E-commerce platform architecture

# Summary

# Assumptions

With any architecture approach, it is often the case that we need to dive into a deeper understanding of the requirements which are not entirely available at this point and designing an e-commerce solution for Amazon is different from Takealot. To this I have made some assumptions about the platform for this document:

1. This is a regional specific e-commerce solution and not a global platform. For example Takealot only services South Africa.
2. I am assuming that we have eliminated implementing open source solutions (like osCommerce) or SaaS solutions (like Shopify) and have identified the need for a bespoke system.
3. This is a public e-commerce solution and not one that is limited, for example a wholesaler who only sells to a limited number of pre-registered sellers.
4. I am focusing on the MVP of an e-commerce platform, which includes
   1. Home page
   2. Search
   3. Basket
   4. Product Listing
   5. Product Detail Page (PDP)
   6. Checkout (including payment)
   7. Account management; i.e. signup, signin, account details

e-Commerce has other key aspects, like integration into a ERP for fulfilment, integration into a product management system, and the system would also have non-key aspects, like wish lists or product assistant tools which are out of scope for this to keep this architecture to a manageable size.

1. This is a greenfield project, and nothing exists at this stage.
2. The technology chosen has been done with practicality in mind; for example, while Svelte may offer benefits over React, we may struggle to hire skilled Svelte developers compared to React.
3. I have assumed this is a large scale solution, something between a Woolworths online store, up to Takealot level.

# High-level structure

From my experience the scale of an e-commerce platform requires more than a single cross-functional team can deliver, and so we will need multiple teams. Applying Conway’s law to the multiple teams and the belief that teams should own, end-to-end, what they deliver does lead us to needing ways for teams to build discreet pieces of technology which can be brought into a single experience; both for the frontend and backend.

## Frontend

The approach on frontend side is to embrace the micro frontend [[1]](#footnote-1) approach which will allow each team to build out their portion of the UI independently. While micro frontends are envisioned to allow teams to use any frontend tech and have them interoperate correctly, from an organisation structure I would standardise on a single framework (likely React) to ensure that people can work across teams, lowering support costs, and building up repositories of shared knowledge. This also would allow for a component library and common style sheets which work with the framework to be built and maintained providing a unified user experience.

Having a micro frontend solution, does require an additional core team frontend core to build out the routing and the logic to provide the ability to register the custom elements and pages, and provide the common messaging framework for intercomponent communication.

This means that each of the 6 feature teams defined in the assumptions, will be able to build their piece of the UI. I am going to take basket as an example of what they would be responsible for:

1. An add to basket custom element which would be used by the PDP team (or in future teams who own specific PDPs) to add the button to their page.
2. A basket icon custom element which sits on the top of the webpage and is updated when items are added to the cart using the messaging framework (initiated by the above element)
3. A page of the basket which handles quantity of items, removal of items, and offers a path to checkout which would be registered as a specific route in the front end system.

## Backend

For similar reasons to the frontend, we are adopting a microservice architecture where each team builds their own backend API to meet the needs of their UI components and other services. Again, like with the frontend, it will be recommended to standardize on a common language and runtime so that people can move between teams and shared knowledge can be built up.

In terms of what the backends do, that is determined by the team themselves and there should be no prescribed rule. It may make sense for the product catalogue to offer a GraphQL based system so that PDPs, listing pages, checkout and basket can query in a way which makes sense for them. Checkout may offer a REST based service interface so that it has a more transaction experience which aligns with how it works. It would be up to each system to decide what would work best.

## Platform

We will also need a platform team whose job it is to run the infrastructure and provide tooling for the development teams. This starts with simple things like source control (recommendation here would be GitHub or similar hosted option), ways of sharing APIs (like Backstage by Spotify[[2]](#footnote-2)) and running the cloud platform which this runs on top of. This tooling should have default pipeline sets too, so that a new team should be able to commit and it produces the correct artefacts for deployment.

The platform team should also have ways for teams to launch and configure their services and frontend without needing to do everything manually. This likely is a configuration system which links the source control to a team and handles the creation of namespaces, and provisioning of servers. This should also handle the security settings if the service is available only to other services (like a product catalogue may want, since it is the other services which will service the data to the UI) or if they want it available on the internet. This should also allow the configuration of routing and ways of adding new pages in to the micro frontend architecture.

What I have not covered is the detail of how everything from frontend to backend is hosted because that is also the responsibility of the platform. Ultimately the teams should be producing a docker image which is run on top of a managed K8s infrastructure. This allows the APIs to communicate internally and make use of K8s namespaces for partitioning. It also allows them to be deployed individually and with their specific needs. This also enables us to make use of K8s cronjobs for execution of actions at a specific time.

# ERD

The ERD for this MVP is really simple, basically just 3 things:

1. A customer which contains all the details of the customer like name, address, credit card etc…
2. A basket which is one to one mapped to a customer and stores product items and how many of them.
3. One of the most important, the product item itself which has all the details of a product.

A diagram of a product item

Description automatically generated

Obviously this will grow as the complexity moves beyond the initial MVP. Below is an example of where we add

1. Adding wish list functionality
2. Breaking out shipping address from customer so we can enable multiple shipping addresses for a customer
3. Breaking out banking details from a customer so we can enable multiple payment options for a customer
4. Adding orders which can be used to track the order for the customer in the system, and not purely in an external ERP system
5. Adding product lines to the product catalogue to define groups of products. In this case it is one to many, enabling scenarios like *Cellphones* listing all the cellphones but it could become many to many as a Samsung phone might be in *Cellphones, Samsung, Android* etc… All this should be enabled over time.

A diagram of a customer

Description automatically generated

At the core, the idea of a universal ERD though fails when considering the structure of the teams – rather the ERD would be defined by the teams who own their vertical stack and integrate with others, and as each team evolves their service. The role of architecture in this is to enable communication and lower costs by finding duplication, not instruction in the design of the system.

# Data Management

Data management is fundamentally up to each team to solve and teams should be empowered to use relational or NoSQL DBs as it fits their need. Teams may even choose to use both. Examples of how this may playout:

1. Basket team may choose to have the data run in a NoSQL DB to gain the performance advantages gained by trading off atomic aspects of a DB.
2. The checkout team, may choose to run a relational DB because they see the benefits of ACID in ensuring that there are no issues with payments not going off and items being shipped.

The only aspect here that would be recommended is to use hosted database solutions from the cloud provider, rather than running their own inside k8s. This is due to the significant performance and cost benefits that are enabled by this.

# Security and Compliance

This is a huge topic for any system, and could be its’ own document entirely, so for a high level architecture it would need to be tackled in 2 ways: systems and education.

## Systems

The systems part is broad, as each part has its own need for running but the key points that I would look into are:

1. Ensuring end-to-end encryption and encryption at rest for all data
2. Ensuring that there is a reliable and secure API gateway that sits in front of the actual APIs used from the web and/or apps
3. OAuth implementation for authentication of customers
4. Internal systems authenticate with each other and the underlying platform using a keys which are automatically rotated and managed.
5. No access to production machines from development.
6. Use of IAM tooling in the cloud provider to ensure staff are scoped to just their namespace.
7. Automated scanning for keys and passwords
8. Automated code scanning for security issues
9. Automated dependency scanning of the code and container images
10. Providing a common securely locked down base image for the other images to run on

## Education

Regardless of technology systems in place, security is at its core something that people need to understand the implications. This would need to happen with security reviews of designs in terms to help spot concerns in the systems, and training of staff on a regular basis.

# Performance Optimizations

From a performance aspect, again the priority to solve this would live in the teams as each part would need to define its own needs, for example, product catalogue may choose to run a cache layer on top of their data, which allows them to increase performance of the queries to them. A PDP may not cache product info, because they always need to show the most recent prices.

On the frontend side, there would be options for server side pre-rendering and minification of assets to improve performance.

Finally since this is a local eCommerce solution, having it run out of data centres in that location to lower latency would be critical.

# High Availability & Disaster Recovery

The core focus around disaster recovery with this proposed solution is that everything should be handled using infrastructure as code so that nothing is hand made. This is yet another reason for the platform team providing a strong way for teams to onboard themselves via configs that are not specifically the terraform (or similar) scripts. Since this is a single region solution, this model should allow it to fail over to another close region with relative ease. To do ensure this, every 3 months the solution should be made to switch over so that issues can be found and fixed.

Making use of k8s also makes this easier to move around, but also helps with high-availability as it means we can make use of multiple pods to handle requests and use tooling like k8s autoscaling to increase the availability.

# Monitoring and Logging

Observability must be a core aspect that the platform team offers and teams offer. The platform team could provide a k8s sidecar to make collection of metrics easy without having to push them to somewhere. This would enable the APIs to publish them locally (in something like Prometheus format) and it can be pulled into a central storage and provided by a monitoring dashboard (say with Grafana). Again, since this is a templatized system, common metrics (such as the Golden Signals) could be handled automatically and teams then only need to add metrics for their usage.

This same model would work for logging, where the side car could pull logs from STDOUT meaning the pods do not need storage or writing to storage. This would be automatic for the teams, and pushed into a cloud service for the ability to search or using something like the ELK stack to expose.

1. https://micro-frontends.org/ [↑](#footnote-ref-1)
2. https://backstage.spotify.com/ [↑](#footnote-ref-2)