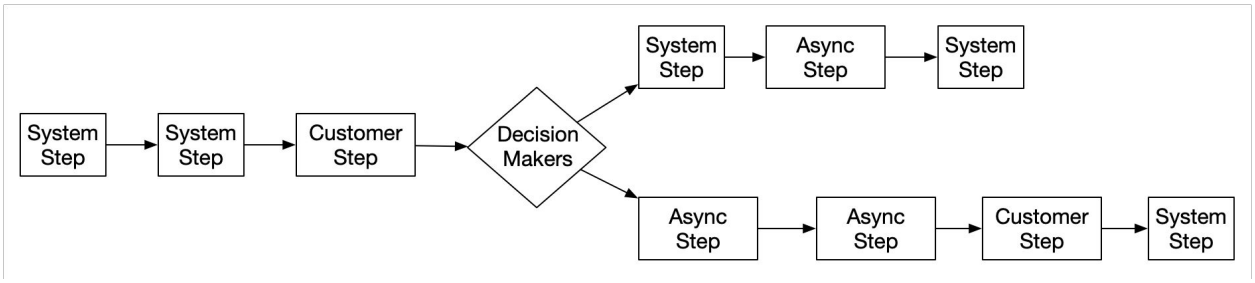
Axis created an in-house orchestration engine which was DSL-based. The technologies to build the engine were Kotlin and Project reactor. The Kotlin DSL made it easy for developers to create orchestration flows by composing different steps together either sequentially or in parallel.

Every step is backed by mongoDB persistence and every step output is stored. This helps in making the user journey more resilient. This is needed because often the journeys span across multiple days.

Steps can be of following types:

- A System Step is simple and completes when the handle function ends execution.
- A Customer Step requires customers input which is usually associated with a browser interaction.
- An Async Step completes with a Kafka message

Here's an example of a workflow:




**[What were the learnings and challenges]**

- Creating a custom orchestration engine requires a lot of upfront time and effort. At the same time, it will also demand continuous maintenance over the years.
- Bugs in the engine can mean unpredictable and hard-to-debug issues that can affect multiple workflows. Documentation is key here. Also, having a team that has ownership and knowledge of the workflow engine codebase helps.
- Orchestrating multiple services, tasks, and interactions can lead to complex workflows. Maintaining a clear understanding of the end-to-end process and the relationships between different steps becomes challenging.

# Back-end

- Clear documentation with sequence diagrams are always required and proper functional knowledge is essential as one single mistake could lead to multiple complications. Axis had automatic workflow diagram generation that was useful for understanding as well as getting validation from stakeholders.
- Testing workflows comprehensively can be difficult due to their asynchrony, concurrency, and interactions with external services. Integration testing is particularly challenging because it involves injecting mock services.
- Easier to debug when there is a failure as the state is maintained in the DB. Using a combination of state and logs, it is easier to find the root cause of the issue.
- Do a thorough tradeoff analysis on build vs buy: we built some of the workflow engines because there were features like rerunning validations on resume, parallel sync and async calls, DSL-based workflow definition, etc. that were not available with tools back then. However, newer frameworks like Temporal may address some of these concerns.
- Since journeys can span across multiple days sometimes weeks, consider how you are going to upgrade the journeys while maintaining compatibility with existing data
- Consider horizontal scaling - unsafe state management might mean concurrent journeys overwriting each others states

**[References]**

Modern day service orchestration using DSLs - Axis case study

# Back-end-Arch-004 - Authentication and authorization modularized into API Gateway instead of hand-crafted and proliferated identity solutions

**[What is it]**

Use API Gateway to authenticate and authorize an incoming API request by taking in an access token. Once authenticated and authorized, the request is enriched with the validated user identity. Using API gateways for this functionality is a much cleaner approach than building custom solutions and integrating hand-crafted libraries into various microservices.

**[Where have we used it]**

This pattern was used in following projects:

- GoJek - where we leverage Kong API gateway
- BP - where we leverage AWS offerings
- Otto - where we leverage AWS offerings

**[Why did we take this choice]**

Since authentication and authorization is pretty much needed for most APIs, the API Gateway pattern recommends doing this in one place - the API Gateway - and propagating the validated identity to all services.

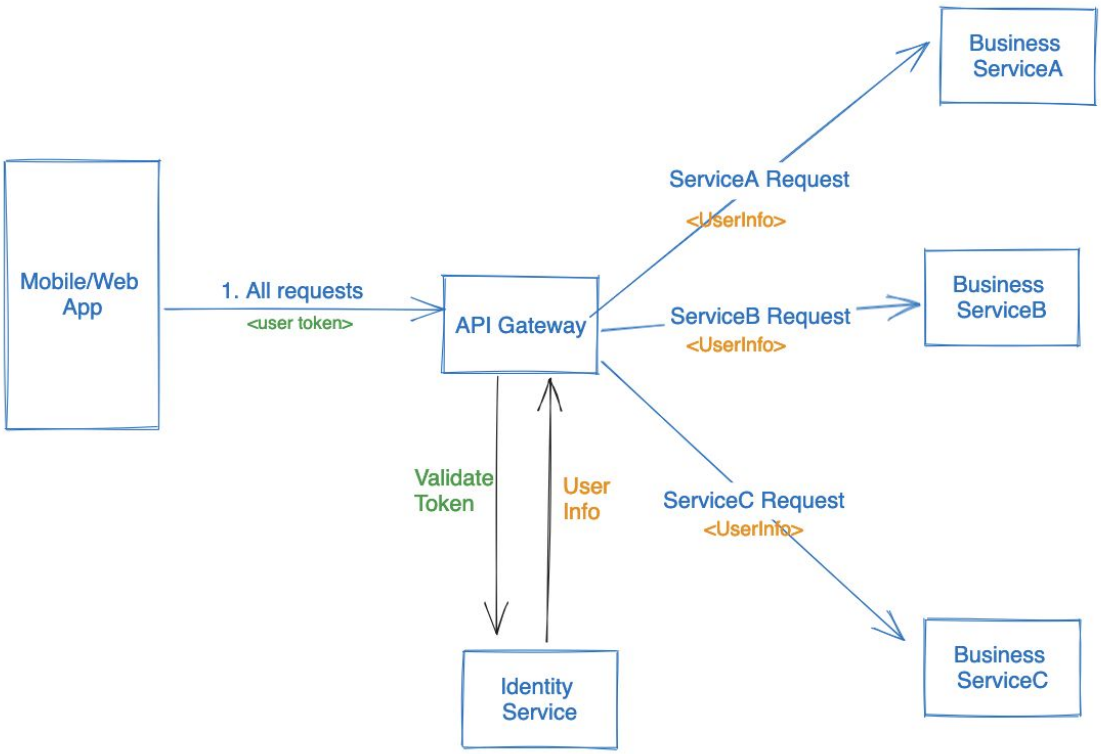
Reasons behind using this approach:

- Each microservice does not need to re-implement identity resolution, thereby ensuring an accurate and secure implementation.
- Improves overall security of the microservice ecosystem by limiting complex implementations to specific parts.
- Uniform user identity implementation across microservices
- Externalized authorization
- Caching of user info for the validity of user token will reduce the load on identity service.

# Back-end

**[How did we do it]**

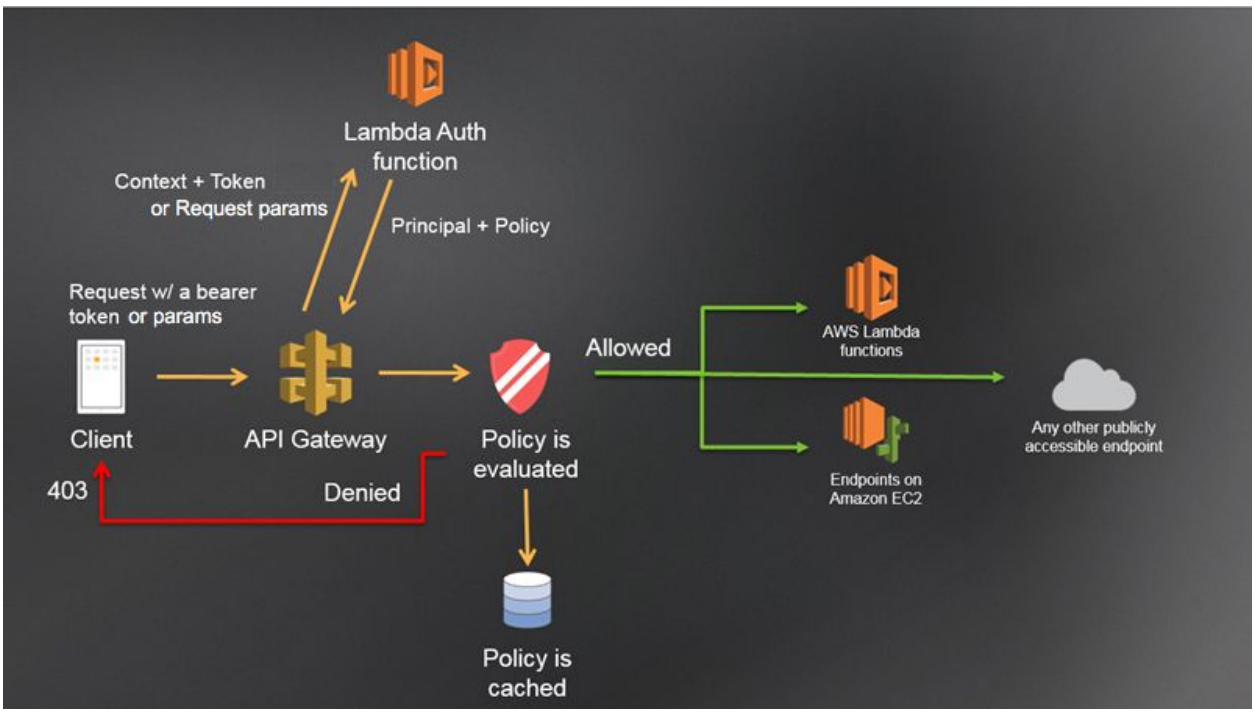
Nearly all user-facing API implementations would need a guarantee that the user mentioned in the API request is the one calling the API. To achieve this, we mandate an “access token” in the request, and resolve this access token to a user-id with the help of some kind of identity service. The main takeaway is that we never take a user-id in the incoming request. The below diagram shows a high level approach used at GoJek:




The super app hits a Kong endpoint, which resolves the token with its own Identity service. Thus, the business microservice squads (GoRide / GoFood / GoSend teams) could register their endpoint with Kong, and get the user information in the request.

# Back-end

BP and Otto leveraged AWS offerings - specifically AWS API Gateway and Lambda authorizer for API authentication and authorization. The following diagram shows the approach:




The access token can either be something which has to be validated with an Identity provider (as in the case of BP, where the access token is opaque) or a self-contained JWT token (as in the case of Otto).

The AWS API gateway provides a way to authorize requests using 'Lambda Authorizer' integration - A customized lambda to authorize the incoming requests with roles and scopes. The logic to validate and extract the user information, roles, scopes to authorize etc., can be written within the Lambda authorizer, which is then made to return an IAM Policy back to the API Gateway.

The gateway transparently validates the policy before forwarding it to the target APIs or lambdas.

# Back-end

**[What were the learnings and challenges]**

- AWS AppSync (GraphQL service) also has support for Lambda Authorizer.
- Most API Gateway products usually have out-of-the-box integration and support for authentication and authorization.
- Theoretically this introduces a single point of failure, however in practice, we never had a problem.

**[References]**

1. Lambda Authorizer for AWS API Gateway here.
2. Approach for Kong here.
3. Lambda Authorizer for AWS App Sync here.

# Back-end-Arch-005 - Non blocking server-side APIs for efficient resource utilization and scalability

**[What is it]**

The concept of “non blocking”, in the context of APIs, means choosing a server implementation which does not allocate one dedicated thread per incoming http request. In a non-blocking approach, the server retains a small fixed set of threads. Each incoming request is randomly allocated one available thread to work with. As long as the request performs CPU-bound activities, the allocated thread is used to service it. When the request performs IO activity, the thread is recycled by the server back to the pool. When the request completes the IO activity and needs to proceed, the server allocates any available thread from the pool. This technique allows the threads to not be blocked for the duration of IO. In that duration, the thread is used to service some other request.

**[Where have we used it]**

All projects (ex: TBC and Space Bank) using the modern ASP.NET (Dot Net) stack should automatically be using it.

Many teams (ex: MB-VerSO-PMD) in Mercedes account use Kotlin with Ktor framework. Ktor is asynchronous and non-blocking by default.

Teams (ex: MB-OTR-Gateway, Tesco, OTTO-Dragonfish) using Spring 5 Webflux also uses this technique implicitly via Java Reactor framework.

Teams using Node.Js in the backend automatically use this technique. Most notable being ExpressJS.

Teams (ex: Voyager, Experian) using Go Language for backend apis will automatically be using this technique. Go uses a technique called GoRoutines which allows asynchronous execution, which is especially evident for IO routines.

**[Why did we take this choice]**

The primary benefit of this technique is increased concurrency per server instance. There are a few considerations to make the decision of going with a non-blocking I/O paradigm.

For a given server instance,

1. Do we expect a high, and perhaps unpredictable, number of concurrent requests?
2. Are the requests predominantly IO bound - from the perspective of this server instance?
3. Is the team mature enough to adapt to slightly different coding practices?

If the answer to all the above questions is YES, then a non blocking, asynchronous, tech stack should be considered for the project.

**[How did we do it]**

Non blocking (aka asynchronous) APIs means an implementation where the number of concurrent requests that the server can admit is not limited by the number of threads that the server can spin. Our accounts have implemented non-blocking APIs with the help of several frameworks or libraries:

1. NodeJs for backend apis. For example using ExpressJs.

   NodeJs is a purely asynchronous platform. Any attempt to create a synchronous system on it is not only counter intuitive, but will also severely degrade server performance. From ingress to egress, all nodejs based operations and libraries are asynchronous. All IO libraries like file and database libraries are capable of returning a `Promise`. Depending on whether the consumer code is written in ES6, Typescript or older ES5, the technique to `await` upon the promise will change.

2. Modern ASP.NET (Dot Net) platform

   Older Dot Net Framework and modern Dot Net Core, both are capable of developing both synchronous and asynchronous api servers. Both C# and VB.Net languages support asynchronous programming with `async` and `await` keywords. The major difference between conventional synchronous and modern asynchronous apis is to have `async` controller methods and returning `Task<T>` instead of a concrete object instance. There are other subtle differences as well. For example `Thread.Sleep(100)` vs `await Task.Delay(100)`. Some .NET APIs started providing out of the box async model implementation only from .NET 4.5 onwards. Hence the implementation of such APIs in older tutorials/code samples may vary from modern implementations of their async avatars.

Taking async to the next step, we can stream data remotely from API to client with `IAsyncEnumerable` support. This way both client and server won't be blocked. Controller methods directly returning instances of `Foo` are NOT asynchronous, and hence not non-blocking.

3. Kotlin JVM + KTor.

This is so far the most seamless non blocking asynchronous server framework built atop JVM. The best thing about this combination is that it fully leverages Kotlin's language level support of Coroutines. This means many of the idiosyncrasies peculiar to thread-reuse and reentrant-functions can be ensured at the compile time.

In some ways this combination is even better than ASP.NET, which has to deal with backwards compatibility of some legacy language features. For example, with Dot Net, you have to explicitly return a `Task<T>` even after marking a method with keyword `async`. Kotlin does away with this and introduces a much richer syntax of un-decorated return types under `suspend fun`. Kotlin language can be used with many popular server frameworks like SpringBoot, but with the KTor framework its coroutines are seamlessly integrated.

4. SpringBoot with Java 8+

SpringBoot with Java is conventionally synchronous by default. SpringBoot is built atop Java EE's Servlet framework, and relies on the threading system of the underlying server. So depending on which server is used - Tomcat, Jetty, JBoss, Undertow etc, the threading mechanism and configurations differ slightly. For example, in default configuration SpringBoot on Tomcat server defines `server.tomcat.max-threads=200`. That is 200 maximum concurrent requests, even if all of the 200 requests are waiting for response from the database or some upstream service.

To some extent, this hard limit on thread-count can be alleviated by using the `@RestController` methods returning `CompletableFuture<T>` instead of a concrete object instance. This needs an additional `@EnableAsync` configuration. Do note that true throughput is still limited by the thread count defined in the `@EnableAsync` configuration. The improvement we get is that now the server will not return 503 Service Unavailable upon reaching the 200 concurrent request mark. More than 200 requests can now be held open for processing, all waiting on the executor thread pool while not blocking any of the original 200 server threads.

5. SpringBoot with WebFlux stack.

   While conventional SpringBoot is built atop the J2EE Servlet framework, some servers like Netty allow for an inherently native non blocking asynchronous request processing. Even though the servlet framework cannot leverage that capability, there is a Java Reactor Framework which can. Spring 5 onwards supports the Reactor Framework. The asynchronicity they provide is signified by controller methods returning `Mono<T>` or `Flux<T>` instead of a concrete object instance. Spring 5 WebFlux stack can be crossed with Kotlin. This way, the `Mono`/`Flux` technique can be hidden behind Kotlin's Coroutines. This removes a lot of syntactic complexities of the reactor framework, but introduces Kotlin Language instead.

6. Akka Toolkit and Runtime

   Running on top of JVM, Akka toolkit facilitates development of not just concurrent but also distributed applications. Akka comes with a slight higher learning curve, but provides more features and flexibility. Similar to Spring WebFlux, Akka is also asynchronous by default and requires a slightly specialized way of coding. An incoming HTTP request is picked up, parsed and then the computing resources (thread) are relinquished until the response is available on the cluster. No busy waiting happens. The cluster in its turn is entirely message driven. The incoming request is transformed into an `ask` message and is routed to a relevant `Actor`. The actor object processes the message and returns the reply. If a database call or similar IO is required, that action is delegated to another specialized actor and the previous actor is recycled to work on any other available message.

**[What were the learnings and challenges]**

1. The term non-blocking is also used in some other similar but different contexts in computer science. One such example is in the context of locks. When a shared resource like a global variable is accessed by multiple processes (or threads), it becomes necessary to have a mechanism for locking (temporarily preventing) access to that resource when one process is updating it. This is done to prevent two parallel processes overwriting or corrupting each other's data due to simultaneous write. These locks are also known as Mutex and Semaphores.

   In the context of this signal, the term Non-Blocking is used to refer to some algorithms which allow concurrent processes to safely access shared resources without resorting to locks. Such algorithms are also known as lockless algorithms, in addition to being called non-blocking algorithms.

2. As per the design of most of the frameworks and platforms, asynchronous non-blocking paradigm shines well in I/O-intensive operations. CPU-intensive operations are better off handled in a multi-threaded fashion.

3. Do not set asynchronicity as a first order requirement. Our clients want a well working scalable solution. If that solution can be implemented via conventional synchronous ways, then be it. Asynchronicity should be a natural solution and not a force fitted one.

4. Team capability and maturity should be taken into account while planning for these technologies. There are some nuances to learn and stumble into. If the team is not set for some moderate learning in a short span, introducing asynchronicity is perhaps not the right time. Asynchronous programming improves user experience but involves a learning curve especially in advanced scenarios or in exception (error) cases. If not handled properly actual exceptions will be lost and will introduce bugs which will be difficult to backtrack.

5. If a project opts for a non-blocking asynchronous paradigm it is advisable to prefer platforms having asynchronicity as a first class concept rather than supported via libraries or wrappers. For example, Java, the language and its inbuilt classes (the Java Class Library) achieve some form of asynchronicity via `CompletableFuture<T>` and related techniques. While this does enable Java to implement asynchronous http servers, the result is usually convoluted. The developers need to be very careful in performing IO like database and network calls. For example the entire JDBC library either won't work or severely degrade server performance. Only libraries that support some form of `Future<T>` should be used. Compare this to the language and compiler level recognition of asynchronicity in NodeJS and Spring WebFlux. There it is asynchronous by default. It would be super difficult for a developer to mistakenly mix synchronous and asynchronous methods in the same flow.

6. Test test and test. While multithreading is super difficult to test, asynchronicity is comparatively easier. All of the above mentioned technologies also supply their testing frameworks. Asynchronous systems should also be load tested, even more rigorously than their synchronous counterparts. Thread leak happens, often due to mistakes in coding. Given that most such platforms work with only a handful of threads, such a leak causes sudden and abrupt choking of the server. Hence load tests are mandatory for asynchronous systems.

7. Do not mix synchronous and asynchronous libraries. Especially for the IO operations like network, disk, database etc. Synchronous database libraries like JDBC should not be used in asynchronous context. In Kotlin world, there are libraries like JetBrains Exposed. Similarly there is R2DBC for Spring WebFlux.

8. If Synchronous libraries like classic JDBC need be used within an asynchronous context, care must be taken to isolate the synchronous and asynchronous parts of the code with some sort of executor thread pool. The techniques of safely performing synchronous operations within asynchronous context is beyond the scope of this text.

9. Logging exception stack traces could be an issue with asynchronous systems. Logs, especially exception logs, should also be tested at least manually to ensure their usefulness. The developers can use key/value logging techniques to capture sufficient contextual information. One such technique could be Markers

10. Tracing, especially with MDC type correlation Ids can be difficult and sometimes error prone with asynchronous contexts. Logs should be tested, at least manually to ensure whether the context-id is being logged. Care should be taken with logs originating from within libraries, as they may or may not plug into the ambient MDC mechanism.

11. An alternative for java reactive programming - Project Loom.
    From the perspective of Java language - Though the reactive asynchronous programming style is very much favored when the application is predominant of I/O operations and gives us an efficient resource utilization, it comes with the cost - like the learning curve and maintenance due to its different coding style, the necessity of all the dependent framework/library has to be on the reactive stack as well (meaning a full non-blocking code). What if we could still use the resource efficiently without any cost ? This is where "Project Loom - Virtual Threads", introduced as part of java 19 comes in.

    Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and observing high-throughput concurrent applications.
    It preserves the thread-per-request model while utilizing the hardware optimally, as opposed to reactive thread-sharing model using the concept of "Virtual Threads". For java developers this gives the mindset change from threads "being costly" to now "being cheap". So, if we need any I/O task to be performed, we can just create a thread and execute them, eliminating the concept of thread pools.

# Back-end

**[References]**

- Asynchronous I/O - Wikipedia
  A form of input/output processing that permits other processing to continue before the transmission has finished
- Semaphore (programming) - Wikipedia
  A variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system.
- Non-blocking algorithm - Wikipedia
- Controller action return types in ASP.NET Core web API | Microsoft Learn
  Implementation reference for synchronous and asynchronous controller methods in Dot Net with C#
- Ktor
  Ktor is a Kotlin framework that allow developers to write asynchronous clients and servers applications, in Kotlin
- https://docs.spring.io/spring-boot/docs/2.0.9.RELEASE/reference/htmlsingle
  Spring Boot Reference Guide. Contains server default configuration.
- https://howtodoinjava.com/spring-boot/spring-async-completablefuture
  Code sample to create asynchronous methods in the Spring framework with the help of `@Async` and `@EnableAsync` annotations that use a thread pool on top of Java ExecutorService framework.
- Guide to Spring 5 WebFlux | Baeldung
  Code sample to create fully asynchronous and non-blocking application built on event-loop execution model, specifically Java Reactor Framework using Spring WebFlux.
- Project Reactor - Learn
  Reactor is a foundational library building for reactive fast data applications on the JVM. Used in Spring WebFlux, RxJava and many more.
- Non-Blocking Spring Boot with Kotlin Coroutines | Baeldung on Kotlin
- GitHub - JetBrains/Exposed: Kotlin SQL Framework
- R2DBC
- net.logstash.logback.marker java code examples | Tabnine
- Embracing Virtual Threads

# Back-end-Arch-006 - Event Schemas for an Event Driven Architecture

**[What is it]**

Event schemas are about having a binding contract (with format and structure) between producers and subscribers of events in an event-driven architecture (EDA). Event Schemas can be defined in various schema-driven formats like Avro, ProtoBuf or plain JSON.

A schema Registry acts as a shared repository of schemas that allows applications to flexibly interact with each other while adhering to a common contract. A super set of Schema registries is an Event Catalog. Catalogs help with event schemas management and metadata/documentation for a large system.

**[Where have we used it]**

We use it in the Falabella account with GCP pub/sub.

**[Why did we take this choice]**

Event driven architectures decouple the producer and consumer of events, but that means the consumers may be oblivious to the producers and any upstream changes to the event structure can go unnoticed.  Unlike REST APIs that enforce contracts and testing, this is a relatively new practice for events.

It also acts as an executable documentation for the Events within a system. Tests could be failed in case contracts are breached. Schemas also support schema evolution (through schema revisions & versions) so that both producers and consumers can evolve at different rates.

# Back-end

**[How did we do it]**

We used the native GCP pub/sub feature of event schemas.  All major pub/sub and streaming systems support schema validations.

Schemas help developers to do a fixed schema validation on the message before it is inserted into a Pub/Sub topic. If schema compliance check passes, it gets published to the topic else it won't make it to the topic and is rejected.

Schemas can be created  using the Google Cloud console, the gcloud CLI, the Pub/Sub API, or the Cloud Client Libraries.  Once a schema is created, it can be associated with multiple topics for  reusability. All major pub/sub systems support schema validations.

**[What were the learnings and challenges]**

For an existing event driven system, it's a learning curve and takes a lot of coordination to implement event schemas across multiple systems and services. For e.g. if there are 10 subscribers to a given schema topic, and a schema has to be enforced, it takes time and effort to coordinate it so failures can be avoided.

**[References]**

GCP - Create and manage schemas | Cloud Pub/Sub Documentation

# Back-end-Arch-007 - Asynchronous Event Pushing for reporting in microservices

**[What is it]**

In the microservices world, the services are contained within a bounded context and have individual  databases within that context. While this works well for individual services it poses a major challenge when it comes to reporting. Reporting generally requires aggregating data across microservices and to do that efficiently without impacting the performance is the key. While the fastest and easiest way to achieve this would be to aggregate the data by pulling the data from individual databases or individual microservices, these are not the efficient way. What we can do is to follow an event-based push model to pump data into the reporting service database. While event based pushing comes with its own complexities, it results in a highly performant reporting service with timeliness of data.

**[Where have we used it]**

We have used the event based pushing for reporting at GXS account, a digital bank based out of Singapore

**[Why did we take this choice]**

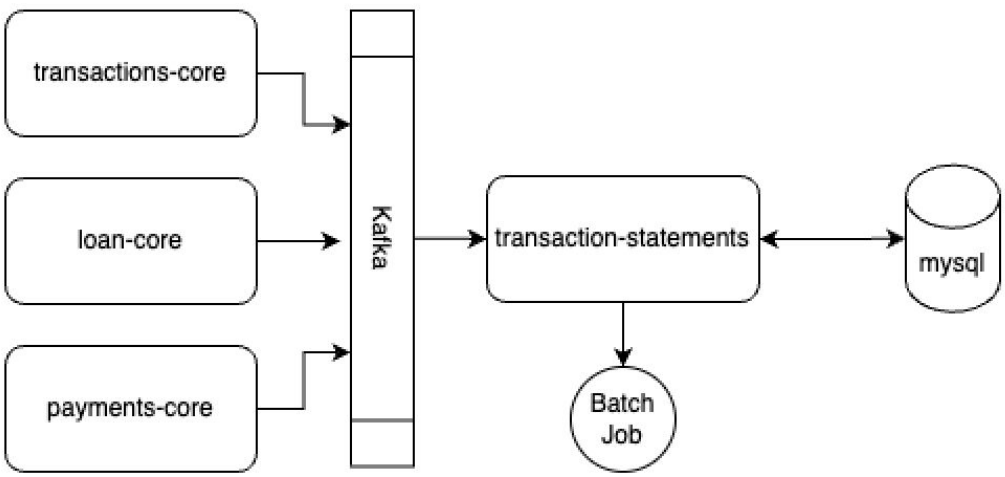
Event based pushing in the context of reporting has the following advantages:-

- **Asynchronous processing -** Non blocking communications. The individual microservices emit the events which are asynchronously processed by the reporting service resulting in loose coupling.
- **Reporting data format** - Even though individual microservices store the data in the format that is suited for them, the reporting service can process the events and store data in the format that is suited for reporting.
- **High Performance -** Since the reporting service has its own database that has data stored in a format that is best suited for reporting, this results in better performance compared to database pull or http pull.
- **Maintains timeliness of data -** Since the individual services emit the events and is immediately processed by the reporting services, timeliness of the data is good.

- **Maintain bounded context -** Since the reporting service is not accessing the individual microservices database directly, it maintains the bounded context.

**[How did we do it]**

In our account, the requirement wrt reporting was to generate transaction-statements for all the users every month. The transaction-statement should include data such as details of the loan, opening and ending balance, the transactions that have happened within the account and the payment details. We were dependent on three major services such as transactions-core which emitted transaction specific events, loan-core which emitted loan specific events and payment-core which emitted payment specific events. These events were processed and transformed by the transaction-statements service and are stored in the mysql DB. We used Kafka for event based communication and golang based microservices. We also made use of json columns in the database to increase our performance. Eventually a month end batch job will run, which generates transaction-statements for individual users. Since we are not relying on external API calls and all the data resides in our own database, we were able to achieve the performance that we desire.



**[What were the learnings and challenges]**

- Spend enough time to design the database for reporting databases instead of blindly copying the individual services schema. This would ensure that the data is stored in the format that is best suited for statement generation.
- Comparatively complex to implement when compared to other approaches such as db pull or API fetch.
- If the events are emitted by different teams, aligning on a contract and modifying the contract when new use cases come in is time consuming.
- Latency in the reporting database can cause some minor issues with respect to data loss.
- The reporting database usually will be very large, and is generally the log of events occurring in sequence. Requires separate data engineering team to handle this. There could be several layers of database each having its own level of granularity. For a statement to run for a particular month, appropriate timestamp in the where clause is a must.
- The batch job for statement generation is generally run an hour past midnight to give enough time for all events to be available. This ensures timeliness of the data.

**[References]**

[4. Reach-in Reporting AntiPattern - Microservices AntiPatterns and Pitfalls [Book]]

# Code and code-level design

## Back-end-Code-001 - Code structure and design validation as automated Architecture fitness function

**[What is it]**

Architecture fitness test is a concept where we validate the codebase structure to make sure we are not deviating from the convention. This validation can be manual or automated, but we recommend automating. The tests are designed to check various aspects of the software's architecture, including its design, dependencies between components, and naming conventions. Once the automated process is available, it can be run in CI/CD pipeline to detect and rectify architectural problems at an early stage in the development lifecycle and ensure everyone is following the agreed code convention.

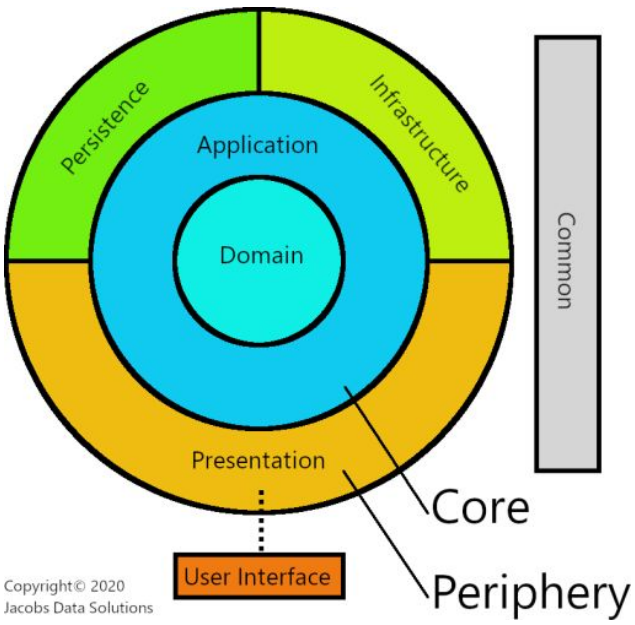
**[Where have we used it]**

- **SPACE**: Within the Space Modernisation Team, we've incorporated Architecture Fitness Tests into our microservice template. This integration serves to verify that our design principles, dependencies among the different layers of our Domain-Driven Design (DDD) architecture, and naming conventions align with the standards we aim to uphold.

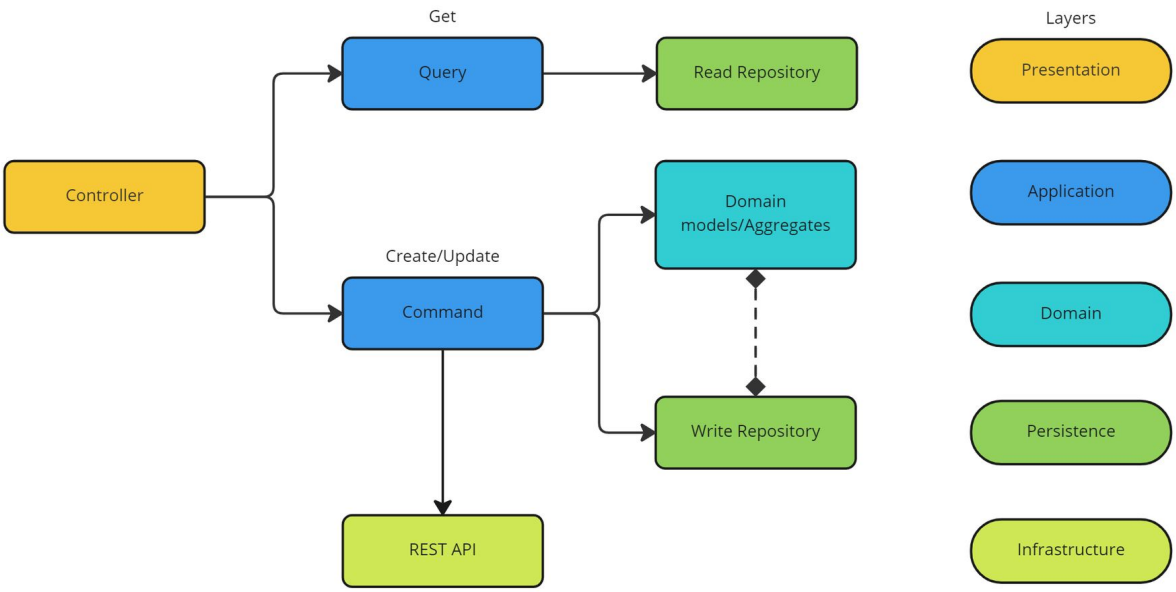
- **OTTO**: The Dragonfish team in the OTTO account uses this concept. We make use of DDD with Hexagonal patterns. This concept helped us to define rules related to the communication between domain and adapters, class/method naming conventions. We also validate the dependencies on the third party libraries within the domain, immutability, cyclic dependencies etc.

**[Why did we take this choice]**

The **SPACE** codebase follows the structure given in the next page.




Let's take a deeper look at the architecture and their segregation:




Our application follows a design paradigm that combines Clean Architecture with Domain-Driven Design (DDD), "Clean DDD." We've also incorporated elements from CQRS, which involves breaking down Application Services into smaller, operation-level classes represented as commands and queries. This architectural approach aims to untangle complex interdependencies between layers and establish well-defined domain entities.

The role of Architecture Fitness Tests in our development process is to identify violations of these architectural rules and assist developers in rectifying them. For example, our architecture dictates that the API/Controller layer must always rely on the mediator (Mediatr) to communicate with the application layer. The controller should never establish direct communication with the domain, infrastructure, or persistence layers. These dependencies are examined and identified using Architecture Fitness Tests.

**OTTO**:
This technique helped us to maintain the code convention without manual validation from experienced developers/architects. Given that it automatically validates the changes as part of CI/CD pipeline, everyone is able to make changes directly to the main branch and achieve the business goal faster. It also helped to have common understanding between every developer irrespective of experience in the team. If due to some business/technical decision there are some changes that need to be made in the code design, it helps to understand the impact for the existing codebase.

**[How did we do it]**

**SPACE**:

We have implemented Architecture Fitness Tests in .NET using NetArchTest library. This library allows you to create tests that enforce conventions for class design, naming and dependency in .Net codebases.

Dependency Rules Checked by our Architecture Fitness Tests:

1. Controllers should not depend on Persistence layer
2. Controllers should not depend on Domain layer
3. Controllers should not depend on Infrastructure layer
4. Controllers should depend on Mediator
5. Domain should not depend on any layer
6. Api should not depend on ApplicationLayer DTOs
7. Api should not depend on Domain
8. Application should not depend on Persistence layer
9. Application should not depend on Infrastructure layer

Naming Conventions Checked by our Architecture Fitness Tests:

1. Controllers should have name ending with Controller
2. Controllers should inherit from base controller
3. Requests should have name ending with Command or Query

4. Request Handlers should have name ending with CommandHandler or QueryHandler
5. Repositories should have name ending with Repository
6. RepositoryInterfaces should have name ending with Repository
7. ConsumedEvents should have name ending with Command
8. ProducedEvents should have name ending with Event

*Example Tests:*

```
public void Controllers_ShouldNotDependOnPersistence()
{
    TestResult result = ShouldNotDependOn( layer: ControllersNamespace,  params dependencies: PersistenceLayer);

    Assert.True(result.IsSuccessful,  userMessage: result.GetFailingTypeNames());
}
```

```
public void Controllers_ShouldNotDependOnDomain()
{
    TestResult result = ShouldNotDependOn( layer: ControllersNamespace,  params dependencies: DomainLayer);

    Assert.True(result.IsSuccessful,  userMessage: result.GetFailingTypeNames());
}
```

```
public void Controllers_ShouldDependOnMediatr()
{
    TestResult result = ShouldDependOn( layer: ControllersNamespace,  params dependencies: MediatrNamespace);

    Assert.True(result.IsSuccessful,  userMessage: result.GetFailingTypeNames());
}
```

**OTTO**:

The Dragonfish team made use of ArchUnit as Java is the primary language but similar tools are available for other languages.
Refer this GitHub gist for example architecture fitness test.

**[What were the learnings and challenges]**

1. It helps improve the developer efficiency in understanding and writing code, time on code reviews are saved.
2. This helped onboarding multiple teams(6) quickly and made sure every team adhered to agreed design and conventions.
3. One primary advantage of architecture fitness tests is their ability to detect architectural issues early, facilitating the maintenance of a clean and well-structured codebase from the outset.
4. Fitness tests ensure consistent enforcement of best practices and coding standards, aligning the architecture with predefined guidelines across the team therefore helping with maintainability..
5. Reducing technical debt becomes more attainable with fitness tests identifying architectural violations early, simplifying future feature additions and modifications.
6. It is integrated in the CI pipeline which prevents deployment on failure of tests that break the architectural rules.
7. The tests themselves serve as a form of documentation. By looking at the fitness tests, developers can understand the intended architectural patterns and rules.
8. Once the tests are available, it works like an Architecture Decision Record to understand the agreed conventions even if the initial team members are not available
9. Mostly initial one time investment, later just need to modify/update rules with the architecture change
10. Architects / Tech Leads doesn’t need to manually validate everyone's code changes once the tests are available due to that they can actually focus on things which helps in the business delivery

In summary, architecture fitness tests can be a valuable tool in a .NET microservice project, promoting architectural consistency and early issue detection. However, they also come with some overhead, both in terms of setup and maintenance, and may require some adjustment to the development workflow. Ultimately, the benefits of improved code quality and adherence to architectural best practices can outweigh these challenges.

**[References]**

1. https://medium.com/@suman.maity112/architecture-fitness-test-6f64d9a266c
2. ArchUnit User Guide

# Back-end-Code-002 - Monorepos for back-end repos as a sweetspot between cohesiveness and flexibility

**[What is it]**

A mono-repository is a version-controlled repository that contains many related but independent projects. These can be managed by the same team or multiple teams.

**[Where have we used it]**

We have used monorepo concepts in the OTTO-Dragonfish team to maintain terraform modules and common java modules which can be reused across other microservices.

Few teams (within Tesco) rely on this approach - where the monorepo doesn't mean one repository for the entire org, rather a repo with a highly cohesive services/modules within a team (eg. ordering)

**[Why did we take this choice]**

At **Otto**, Initially we had separate repositories for every small common code (like terraform modules, common java lib etc). Over time we observed we had to maintain lots of repositories each having very less number files, therefore less lines of code. When we wanted some sort of patch management (dependency update) or any small change, we had to go to each and every repository and make the changes. This process was very time consuming therefore we wanted to move towards Monorepo for small lib/modules. Note: We still have separate repositories for each micro service.

Talking about the rationale at **Tesco**: Consider a scenario where a group of microservices update the single order document/record throughout the customer journey - eg. from order creation services to confirmation services with various services involved in the flow. Due to its domain rich cohesiveness we can group and place them into the same repository along with the technical common frameworks supporting it.

In summary, several advantages can be realized by grouping related modules together:

- It reduces overall maintenance effort
- It provides more readability
- Refactoring effort becomes easy

# Back-end

**[How did we do it]**

Moving from separate repositories for terraform modules and common libraries to Monorepos was easy at Otto. Basically, we had to create a new GitHub repository and copy/paste the existing code to it. For our case, we have monorepo based on code/technology type, for example, we have 1 monorepo for terraform modules, another one for java libraries, another one for test suites (like, End-to-End test, load tests etc) and so on.

Given that one repository contains code for multiple modules, the CI process becomes a bit complex. In the Dragonfish team, we are not using any tools - rather we're using our own scripts so that depending on the changes, we figure out which module needs to be built/publish a new version. Here is the sample Github workflow used at Otto.

We follow a similar workflow for Terraform modules monorepo as well.

For a classic tech stack - say Java with springboot:  A maven/gradle multi module approach serves the implementation of mono-repo. The situation outlined in the article here is very similar to how the implementation is done. Working with a Multi-module based application in a modern IDE like Intellij becomes handy, and lots of build automation tools are available supporting the multi-module build corresponding to the languages chosen. Eg. Maven (java specific) and gradle (diverse set of languages)

**[What were the learnings and challenges]**

- Common agreement is needed within team when deciding the monorepo structure
- Depends on the technology, there might be tools available which can help to maintain monorepo better
- It's better to find ways to optimize overall build time in monorepo, for example, run checks only for the changed modules otherwise the pipeline will run for longer. At Tesco, we use maven build cache extension to reduce the overall build time for the sub modules.
- Pipeline script will become bit complex
- Might become a bit overwhelming for the new joiners depending on the monorepo structure/group

**[References]**

- Is it the era of mono-repo?. What exactly is mono-repo? | by Suman Maity | Medium
- Tools
  - Lerna - Can be used to maintain JS monorepo, which will link and try to reuse the node modules. By doing this it reduces the overall build time
- Monorepo with Java, Maven and GitHub Actions, including basic example - DEV Community
- Curly Braces #1: Java and a project monorepo

# Back-end-Code-003 -Observability as code

**[What is it]**

Observability as code is the practice of using code to automate the creation and management of observability infrastructure. This includes everything from setting up alerts to creating dashboards and visualizations. This is not to be confused with Infrastructure-as-code, the more widely known technique of provisioning infrastructure such as compute resources, storage resources, gateways, security rules etc. using code.

**[Where have we used it]**

We have used this technique in Otto (Boxfish and Turtle teams) and Axis Maximus.

**[Why did we take this choice]**

- It makes configuration of dashboards and alerts repeatable and automatable instead of manually configuring from scratch across environments and for new microservices.
- Consistency across environments and across microservices, in the way that dashboards are set up, metrics that are tracked and how monitoring and alerting works.
- Configuration and management of observability assets like alert threshold, duration and configuration values help us in avoiding configuration mistakes, errors or overlooks during deployment. Reviewing changes can be done easier on code instead of visual dashboards where configuration parameters might be tucked away underneath the UI.
- As the scripts are automated and consistently maintained, the same can be used for trailing disaster recovery drills.

# Back-end

**[How did we do it]**

Axis uses Grafana and Prometheus for observability and all monitoring performance of applications. We use Helm to deploy grafana and prometheus (and everything else) in all environments. The helm charts, along with all the grafana dashboards (using dashboard json files), prometheus configs are committed in bitbucket. Whenever the automated pipeline runs, it takes the dashboards definitions and prometheus configs from bitbucket and applies them to the environment. This way, we can keep the dashboards in sync with the code.

When we had to update the grafana instance recently, it was easy to recreate the same previous dashboards thanks to this practice.

The alerting and monitoring infrastructure at Otto Boxfish and Turtle teams include the following AWS capabilities:

- Cloudwatch logs and custom alarms
- Lambda that gets triggered from SNS
- Event rules in EventBridge
- Grafana dashboards that help visualize technical and business metrics from Prometheus
- Quartz jobs

Terraform is used to create and configure all the above.

**[What were the learnings and challenges]**

- When using observability as code, make sure the code is compatible with the version of the observability tool. For example, when using grafana visualizations as code, test the code with a new version before upgrading.
- Prometheus server was initially set-up, which stores data for a 15 days span. We were not using any tools for keeping this data safe for a longer period, hence our dashboards did not provide correct insights. After integrating with AWS Prometheus as service, we are now able to store data for 150 days (which can be configured to more).

# Back-end-Code-004 - Rich Domain models over anemic domain models

**[What is it]**

The **Domain Model** is a central component of domain driven design. In an Object oriented paradigm, the Domain Model should include entities and value objects which not just carry the state but also the associated business logic thereby making the Domain Model rich. Incidentally, this also aligns with the **Encapsulation** principle which states that **state** and **behavior** should exist together. This is in contrast where domain model holds only state (aka anemic domain model) and business behaviour resides in utility classes, services, etc.

**[Where have we used it]**

At **Otto Lionfish,** we refactored our anemic domain classes in the legacy codebase. We've also been following this principle at **John Deere.**

**[Why did we take this choice]**

1. Multiple methods/business logic related to the domain model/class got scattered across the project. Hence the number of tests increased.
2. Changes to the model required extensive commits and code alterations. Changes to business logic should ideally not require touching a large number of areas of the codebase to avoid introducing bugs.
3. Overuse of static methods, limiting access control.
4. Anemic Domain Models scattered business logic across services leading to unawareness that a particular behavior for the model is available somewhere and led to duplication of code as the behavior was required by other features.
5. Updates to the business logic were sometimes out of sync b/w the multiple code copies scattered across services.
6. The end goal was to move our services to DDD and richer domain models being key features of DDD, it helped in resolving above mentioned issues.
7. Having a richer Domain Model helps to achieve layered architecture by keeping the business logic concern separate from application, ui and infrastructure concerns. This in turn allows improved maintainability and extensibility of the code.

**[How did we do it]**

With tools like IntelliJ, we efficiently identified all domain model usages throughout the project. While actively developing new features, we initiated code refactoring to eliminate the signs of anemic domain models, relocating relevant functions into the domain model classes.

We have been providing reference material on creating rich Domain Models while onboarding new joinees. In addition, we also covered this topic in our team learning session to spread awareness. We also emphasize this topic with Tech Leads of our teams. We share reference material on creating Rich domain models with new joinees onboarding the account. We also covered the topic in our team learning session.

**[What were the learnings and challenges]**

- **Otto Lionfish:** At the Lionfish team we had to work on some legacy code and a couple of new microservices. The plan was to pitch in Domain Driven Design(DDD) for development. For gathering the requirements related to the new services we were regularly in touch with clients. The outcome of these connects was to build the domain for the new service. Once the domain with entities, value objects, attributes, policies, related behaviors etc was ready. The clients also got an awareness of what DDD is. The scope was very clear and when the implementation of Domain driven design was done. We were able to build Richer domain models just by following DDD principles. This gave us an opportunity to pitch in DDD for refactoring the legacy codebase for existing microservices.

- **John Deere:** While the topic of Rich vs Anemic domain model is quite popular, not all developers have the same understanding of the topic. In addition, very few of them actually practice it. Hence, it's better to not assume that devs know and socialize the topic of rich domain models broadly and frequently.

**[References]**

- AnemicDomainModel
- Anemic vs. Rich Domain Objects | Baeldung

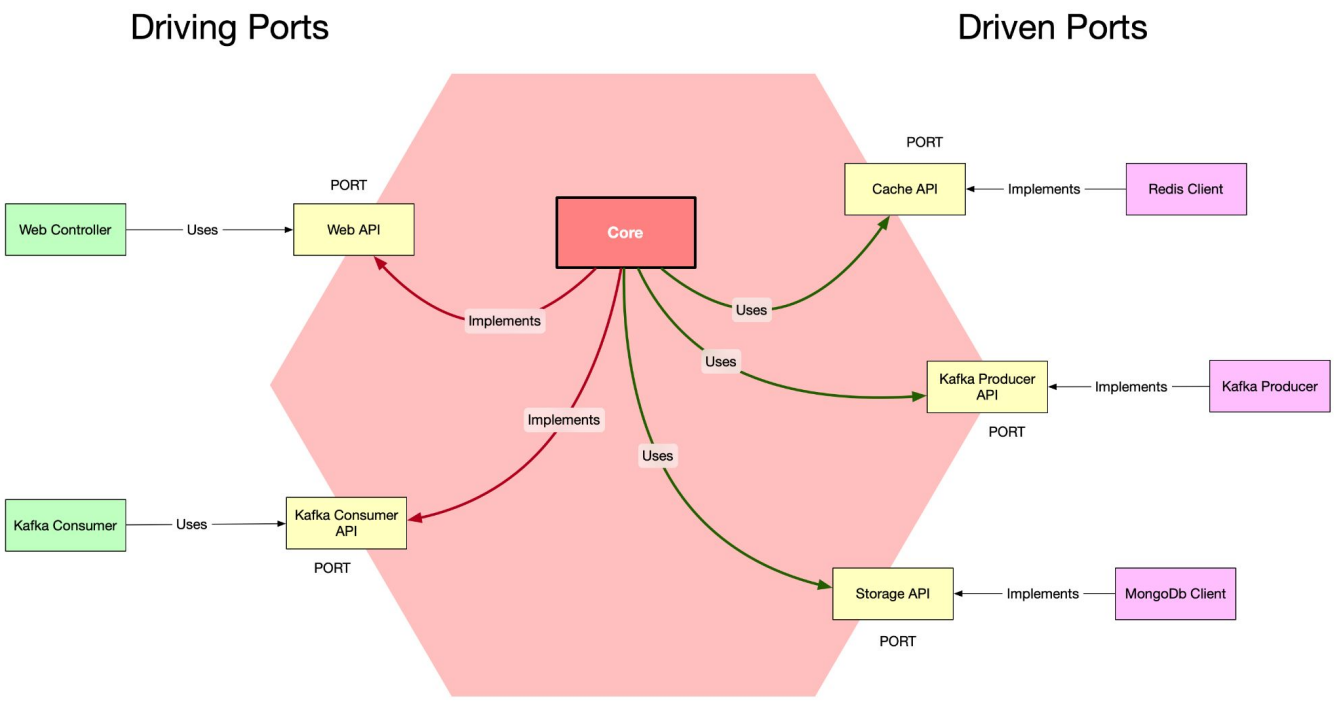
# Back-end-Code-005 - Ports and Adapters (Hexagonal) Architecture for backend

**[What is it]**

Ports and Adapters architecture (aka Hexagonal Architecture) is a more evolved and modular version of the traditional layered (3-tier) architecture.

There are three components of this architecture. Ports, Adapters and the Core. Ports can be thought of as the entry and exit points of the application. In general terms, they are the interfaces the app exposes to the outside world. Adapters are the implementations of these interfaces. Core is the heart of the application. It contains the business logic and the domain model. Core uses the ports to interact with the outside world but does not know anything about the adapters (implementations).

The application is built by assembling the core with the adapters. The adapters are injected into the core, typically using dependency injection. Ports are of two types. i.e. Driving (Input) port and Driven (Output) ports. Driving ports are used by the outside world to interact with the application. eg. Web request handlers, Kafka consumers etc. Driven ports are used by the application to interact with the outside world. eg. Database clients, Kafka producers etc.



**[Where have we used it]**

In Axis Bank's Maximus project, we have used this architecture for a backend service. The Platform Engineering Team developed this service.
This technique has also been used in the Otto and ParentPay accounts.

**[Why did we take this choice]**

At **Axis bank,** the service we built was not a domain service but a tool to be used for testing. Its purpose was to serve as a mock server for all other real services for HTTP and Kafka interactions. The mocks were backed by a pact broker. This requirement was generic but complicated. There is no other tool which can do this.

Since the domain of this service is "meta", about http request, event messages, contracts, etc, we struggled initially to model the application. The complexity was high due to custom triggers and schedules. We searched about this problem and came across Hexagonal Architecture. We decided to give it a try. It worked for us. Ports and Adapters helped us in following ways:

1. **Unambiguous design of domain:** This architecture forces us to put domain in the center of everything. Everything depends on core and core depends on nothing. Because of this it became very easy for us to create the correct domain and its logic. Before implementing this architecture, whenever we wanted to make any change we had to think where would change need to go. After the architecture, this need for thinking completely disappeared.

2. **Increased testability:** Because every adapter (implementation) was made specifically for a port (interface), this made everything testable from day 1. Since everything comes together with the help of wiring all dependencies together, we could decide the integration degree of tests by simply choosing which real adapters to use and which to mock.

3. **Tamed complexity:** We felt that this architecture helped us tame the complexity of software by making code predictable and intuitive.

# Back-end

**OTTO account:**

- Hexagonal Architecture is a perfect pattern to achieve the clear abstraction of the domain code from the application code, so we can identify the domain logic with great ease and this would help us maintain a cleaner domain. The team followed Domain driven design, and this technique was in sync with DDD.
- Isolating the external communications using ports and adapters also enabled switching between different technologies for the external connections easily without disrupting or touching the domain code.
- It helps us to isolate the core, domain model and business logic from the external system and dependencies that are prone to change more frequently

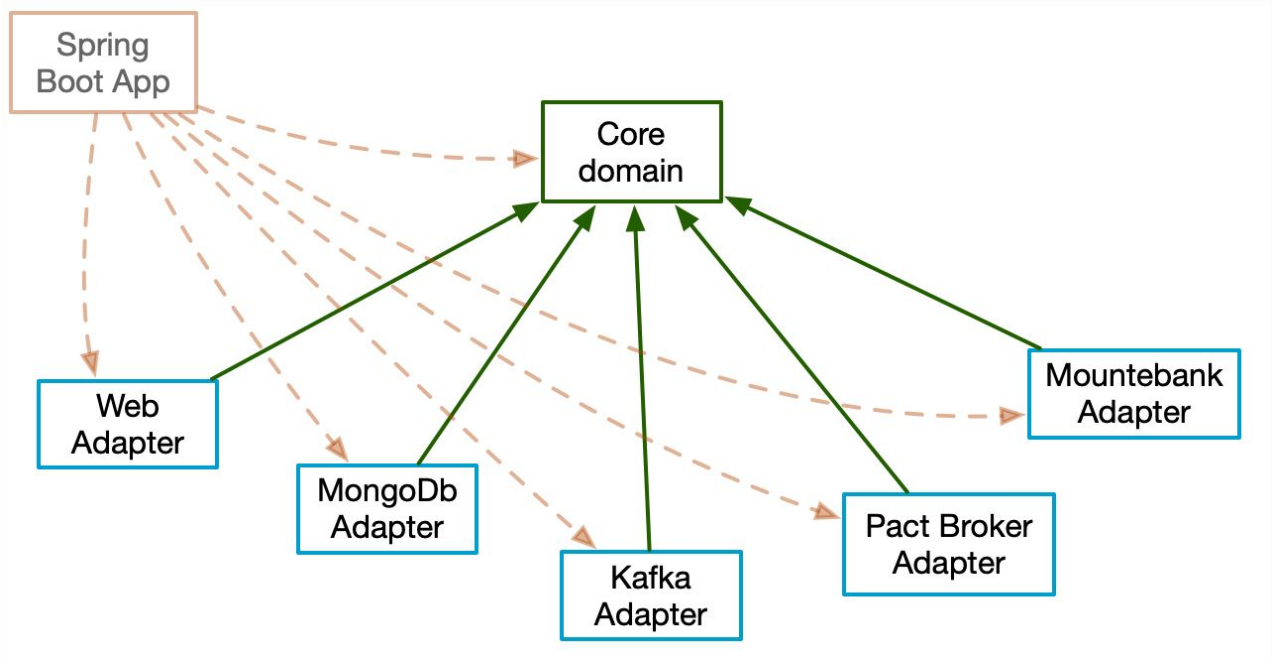
**ParentPay account:**

We had a web based centralized school management system where we were responsible to show a **Premium** feature for any student who the teacher or admin is viewing. Data shown varied based on location, role, languages etc. We used hexagonal architecture using .NET framework along with DDD principles to maintain projects for our backend for ease of development and loosely coupled modules. We had 4 projects namely: **Application**, **Domain**, **Infrastructure** and **Tests**

- Application took responsibility as an **adaptor** mapping the domain with infrastructure. This contains the api controllers and DTOs, etc.
- Domain contained the value objects, entities/aggregates and interfaces related to business values/logic.
- Infrastructure contained IN and OUT **ports** for communication with external services using db contexts, migration sql scripts, etc.
- Tests contained namely integration and unit tests for good code quality and coverage.

**[How did we do it]**

**Axis Bank:**




This was a simple Kotlin spring webflux application. The key was to make every adapter as a separate Gradle submodule. Core was a separate module on which every adapter depended for ports. Using submodules helps a lot because as opposed to packages, submodules require explicit dependency specification.

Spring boot application is a separate module which is mostly empty. It does the job of wiring all components together using constructor injection and spawning the server.

**Otto account Drangonfish Team:**

We made use of the basic Java interface while designing the core/domain part. The domain also contains the main business logic. The adapters package implements the interfaces present in the domain packages. We made use of the Spring dependency injection concept to autowired all the required port/adapters.

**[What were the learnings and challenges]**

1. Interestingly, core is not always free of all dependencies. Some dependencies may be necessary depending on the tech stack. For example, in our case, all the interfaces returned Mono<T> instead of T. The dependencies in core, if any, should be very fundamental in nature.
2. Might be a bit difficult for new joiners to understand the concept but here fitness functions help a lot. Introduce automated fitness functions to respect the Hexagonal pattern conventions
3. This might be needed in a service where frequent changes are expected in the ecosystem. Otherwise, hexagonal pattern would increase the effort to add new functionalities and interfaces

**[References]**

- Hexagonal Architecture with Java and Spring
- https://github.com/otto-de/gitactionboard/tree/main/backend/src/main/java/de/otto/platform/gitactionboard

# Testing

## Back-end-Testing-001 - Leverage testcontainers for tests that involve databases, other services or even cloud components

**[What is it]**

Instead of DB-access components like DAOs or repository classes test against a mock object or a real database, use testcontainers to spin up light-weight and fresh database instances for each test run. Also leverage testcontainers to spin up service dependencies (microservices, external services or even cloud components) quickly and efficiently for local runs or testing.

**[Where have we used it]**

- In **Exinity**, we migrated persistence layer tests from using mocks into using testcontainers. We also used testcontainers for the database while running integration tests.
- In **Kroger**, service integration tests use testcontainers for databases as well as service dependencies
- In **SIAM Makro**, we used testcontainers in integration tests that interface with cloud services like AWS SQS

**[Why did we take this choice]**

Test containers hit the sweet-spot between brittle mocks and dedicated dependencies (services/databases/cloud components)

- Testing the DB access components like DAOs or Repository classes with mocks doesn't add much value and is too much dependent on the underlying persistence libraries - say Hibernate for Java.
- A dedicated database test server (either for persistence layer testing or for integration testing involving databases) is too much of a maintenance overhead and also will become cumbersome in the long run as the data created during multiple runs need to be cleaned up. Besides, database availability and infeasibility for parallel runs also becomes a problem.

- When performing integration tests with several subsystems and third party services, testing against a constantly running service instance is often not practical or cost-effective.
- Using mocks of the service client library object is brittle. It could break with a slightly different usage of the client and with minor upgrades of the client library as we are dependent on the implementation details.
- Using in-memory solutions like testcontainers yield faster test suite run time as the database/interfacing service is local to the developer laptop or the CI server.

**[How did we do it]**

Exinity and Kroger used the postgres testcontainer for their persistence layer tests as well as integration tests. Siam Makro used testcontainers of cloud services like AWS SQL using LocalStack and for interfacing with services such as TypeSense.

A great deal of test tooling is also possible to simplify tests. Initializing testcontainers sometimes requires a bunch of boilerplate code. One of the accounts created a tooling where the developer could just add an annotation @DbTest over the test class, which would effectively run the boilerplate initialization needed for the test class to work with the database testcontainer.

**[What were the learnings and challenges]**

- There are a number of readily available Testcontainers which can be just configured for use
- Custom Testcontainers can also be created from any docker image. This was used in the Kroger account to create containers of the other microservices on which the system under test depended upon.
- The postgres db container initialisation did just take about 20 seconds in the local laptop and it could be shared across the entire test suite of thousands of cases. Of course this is a trade-off between speed and flexibility of having parallel tests. Refer to the implementation for singleton testcontainers.
- Once the container is up, the tests against the databases did not take more than a few milliseconds. We ran over 400 cases against the db in about 2-3 seconds.
- Test containers support Colima, so there is no dependency on a Docker desktop license.

# Back-end

- One challenge that we can face in the CI server is if the CI node running the tests is already a docker container running as part of a Kubernetes cluster. We end up in a “Docker-in-Docker” scenario. For instance, we faced this when using a Jenkins setup with kubernetes cluster. In this case, the tests won't run without additional setup. Using one of the methods described here would help.

**[References]**

1. Reference implementation in Catalyst

# Back-end-Testing-002 - Bi-Directional contract testing for frictionless testing of distributed systems

**[What is it]**

Contract testing preempts services having mismatched contracts with other services from getting deployed. Bi-Directional contract testing takes this further and leverages existing OpenAPI provider contracts, helps manage the workflow and continuous deployment of tests.

**[Where have we used it]**

Within the **Space** Modernisation account, we've established bi-directional contract testing through PactFlow in 6 streams.

**[Why did we take this choice]**

As the account expanded and numerous diverse streams operating across interdependent services and events were introduced, the practice of contract testing became crucial. Contract testing ensures the harmonious transmission of messages between providers and consumers. Besides, the scope of end-to-end testing can be reduced.

However, there is additional complexity and a barrier for entry because teams have to hand-craft solutions to integrate contract testing with CI/CD workflows and make sure pre-deployment and post-deployment steps are done for each environment. For example, when Kroger tried integrating contract testing with their CI/CD pipelines, deployment to every environment had to be sandwiched between additional steps which made the CI/CD workflows long and unwieldy.

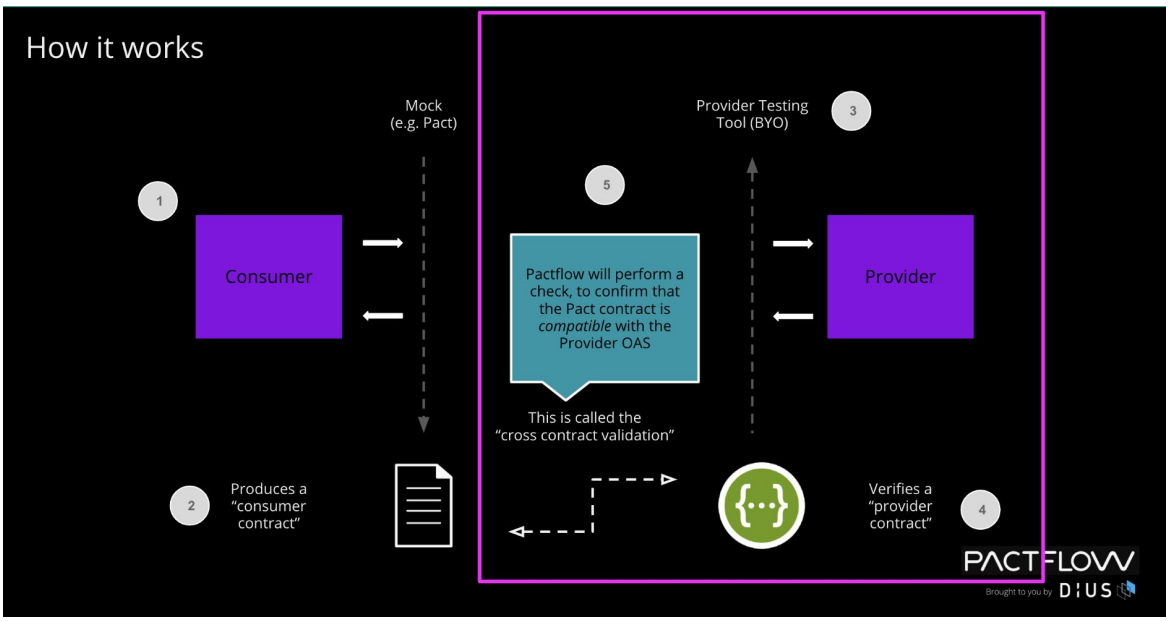
Bi-Directional Contract Testing expedites this procedure by leveraging the existing OpenAPI Specifications (OAS) as the provider contract, thus necessitating pact tests solely on the consumer side. Unlike in the case of Consumer-driven Contract testing (CDC), contracts can be derived from mock setups in existing integration tests.  Most of the contract testing tools provide only consumer driven contract testing, but Pactflow provides bi-directional contract testing support, hence it was picked. Using Pactflow, we were able to perform contract testing for both APIs and kafka events. It can also be used against external APIs.

# Back-end

**[How did we do it]**

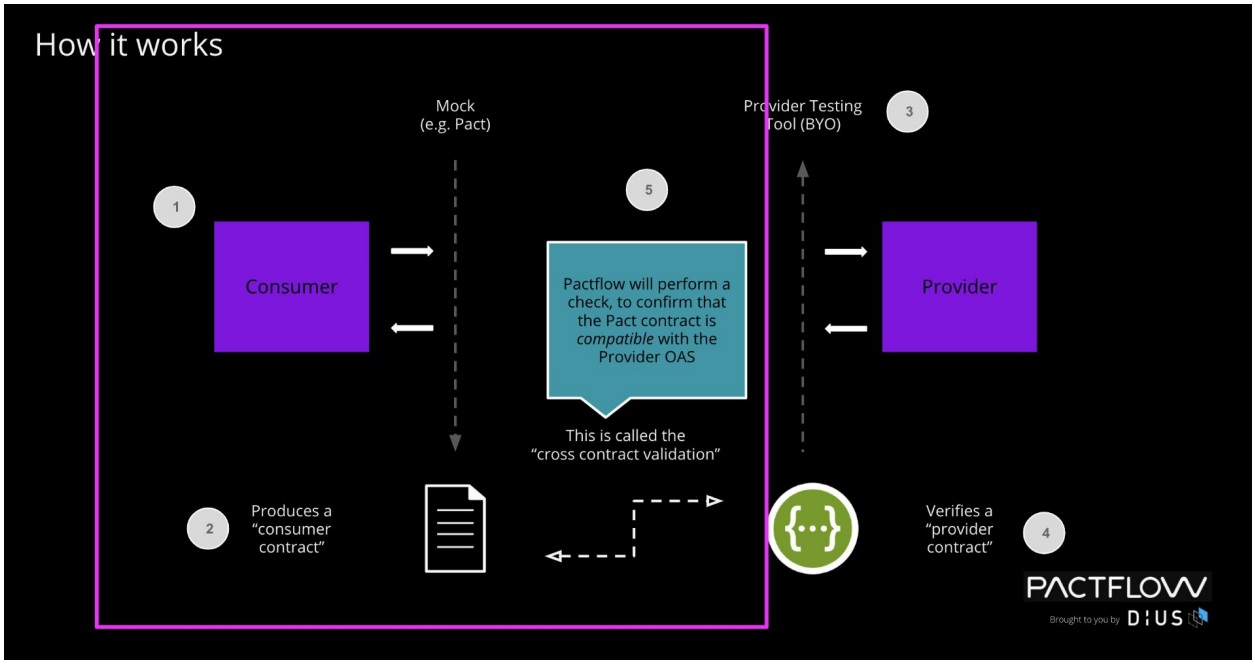
Pactflow was used to implement bi-directional contract testing.

*On the Provider Service:*




1. Provider starts with its specification (for example, Swagger - an OpenAPI specification) called the Provider Contract.
2. The Provider Contract is tested against the provider by generating verification results of component tests (via Swashbuckle).
3. The Provider Contract is uploaded to PactFlow.
4. When we call can-i-deploy in the CD pipeline the cross-contract validation process in PactFlow generates a Verification Result ensuring the provider doesn't break any of its consumers.
5. If that passes, we deploy the provider and record the deployment via the pact-broker record-deployment command.

*On the Consumer Service:*




1. Consumer tests its behaviour against a mock (say PactNet).
2. The Consumer Contract is produced in the form of a pact file that captures only the actual interactions generated by the consumer code.
3. The Consumer Contract is uploaded to PactFlow.
4. When we call can-i-deploy the cross-contract validation process in PactFlow generates a Verification Result determining if the consumer consumes a valid subset of the provider contract.
5. If that passes, we deploy the consumer and record the deployment via the pact-broker record-deployment command.

**[What were the learnings and challenges]**

- Pactflow supports both API contract testing and asynchronous messaging (kafka testing).
- User-friendly dashboard to manage integrations, check compatibility and view mismatches in between contracts.
- Pactflow is compatible with pact consumer driven contract testing and additionally provides bi-directional contract testing.

# Back-end

- Can-I-Deploy and its benefits like detailed information about what failed, why and fixing bugs faster with this information.
- Regardless of having many environments, we can manage and view the list of deployment environments used by our teams.
- Pactflow can be used as self-hosted deployment or fully-managed service. (In the case of Space, it is a fully-managed service)
- Restrict system access to Pactflow with user roles and permissions
- Web hooks and secret management is also available.
- Use tools such as Cypress, Postman and Wiremock to write contracts
- Contract Testing for kafka events is still in development for .NET.
- Even experienced developers take time to understand the end-to-end workflow of contract testing. Comparatively end to end tests are easier to understand.

**[References]**


1. More Features of PactFlow

# Back-end-Testing-003 - Test transaction headers

**[What is it]**

Test transaction headers are attributes that are part of the contracts of all the enterprise REST APIs and events. For example, when an API receives the attribute “testTransaction” as a header in the request, it treats the request as a test that the system is undergoing. When an event handler receives an event with an attribute “testTransaction”, it understands that the event is a consequence of a test that the system is undergoing and changes its behavior accordingly.

**[Where have we used it]**

- TESCO - Order Fulfillment System, Transport, Customer Order(in all the domain teams)
- OTTO - test tag is used heavily with in otto account

**[Why did we take this choice]**

There are some scenarios where it is important for tests to not affect or corrupt actual production data  in any way.

- Load testing: At Tesco, load tests are performed regularly in production, during off business hours, to simulate a large number of users or transactions to test the performance of the system. This is done to ensure that the system can handle peak loads, such as during Christmas, or any special events such as The King’s Coronation. It is important to test in production to get the most accurate results.

- Onboarding test stores for the whoosh proposition(https://www.tesco.com/zones/whoosh): This is to minimize the changes of failures when the store actually goes live in order to support same day quick deliveries. The tests are important because the systems integrate with third party logistics for order delivery and due to various reasons, not all the integrations could be tested in a pre-production environment.

# Back-end

The test orders also need to be canceled/archived/deleted. The test transaction header was quite useful for all the domain teams to dynamically change their behavior during tests and cancel the orders once the tests were completed. These were also used to control the number of requests sent to third party fulfillers so as to not bombard the external systems with a huge number of requests.

**[How did we do it]**

All the domain teams in the Tesco account agreed to have a “testTransaction” attribute as part of the APIs/event contracts and persist this attribute as part of the domain entity to their data store.

For example, when running load tests, test orders are created with the testTransaction attribute set to true. This attribute is persisted along with the order entity to the data store. The REST API that is responsible for creating an order accepts the testTransaction attribute as a header parameter as in,

```
@Post(produces = MediaType.APPLICATION_JSON, consumes =
MediaType.APPLICATION_JSON)
@MetricCategory(ScribeEndPoints.CREATE_ORDER)
@CustomCounterMetrics(endpoint = ScribeEndPoints.CREATE_ORDER)
public Mono<HttpResponse<?>> createNewOrder(
@Body CreateOrderApiRequest orderRequest,
@Header(name = HttpHeaders.AUTHORIZATION) String authorization,
//All the other headers
@Header(name = RestHeaders.TEST_TRANSACTION, defaultValue =
StringUtils.FALSE)
Boolean testTransaction) {}
```

# Back-end

Once an order is created and persisted, the OrderCreated domain event that is published contains the testTransaction attribute as well, as in,

```
{
  "id": "@{PREFIXED_UUID}",
  //Other high-level attributes,
  "payload": {
    //Other payload attributes,
    "testTransaction": false,
    "revision": "@{ANY_NUMBER}"
  }
}
```

Depending on the use cases, the domain services implement appropriate mechanisms to clean up the test orders after a while. For example, the Transport Slot Reservation API provides an endpoint to delete all the Reservation slots that were created as a consequence of the load test.

```
@RequestMapping(value = "/cancel-test-reservations", method = PUT)
  public void cancelTestReservations() {
    LOGGER.info("Triggered the job to cancel test reservations -
asynchronously");
    scheduledTasksService.cancelTestReservations(correlationId().value());
  }
```

In OTTO, we have common test tags. Depending on the test tag, downstream teams can decide whether they want to process the event or skip it. For example, we have the common test tags shown in the table on the next page.

| | |
|---|---|
| PerformanceTest | **Automatically** created test events with **big amount of data** |
| SystemTest | **Automatically** created test events to test specific business cases **within one team** |
| End2EndTest | **Automatically** created test events to test specific business cases **within multiple teams** |
| ManualTest | **Manually** created test events |

There are simple attributes in the event contract. Also, the same information is available in the SQS/SNS message header. Here's a sample event:

```
{
  "eventId": "cc519a17-bb2e-4f24-8986-e87ae5fdf084",
  //Other event attributes,
  "test": {
    "scope": "dragonfish",
    "type": "ManualTest"
  },
  "data": {
    //Payload attributes,
  }
}
```

It's now up to the consumer team to filter or process the data. Filtering can be done as part of SNS subscription level or the team can consume the event and filter it in the backend code base.

Additionally, one team will pass the test tags from the source team to its consumers.
For example, if there are 3 teams, A, B and C, then team B will pass the test tag to team C if the event from team A has test tags.

# Back-end

**[What were the learnings and challenges]**

- In some of the domain use-case handlers, based on the test headers, introducing minor bifurcation was a challenge.
- Consistent naming convention always helps, API documentation is a must. Having a developer portal and providing other teams access to play around with the APIs was really helpful
- Names of test tags needs to be commonly agreed upon with in the account
- The test tag helped the team to have independent load/manual testing without any dependency on other teams.
- It also reduced the overall testing time when a specific feature needs to be tested as part of a specific team.
- During load testing, it doesn't create unnecessary burden on consumer teams, they can decide whether they want to be part of the load test or not
- If test tag is not part of the event structure from the beginning, certain teams might take longer time to implement it due to other priorities

# Back-end-Test-004 -Shareable API collections to ease testing with multiple scenarios

**[What is it]**

API collections are widely used by developers when they are working on developing APIs. It helps them create various scenarios to test APIs and showcase API capabilities to team or API consumers. But more often than not, these collections remain on developer machines. Some tools try to solve this problem by allowing developers to share their collection with the team. Postman is a notable example of this which created this model as a service.

**[Where have we used it]**

We have used this practice in several accounts such as Caterpillar and First Student.

**[Why did we take this choice]**

API collections are an effective way to showcase various API capabilities. This is often used in desk-checks when your stories are only about creating backend APIs. Making these API collections sharable enables QA teams to quickly start testing since most of the scenarios are covered in those collections. The QA team can build upon these collections and add additional scenarios to it. This can also become the base of automated integration / functional tests where frameworks allow it.

**[How did we do it]**

We have been using Postman as the de facto tool for this for a long time. Postman provides a Newman tool which can be used to automate execution of collections in certain order using command line. This was useful in using collections in automated integration / functional tests which can be executed in CI pipelines. Postman recently dropped support for scratchpad (a way to create and export Postman collections, and check them into source control for sharing with other team members), forcing teams to upload collections to their servers.

Intellij has its own support in the form of a plug-in called `http-client`. It comes bundled with intellij-ultimate edition and easy way to treat api collections as first class citizens in your api repositories. These collections can be executed with IDE itself and from command line as well with additional binaries from intellij. This can help using the same collection in automated tests in CI pipelines.

Other alternatives - Auxl, Insomnia, Bruno, HTTPie

**[What were the learnings and challenges]**

1. Some tools import files in the app before you can use it to send requests. For those apps, when you make changes in the collection, you have to export it back to your file system so you can check-in it back to the repository. Many people miss this step resulting in not sharing those changes back to other members.
2. Make sure that API keys, secrets etc. are not exported into collections and re-imported back to source control. Treat secrets as environment variables instead.

# Back-end-Test-005 -Mutation tests to assess the quality of tests

**[What is it]**

Mutation testing (or mutation analysis or program mutation) is used to design new software tests and evaluate the quality of existing software tests. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant.

**[Where have we used it]**

Mutation testing is used at Otto Dragonfish and Atlassian, and is being evaluated at IDFC.

**[Why did we take this choice]**

Mutation tests helped us figure out the quality of our unit tests. It also finds out the gaps in our tests automatically. It improved the overall test coverage threshold by helping us add the required valid tests. Even while developing the code, mutation testing helped to write better test cases. Thus, there is very early feedback for the tests.

**[How did we do it]**

**Pitest:** In Java + (Gradle/maven), we just have to add PITest as dependency and rest is taken care automatically. The setup for the same is very easy as it is just a config addition to the project and the rest falls into place. Pitest is quite mature and very flexible as a tool, has a good amount of features covered like excluding packages, classes & methods with patterns, applying different types of mutators according to the need and playing around with threads. We can configure the mutators, include/exclude classes, coverage threshold, delta analysis etc using a maven/gradle job config.

**go-mutesting:** This is the tool we are trying out at IDFC for performing the mutation testing in Golang language, the tool is not mature enough as PITest yet, but does the minimum job.

**[What were the learnings and challenges]**

- Time consuming in terms of run time, but if tweaked properly then it can be very effective
- Limit the mutation operators used based on the project environment
- Limit the number of mutations per class
- Tweak the number of threads
- Target only specific portions(Business logic/ service) of your codebase if it is very large or have monolith/legacy structure
    - Don’t run this with every commit as it takes time to run these test....may be as smoke test
- Configure PITest to run the test with history enabled, which will only target the delta for the testing
- A lot of documentation has to be read in order to understand what a particular tool can and cannot deliver at first, which is a bit time consuming for newbies to the concept and tool.
- One will have to come up with the right config / strategy for their project if not it may consume a lot of time and cost might go up due to high usages of the cpu cores on the cloud billing aspect

**[References]**

- **Java:**
    - Blog on Mutation testing using PITest(Java)
    - GeekNight 82nd Edition - Mutation Testing
    - PIT Mutation Testing
- **Golang:**
    - https://github.com/avito-tech/go-mutesting
- Mutation testing tools available in the market
- The Top 23 Mutation Testing Open Source Projects

# Developer Experience

## Back-end-DevExperience-001 - Service templates and starter kits for consistency and quicker bootstrapping

**[What is it]**

Service templates and starter kits are pre-made frameworks or codebases that can be used as a starting point for developing microservices. They typically include the necessary files and folders, on top of a basic framework that can be customized to meet the specific needs of the application.

In the context of clean architecture, service templates and starter kits can be used to enforce the separation of concerns and promote loose coupling. This is because they typically divide the application into different layers, such as the presentation layer, the domain layer, and the infrastructure layer. Each layer is responsible for a specific task, and there is no coupling between the layers.

By using service templates and starter kits, developers can save time and effort, improve the quality of their code, and make their applications more maintainable.

**[Where have we used it]**

REED(.Net core), TESCO(Spring boot), SPACE

**[Why did we take this choice]**

At REED, we faced the below problems,

1. Inconsistent microservices development - Each team had its own way of developing microservices. This made inner-sourcing hard. Teams found it hard to develop, write tests, and raise pull requests on codebases owned by other teams.

2. Unable to maintain microservices - Over a period of time it really became hard to maintain the microservices under the different  domains because they lacked consistency in their design and naming conventions.

# Back-end

At Tesco, other than the aforesaid problems we additionally faced these challenges:

1. Complex microservices architecture - The microservices did not have well-defined structure and had a large number of components scattered across different places, with code violating SOLID principles. Development was hard and the cognitive complexity was very high.
2. Unable to scale microservices - It also became hard to break down the microservice based on the use cases, and scale them separately, based on the production load, because the use cases had code that were interdependent and were tightly coupled.

**[How did we do it]**

At REED, we created a service template project using the guidelines defined here:

- Clean Architecture from Uncle Bob
- .NET Clean architecture implementation

On a high level, the service template had the below separation of concerns,

- WebUI
- Domain
- Application
- Infrastructure
- Startup (This is required only for .NET core projects where the dependency injection needs to be defined and other bootstrapping needs to be taken care of. This project would reference all other projects.)

The service template also had fitness tests using ArchUnit. The architecture decisions were managed using ADRTools.

SPACE account used cookiecutter templating to create the project structure., the implementation details and features were as follows:

1. Microservice project structure with ready to use CQRS pattern
2. Integration tests, pact contract test files, Architecture tests to see if Clean architecture is followed or not
3. An Infra repository to follow the GitOps methodology with all the helm charts without needing to change anything once created
4. Integration of Infra project with ArgoCD
5. Ready to use Github actions without needing to change anything
6. Hooks to scan secrets in the repo, follow specific commit message patterns & to check if build and test passes before committing

# Back-end

At JohnDeere, we have the test strategy implemented and documented in our account starter kit repo which can be referred by teams for implementation details.

**[What were the learnings and challenges]**

1. Choosing the right template or starter kit: There were a lot of options available, but we wanted to keep things as simple as possible. There are various templates with different kinds of implementations, but we decided to stick to the Onion architecture.
2. Customizing the template or starter kit: Once the team started development, there were a lot of proposed customisations. These requests had to be reviewed and understood so that there were no changes in the kit/template that were not standard and hard to understand. For example, a proposal to introduce a “common” folder was rejected in favor of duplicating code across layers.
3. Governance: At Tesco, this is still evolving at the moment. However at REED, the governance framework was set up nicely:

    a. Repository for the service template and the associated starter kits: This was initially setup by the technical architect at REED in collaboration with the technical lead/a representative from each team. This ensured that the base framework was aligned with the organization’s standards and policies.
    b. Provide training on the starter kits/templates: Sessions were conducted during the tech council, teams were provided an overview of how to incorporate these templates onto existing projects or new projects. The kits themselves had very good documentation.
    c. Monitor the use of the starter kits/templates:  In order to ensure that these templates were adopted by the respective teams, backlog items were created on each of the team’s boards and planned on the roadmap. The responsibility of monitoring the usage of these templates was then upon the technical leads on each project. The teams provided important feedback that further helped in the refinement of these kits.
    d. Establish a process for approving changes to starter kits/templates: This was quite important because of the inner-sourcing model that we followed, any developer could raise a pull request with a detailed suggestion on why and how a change could help, which would then be reviewed by the leads/architects and approved/rejected.
    e. Define the roles and responsibilities: A RACI matrix was created with definition of the persona and their responsibilities in this governance process.

**[References]**


1. clean architecture of uncle Bob
2. GitHub - jasontaylordev/CleanArchitecture: Clean Architecture Solution Template for ASP.NET Core
3. GitHub - npryce/adr-tools: Command-line tools for working with Architecture Decision Records

# Back-end-DevExperience-002 - Component stubs and verified component stubs for faster, predictable development and testing

**[What is it]**

Stubs are placeholders - simplified yet controllable implementations used in both development and testing. During development, local setup can be greatly simplified using stubs. In the case of testing, stubs not only help with simplification but also for covering different scenarios that are often hard to reliably reproduce using live dependencies.

In short, stubs are often drop-in replacements during local development and testing as they simulate the behavior of real components or services without having to invoke the real functionality.

However, stubs can go stale over time and thus tend to give false positives. Verified stubs combine the best of both worlds.

**[Where have we used it]**

In **Mercedes-Benz OTR** we have used stubs in the Custamigos Team.

**AxisBank Maximus** platform is implementing the technique of verified stubs in an attempt to make end-to-end testing faster and more reliable.

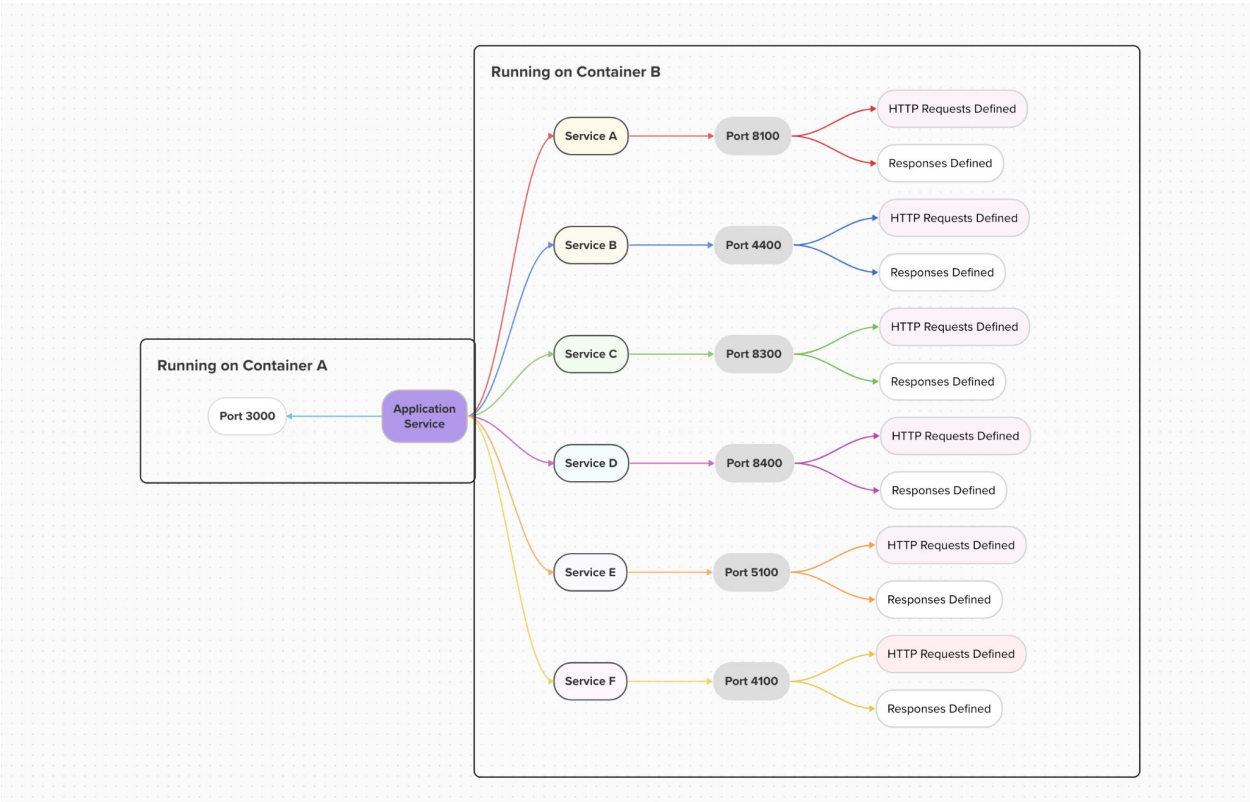
**[Why did we take this choice]**

1. Stubs help in controlling the behavior of certain components or services to create controlled testing scenarios. For example, for an integration testing scenario you may want a dependency to behave in a particular way, i.e. return a particular response in certain cases so that you can test the system-under-test (SUT)'s handling of that special case. This gives you fine-grained control over different scenarios and edge cases.
2. Stubs are particularly helpful when coding in local-setup, wherein you are not interacting with other services within the project and external as well - some of which are not yet fully implemented or are flaky.

3. E2E environment deployments need all services to be up, require manual coordination between QAs of all teams and so on. Besides, the E2E tests themselves are time-consuming, unreliable/flaky and hard to set up. Some of these concerns have been addressed by contract testing. Yet, there was scope to push tests lower on the pyramid.

**[How did we do it]**

At Mercedes-Benz, we have stubs for internal services(services owned by the project) as well as external services. Our Team owns two services - one in Clojure and one in Kotlin. The Clojure Service uses MounteBank and the Kotlin service uses WireMock for the implementation.




As depicted in the above diagram, the application interacts with numerous services and is stubbing the HTTP request calls to these services. When the requests to a port matches a particular defined request, the defined mapped response is sent out.

We use this in our local setup and also in our dev environment. It is useful in local setup because *we can* simulate the behavior of real components or services without actually invoking the real functionality. It is also helpful in testing out error scenarios like getting **Request Timed Out** error which otherwise becomes very difficult to simulate.

Since our dev environment is totally on stubs, the tests running on dev like integration tests, contract tests, functional tests become less flaky and more reliable.

However, mountebank stubs can go stale over time and thus tend to give false positives. To counter the drift issue between stubs and the real services, at Axis, we use the technique of **verified mocks**. A custom **Pact-Stub-Service** has been built over mountebank and pact broker. Internals of the implementation of this service can be found here. It only returns responses from verified pacts, thus giving a guarantee that the response from this service is indeed similar to an actual service. It works not just with HTTP but also with Kafka events. Thus, we generate up-to-date stubs from verified contracts of dependencies (i.e. “providers” as per the contract-testing lingo).

We point our subject service to Pact-Stub-Service for all its dependencies and the tests run faster, consistently and with much less resources. This makes it possible to run end to end tests on local machines which was not possible before.

**[What were the learnings and challenges]**

1. Contract testing since the beginning of an account and introducing contract testing in a mature account are two completely different problems. For example, when a 5 year mature product decides to adopt contract testing, it comes with a big backlog of existing 100s of interactions for which contracts need to be written. That requires effort and budget and prioritization.
2. To use verified stubs, contracts must be written and verified instead of manually setting up stubs.
3. Writing completely manual and verbose contracts takes a longer time. DSLs and abstractions to reduce boilerplate to write contracts help a lot.

**[References]**

1. Mock it with Mountebank !. When it comes to mocking services... | by Priyanka Jagga | DigIO Australia | Medium
2. Kotlin Solutions | WireMock

# Resilience and Stability

## Back-end-Resilience-001 - Resilience and fault-tolerance for service providers and service consumers

**[What is it]**

Distributed platforms inherently come with the problem of unavailability of different components at any point of time owing to network failure / timeouts / resource outage etc. Resilience-improving measures such as circuit breakers and rate limiters help make a system fault tolerant.

Fault tolerance is the property that enables a software system to continue operating properly in the event of failure of some of its components; whereas, resiliency refers to the capability of the system that defines how efficiently it can get back to its normal state from the erroneous state.

**[Where have we used it]**

**AllState** and **Kroger** use resilience4j for **circuit breakers**.

**Axis** uses a **custom rate limiter**. The subject for this rate limiter is not users but products like Personal Loan, Home Loan, Credit Card, etc. Each of these products have their own orchestrator services and they all call the same domain services. The rate limiter is deployed on these common services.

**Otto** uses a cloud solution (AWS WAF) for rate limiting.

**[Why did we take this choice]**

### The case for circuit breakers

Kroger and AllState own and run several microservices. If a service dependency (downstream service) is known to have a high chance of failure due to unanticipated reasons - say it takes too long to respond -  it might be pointless for a service to continually do resource-intensive retry operations.

Instead, the component should accept that the operation has failed and handle this failure gracefully. Forwarding requests to slow components can increase the chance of a **cascading failure** of the system, and hence the use of circuit breakers to mitigate this issue.

## The case for rate limiters

In the Axis account, domain services in Axis are common for all product orchestrators. If a product:

I. Faces sudden demand, OR
II. Is playing a long overdue retry queue, OR
III. Faces an attack

the common services will come under **extreme load**. The impact of one product now starts to spread across the entire platform because that service is underperforming. Rate limiters are therefore applied at the provider-side (in this case, the domain services) so that they don't get hammered with an onslaught of requests.

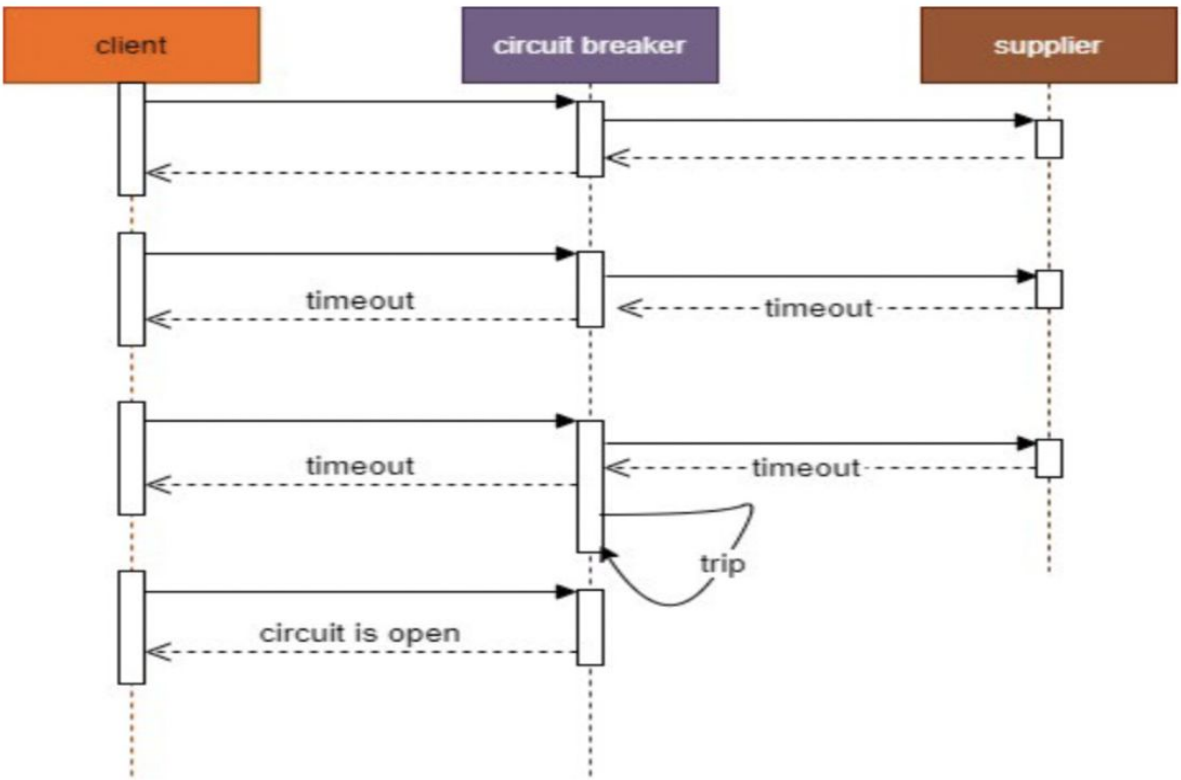
## The case for custom rate limiters

Off-the-shelf rate limiters can help apply a fixed number of requests per time unit or a percentage of requests distribution across all clients. However, these options lock down the resources for a product, even when that product is not under load while other products are, thereby wasting precious resources that could have otherwise been repurposed.

**[How did we do it]**

## Circuit breaker

The diagram on the next page illustrates the working of a circuit breaker. Failed responses from the supplier are monitored by the circuit breaker, and eventually requests do not even get forwarded to the supplier for a specified time - so that the faulty supplier gets time to "cool off" and recover. Thus, when many services are dependent on the faulty supplier, we can avoid cascading failures.




S**pring cloud circuit breaker** provides an abstraction layer across different circuit breaker implementations. Netflix now recommends using **Resilence4j**, which supports Java 8 and functional programming. Netflix Hystrix is no longer in active development, and is currently in maintenance mode.

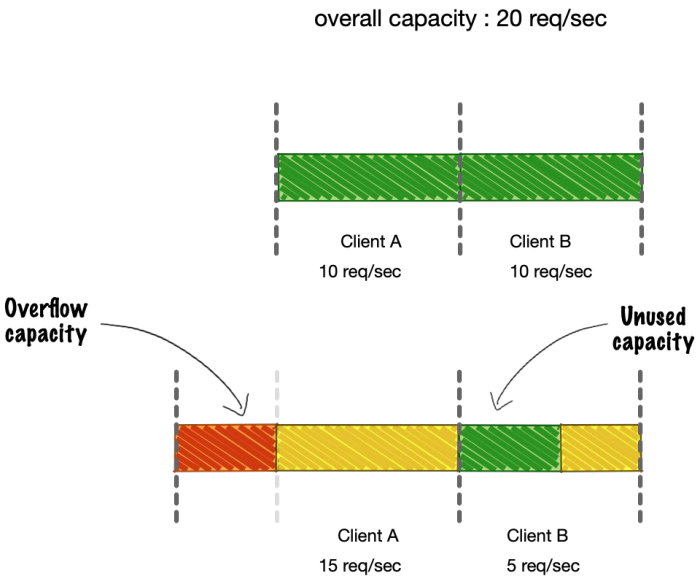
## Rate limiting

In Otto account, all the APIs exposed via API gateway are attached to AWS WAF with custom rules to rate limit the calls. This is done in order to avoid DDOS attacks.

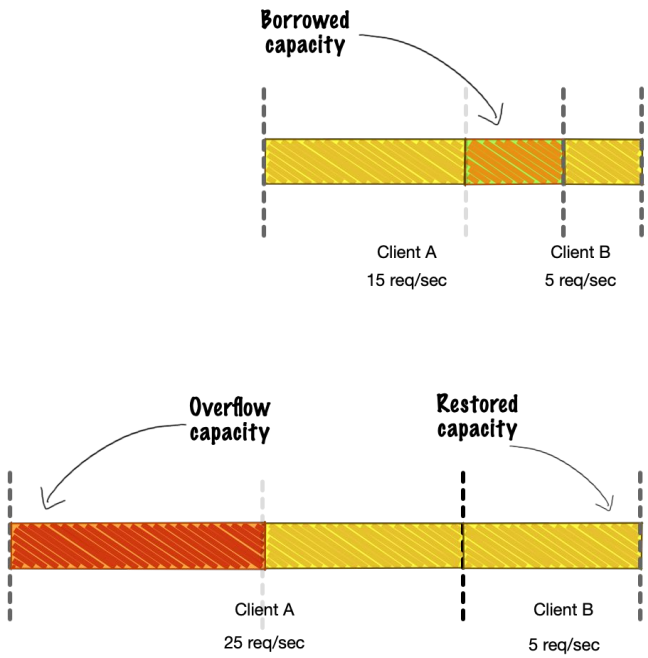
In Axis, the rate limiter is deployed as a sidecar in Kubernetes. We used Scala and the ZIO ecosystem. It uses functional programming which helps manage concurrency easily.

# Back-end

The rate limiter uses a fixed window with our modifications. Everything is the same as the usual fixed window algorithm except capacity distribution logic per window. The total capacity of a service remains constant. However, the distribution of capacity is different for each client in each window. It uses statistics from the last window to calculate the capacity distribution for the next window.




Let's take an example of 2 clients (for simplicity). Client A and Client B. By default the rate limiter starts with an equal capacity distribution for both clients. That means if service capacity is 20 req/sec, each client will get 10 req/sec for the first second cycle (or window). If B underutilized its capacity, say only 5 req/sec, in this window, that will be distributed to other resource-hungry clients in the next window. The distribution proportion is based on the demands of clients in need.




Once the underutilized client(s) B need their capacity back, they will get it back in the next window. This essentially means that if client A starts hitting 25 reqs / sec and client B is hitting 10 req / sec, B will not be impacted by A's demand. Without this logic, A would have taken up almost all the capacity and B would have been starved.

# Back-end

The custom rate limiter was used as a kubernetes sidecar. It does not use any storage and is completely in memory. Rate limiting is done per pod and relies on kubernetes to do fair load balancing. We exposed custom prometheus metrics and created grafana dashboards to observe and understand behavior in production. We made changes in our internal library to send client id in each request as a header, so that clients can be identified.

**[What were the learnings and challenges]**

- One application could communicate with multiple applications. The delay / stability of one downstream application could be different from another downstream application (supplier). Hence the circuit breaker configuration for both of these downstreams may have to be different. A one-size-fits-all circuit breaker configuration may not be the best choice.
- By default all exceptions from downstream application (supplier) are considered as failure resulting in the decision to open the circuit and ignoring the next subsequent flows. Now, based on the application nature, the supplier could return business exceptions as well which is valid for some flows / requests. These types of exceptions should be ignored and considered as success through circuit breaker configuration. In other words, occurrence of business exceptions should not unnecessarily deem the system as unhealthy and open the circuit.

- Benchmarking each service to know its capacity regularly is required to keep the rate limiting effective
- Functional programming is a clear winner for concurrent applications. ZIO ecosystem is good for experimentative functional programming projects where risk appetite is slightly higher. As this was a sidecar implementation, we could potentially change the technology if needed. The functional learning curve with ZIO is smaller compared to the other libraries.
- Rate limiting not only helps improve resilience. It can also be used for when API is monetised and you only want to give limited access to users based on their payment plans. In a multi-tenanted setup, custom rate limiting like the one done in Axis can help in an equitable distribution of resources and control blast radius to one client in the event of that client making an unexpected number of requests.

# Back-end

**[References]**

**Circuit Breaker:**
Martin Fowler on CircuitBreaker
CircuitBreaker in resilience4j

**Rate limiter:**
Request rate limiting - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced

# Back-end-Resilience-002 - Reliable retries for improving resilience

**[What is it]**

Retrying an operation is a common requirement in any system today due to the distributed nature of the systems. Service dependencies - internal or external - could be unavailable, temporarily down, overloaded or slow to respond. In such cases, we typically retry the operations.

Simple in-memory retry mechanisms can be sometimes sufficient. However sometimes, critical business workflows require more robust persistent retry mechanisms. A reliable retry mechanism would need to be:

- **Persistent,** so that retry attempts are not lost in case of system restarts or new deployments. Retry attempt durations can span across hours to days.
- **Backed by a DLQ (Dead Letter Queue),** so that failed attempts can be inspected and retried or modified manually if needed
- **Asynchronous,** so that the user is not needed to wait on the screen for the operation to complete
- **Observable,** so that we can monitor the retry attempts and failures and take corrective actions if needed

**[Where have we used it]**

- **Axis Bank's Maximus** project - We have a Loan Origination System which has many user journeys defined. Each user journey has many steps. Some steps are user steps and some are system steps. The journey typically has a customer step which is to submit the application. This is followed by a series of system steps to disburse the loan.
- **Allstate project -** We work in the claim processing domain at Allstate. Based on the data-critical nature of the business workflow, reliability has been considered one of the most important architecture characteristics of the system. Once any workflow is initiated in the life cycle of claim processing, the source system sends the message / request to the integration component services (as the source and destination systems can’t communicate directly due to technical restrictions) and the integration component needs to process / transform the message before sending to the destination system.

- **FirstStudent** - We have a suite of applications for transporting special needs students to school and back to home after school . Our architecture is based on microservices and we wanted to improve the reliability of message exchanges between these services.

**[Why did we take this choice]**

At **Axis**, once a user submits the application and there are no more user steps left in the journey, the system steps are executed sequentially. The user then waits on the screen for loan disbursal: if an error occurs due to a downstream system unavailability, the user would see an error message and has to come back to retry the operation. This was undesirable for two reasons: user experience and potential loss of business. Users may decide to never come back and occurrence of even a single such case is a loss of millions for the client.

This retry mechanism has contributed to a significant amount of business for Axis bank.

**At Allstate,** we used the retry mechanism to make the system more reliable and sustainable with business demand.  Failures are inevitable in any distributed system. With retry mechanism in place and a few other fallback techniques, we have been able to ensure data is available to all the expected systems without fail.

At **FirstState**, we have service-based architecture as backend and services communicate with each other using kafka topics. We have implemented DLQ for messages which failed in consumption. These messages were inspected to make the system more robust, and do retries in special cases.

**[How did we do it]**

**In Axis**, we implemented an asynchronous retry mechanism. From an end-user perspective, if any failure is encountered during the execution of the last few system steps, we inform the user that the application has been accepted and will be processed in the next few hours to a couple of days instead of reporting an error to the user. Behind the scenes, retries are carried out. If the retry is successful, the application is moved to the next steps and eventually the user receives an SMS stating that loan has been disbursed (or rejected due to business reasons).

Let us look into the details of the implementation. Spring's native scheduling capabilities are used to schedule retry operations:

1. Consumer/handler A consumes a message from its Kafka topic T.
2. If consumption fails, A would simply add the message (its status and payload) to a mongo collection. The mongo collection serves as the retry queue for the whole of the loan processing domain.
3. A Spring-based application monitors M the records in the mongo collection at periodic intervals. It pushes the record's payload on the corresponding Kafka topic T.
4. Consumer/handler A again tries to consume the message from T:
    a. If it successfully consumes, it checks the mongo collection for the message (using the topic and message id combo) and soft-deletes it.
    b. If it fails, it would make sure that the message is added to the collection if not present already. If present, it perhaps increments the retry count.
5. Meanwhile, the monitoring service M would move the fully retried records into DLQ (as a sub-step of step 3) for manual inspection and resolution. There is a UI developed to inspect the DLQ and retry the messages manually.

In order to avoid race conditions, Axis uses shedlock over mongodb.

**AllState** leverages the Spring retry capabilities to mitigate the challenge of unreliable communication between different systems. Once the retries are exhausted the message is routed to a tracking service (more on this later) for further check and inspection. Along with that, using proper logging, monitoring and alerting through Splunk and Datadog we can mitigate failures.

A few implementation details of the Spring retry mechanism used at AllState:

- Add Spring Retry Dependency in dependency management
- Add the @EnableRetry annotation to the Spring Boot application class or configuration class

- Use @Retryable annotation around the block of code which needs to be retried for given number of times , having some delay in between the retries & with fallback implementation (annotated with @Recover) that will be executed if all retry attempts fail
- Fallback logic when retries exhausted
    a. Logging with proper message which will alert for immediate resolution
    b. Sending the message to DLQ for further inspection

Some points on the tracking functionality that is used when the retry count is exhausted. There are two key components to handle the edgecase of exhausted retries:

- Tracking utility - a library (jar) that any service can integrate with. It handles “exceptions” related to exhausted retries by putting these messages and their payloads into a JMS queue
- Tracking service - a service that periodically reads messages from the JMS queue and puts them into a SQLServer database. The service also exposes end-points to query these messages so that developers can troubleshoot and invoke the destination systems manually with these messages.

In **FirstStudent**, we are using kafka as a communication mechanism between services. We used spring kafka library to implement retries in consumers and push messages to DLQ if consumers failed to consume messages. The library also supports back-off retries with configurable retry period and count. We monitored logs to check on which exceptions messages are going to DLQ and adjusted our logic based on that data. We monitored DLQ actively to find out if messages can be retried after changing code.

**[What were the learnings and challenges]**

- It is very important to segregate retriable and non-retriable errors. Failing to do so might result in unnecessary operations being retried and DLQ getting piled up. It is important to keep refining these retry filters over time
- There is no replacement for a manual inspection of DLQ. We need dedicated people and ownership for this activity.

# Back-end-Resilience-003 - Service health-checks integrated with automated restarts

**[What is it]**

Continuous feedback, early failure detection and fault tolerance in distributed systems is a major challenge especially in an ecosystem that is continuously scaling. Service health checks, when tuned correctly, become helpful in these systems. The health checks also should be integrated with container orchestration systems like k8s to perform automatic remediation like restarts in case of failures.

**[Where have we used it]**

Account: JohnDeere
Project: Aftermarket Ecommerce project
Applications: all microservices deployed to k8s (~30)

**[Why did we take this choice]**

Implementing and configuring kubernetes liveness and readiness probes serves a crucial role in enhancing the reliability and stability of our applications. Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides **liveness probes** to detect and remedy such situations. This is more of a self-healing / auto-remediation to bring back the service to life.

Enabling **readiness probes** provides a seamless customer experience to end-users when they access an application by directing traffic to a service only when the application is ready to receive traffic. Determining WHEN an application is ready to receive traffic is based on how the probe is set up. Typically, a readiness probe helps with polling an endpoint that provides information about ensuring service is started, all integrations are established well and is in a state to accept incoming traffic.

**[How did we do it]**

In John Deere, we've implemented liveness and readiness probes as a standard practice for our Spring Boot applications in Kubernetes.

Liveness Probe

We've configured a liveness probe to ensure that our applications remain alive and responsive. This probe periodically sends an HTTP GET request to a specified endpoint, in our case, **/health**. It kicks in 30 seconds after the application starts, giving it time to initialize fully. Then, it continues checking the application's health status every 15 seconds. If the application fails to respond successfully to this probe for six consecutive times, it's considered unhealthy, and Kubernetes takes appropriate actions (Mostly restarts the Pods). In a Spring Boot application, the /health endpoint is typically exposed by Spring Actuator, which provides detailed information about the application's health.

Readiness Probe

The readiness probe is responsible for determining if the application is not only alive but also ready to serve user traffic. Similar to the liveness probe, it sends HTTP requests to /health, but it starts immediately after the application starts running. This probe also checks every 15 seconds. If the application responds affirmatively, it's marked as ready to receive traffic. If not, the probe continues checking until the application is ready..

Additionally, since we use an Application Load Balancer (ALB) Ingress, these probes are also leveraged by the ALB target groups to route traffic only to applications that are both alive and ready.

**Tools & Technologies**: k8s + helm

**Code:**

```
livenessProbe:
  httpGet:
    path: /health
    port: http
  initialDelaySeconds: 30
  periodSeconds: 15
  failureThreshold: 6
  timeoutSeconds: 10
  successThreshold: 1
```

```
readinessProbe:
  httpGet:
    path: /health
    port: http
  initialDelaySeconds: 30
  periodSeconds: 15
  failureThreshold: 6
  timeoutSeconds: 10
  successThreshold: 1
```

**[What were the learnings and challenges]**

1. The Helm template developed at JohnDeere account incorporates these probe configurations as sensible defaults. This ensures that every Spring Boot application we deploy using Helm uses these checks
2. Although we started our implementation of the /health endpoint by also checking the health of dependencies (like databases and downstream microservices), we realized that a single health ping would cascade across the system and multiple services would end up responding over and over again. Ultimately, our health check end-points were made to simply report the alive status and not do any sort of dependency checking within that.
3. For dependency checking, we have a /component-connectivity end-point for each service. The /component-connectivity end-point for service A will make sure its immediate dependencies are reachable. For eg; if service A depends on service B and an S3 bucket, service A's /component-connectivity end-point will hit the regular health check of service B and can access the S3 bucket (say via the HeadBucketAPI). The goal is to integrate this with our monitoring solution, although it has not been done so far.

**[References]**

- All about K8s Liveness & Readiness Probes
- K8s production best practices - health checks

# Security

## Back-end-Security-001 - Static Application Security Testing tools for DevSecOps

**[What is it]**

Static Application Security Testing (SAST) helps identify security vulnerabilities in the source that may make applications susceptible to attacks. This is considered as white box testing since these tools analyze source code.

There are different categories of analysis that are performed and some tools specialize in certain areas::

- Secret scanning tools are used as secret scanners and ensure that code that is pushed to a repository do not contain secrets that can be misused. Examples of tools are Talisman and Trufflehog.
- Static analysis tools for security scan the codebase for security hotspots, say if a *cookie secure flag* is not set. For example, SonarQube
- Dependency checkers and SCA tools check the dependencies used and report on any new security issues reported against those dependent libraries. For example, Dependabot. OWASP Dependency check reports vulnerable dependencies in applications as per OWASP vulnerability database(NVD)
- Container scanning tools help detect vulnerabilities in container images. For example, Trivy and AWS ECR vulnerability scanner
- Cloud security and compliance checks using wiz.io

Tools like Snyk and Trivy provide a more comprehensive coverage and can provide SAST, SCA and container scanning capabilities. Synopsis Polaris provides good integration patterns to have security scans as part of CI pipelines.

**[Where have we used it]**

In the **Kroger** account, we have used all the above mentioned tools in different ways. On **FirstStudent**, we are using Synopsis Polaris for doing SAST and SCA analysis.

# Back-end

On **Caterpillar**, we are using Fortify for doing SAST and DAST scans on application.
On **JohnDeere**, we are using Owasp Dependency Check, SonarQube, Talisman, Trivy
**Otto** uses SonarQube, Github advanced secret and vulnerability scanning, AWS vulnerability scanner, dependabot and wiz.io for cloud security checks.
**Space** uses Dependabot, SonarQube, Talisman, Trufflehog and Trivy

**[Why did we take this choice]**

The general reasoning behind using SAST tools is early feedback on security vulnerabilities - be it passwords or secrets in code, insecure coding practices or vulnerable dependencies.

Most of our projects use open source libraries to create our applications. We have to be careful about not adding security vulnerabilities in our application by using vulnerable open source libraries. Security scans in pipelines make sure that we take informed decisions about which libraries we can use and upgrade when necessary.

There are various tools in the market to accomplish this and tool choice should be made based on parameters like languages supported by tool, capabilities(SAST, DAST, SCA), IDE integrations and price. Some of the notable tools are Snyk, Synopsis, Fortify, Checkmarx, Veracode. All of these can be part of your toolchain and effectively integrated with your CI/CD pipelines.

**[How did we do it]**

Talisman as a pre-commit/pre-push hook on every developer's workspace. This helps in avoiding committing or pushing secrets to the repository.

Trufflehog scanning is part of the build pipeline and fails the pipeline when it discovers a secret that has been checked in.

Dependabot helps us to find and fix vulnerabilities in project dependencies.Source control repositories sometimes come with out-of-the-box integration with Dependabot. For instance, it has been integrated with a few accounts' Github instances and thus becomes available automatically for every repository in that Github instance. Once alerts are enabled, users would get notified on vulnerabilities, automatic pull-requests get created and so on. Space account enabled dependabot alerts, security updates and dependency updates on their Github instance.

# Back-end

Snyk has been made available to all teams to use in 2 ways.

1. Using the web application to import projects/repositories for scheduled scanning.

Integrating the Snyk CLI with the build pipeline to enable teams to address the vulnerabilities at an earlier stage and proactively keep the code secure.

Trivy scanning is part of the build pipeline and fails the pipeline when it discovers a vulnerability in base image

Dependency check is part of pipelines to give reports of vulnerable dependencies as per OWASP vulnerability database(NVD).

**[What were the learnings and challenges]**

- Talisman
  - Ensuring all developers have installed the pre-commit/push hook.
  - On multi-vendor teams, this is harder to implement and generally, the only way is to wait for a failure to crop up and then ask for enforcement.
  - Ensuring that Talisman does not fail due to false positives.
- Trufflehog
  - On shared repositories, it is a challenge to convince the code owners to include Trufflehog on the pipeline. The fear of impacting the build times adversely has been one of the biggest roadblocks.
- Snyk
  - Keeping track of the numbers on the Snyk Dashboard and ensuring that numbers don't creep up required constant monitoring. This was before the integration of the Snyk CLI with the pipeline.
  - Dedicating time in a Sprint to solve for Snyk reported issues has been a challenge. The client product managers treat these as they do with tech debts and are generally hesitant to include such stories in a Sprint.
  - Some of the issues reported by Snyk cannot be solved due to other dependencies. For example, for an application that is embedded within a larger application, we cannot upgrade a library since the new version would be incompatible with the Angular version being used.
- Dependency Check
  - Making sure that reports are reviewed regularly and configuring the build to break if CVE and CVSS score goes beyond a certain threshold.

**[References]**

1. CHECKLIST - Secure Delivery Checklist

# Back-end-Security-002 - DAST for automated penetration testing

**[What is it]**

Dynamic Application Security Testing (DAST) is a type of security testing that focuses on identifying vulnerabilities and security weaknesses in an application while it's running. DAST involves actively interacting with the application to simulate real-world attacks and assess its security posture.

**[Where have we used it]**

We've incorporated the OWASP ZAP scan into the Space Modernisation Account, scheduling regular runs(once daily) within the development and pre-production environments. These scans generate reports containing alerts in accordance with the predefined rules. Since the scan can be easily integrated into the microservice template, there is no additional setup and since the scanning process is automated it does not have to be triggered.

**[Why did we take this choice]**

1. DAST provides a realistic perspective on your application's security posture.
2. DAST generates reports with detailed information about identified vulnerabilities, including severity levels.
3. DAST can be integrated into your software development lifecycle (SDLC) and CI/CD pipelines, automating security testing and ensuring that security is considered throughout the development process.
4. Rapid feedback helps developers address security issues promptly.
5. Incorporating DAST promotes a security-focused mindset among developers and stakeholders.

# Back-end

## [How did we do it]

We have integrated ZAP using GitHub Actions and are using the ZAP full scan that works with more than 130+ rules. Example Reports from Space Modernisation Account:

**Summary of Alerts**

| Risk Level | Number of Alerts |
|---|---|
| High | 0 |
| Medium | 1 |
| Low | 1 |
| Informational | 2 |
| False Positives: | 0 |

**Alerts**

| Name | Risk Level | Number of Instances |
|---|---|---|
| Proxy Disclosure | Medium | 5 |
| Strict-Transport-Security Header Not Set | Low | 3 |
| Non-Storable Content | Informational | 1 |
| Storable and Cacheable Content | Informational | 3 |

**Alert Detail**

| Medium | Proxy Disclosure |
|---|---|
| Description | 1 proxy server(s) were detected or fingerprinted. This information helps a potential attacker to determine<br>- A list of targets for an attack against the application.<br>- Potential vulnerabilities on the proxy servers that service the application.<br>- The presence or absence of any proxy-based components that might cause attacks against the application to be detected, prevented, or mitigated. |

## [What were the learnings and challenges]

1. We used the ZAP scan as a prerequisite/ alternative for pen testing in our account.
2. ZAP Scan helped us to fix some security warnings that were generated by the report which we may not have fixed previously.
3. ZAP provides 3 different scanning options, Baseline scan(30 rules), API scan (60 rules) and Full scan(130+ rules)

# Back-end-Security-003 - Lightweight and secure base images for application containers

**[What is it]**

Containerized applications should use lightweight bases images to get faster CI time, fewer vulnerabilities in the deployable images and low storage cost. Additionally, the docker containers should be run as non root users for higher security. Distroless container images solve this problem.

**[Where have we used it]**

At **John Deere** we have an ecommerce application which supports sales of parts for John Deere farm equipment and John Deere merchandise. We have multiple containerized microservices.  Almost all of them use Distroless container images.

**[Why did we take this choice]**

We wanted our deployable images to be as lean as possible to get faster CI time for pulling base images and publishing the deployable images to our artifact repository (AWS ECR). Lightweight images also help in reducing storage cost on AWS ECR. Distroless container images provided us lightweight base images resulting in lightweight deployable artifacts.

In addition, we found fewer vulnerabilities getting reported in CI by our container vulnerability scanning tool (Trivy) as the Distroless base images are very thin with minimal packages.

We also found that Distroless container images provide out-of-box support to run our application containers as non-root users by specifying the base image sha or tag for non-root users.

**[How did we do it]**

In application docker file, we import our base image from Distroless container images

```
FROM
gcr.io/distroless/java17@sha256:3a4ea21bd7b412b8b6ae61313b39337d8f03b
b6844013810e8e4625d8c765edb
```

The sha specified here is for the non-root user. sha based non-root images can be found here https://console.cloud.google.com/gcr/images/distroless/GLOBAL/static

**[What were the learnings and challenges]**

We pinned the Distroless base image version to specific sha for better control and conscious upgrades. If we had used the non-root tag instead, the base image can change without our knowledge if Distroless makes an update available.

**[References]**

- Distroless Container Images: GitHub - GoogleContainerTools/distroless: 🥑 Language focused docker images, minus the operating system., https://console.cloud.google.com/gcr/images/distroless/GLOBAL/static
- Running docker containers as root users is not advisable from security perspective: Running Docker Containers as ROOT: | dockerlabs