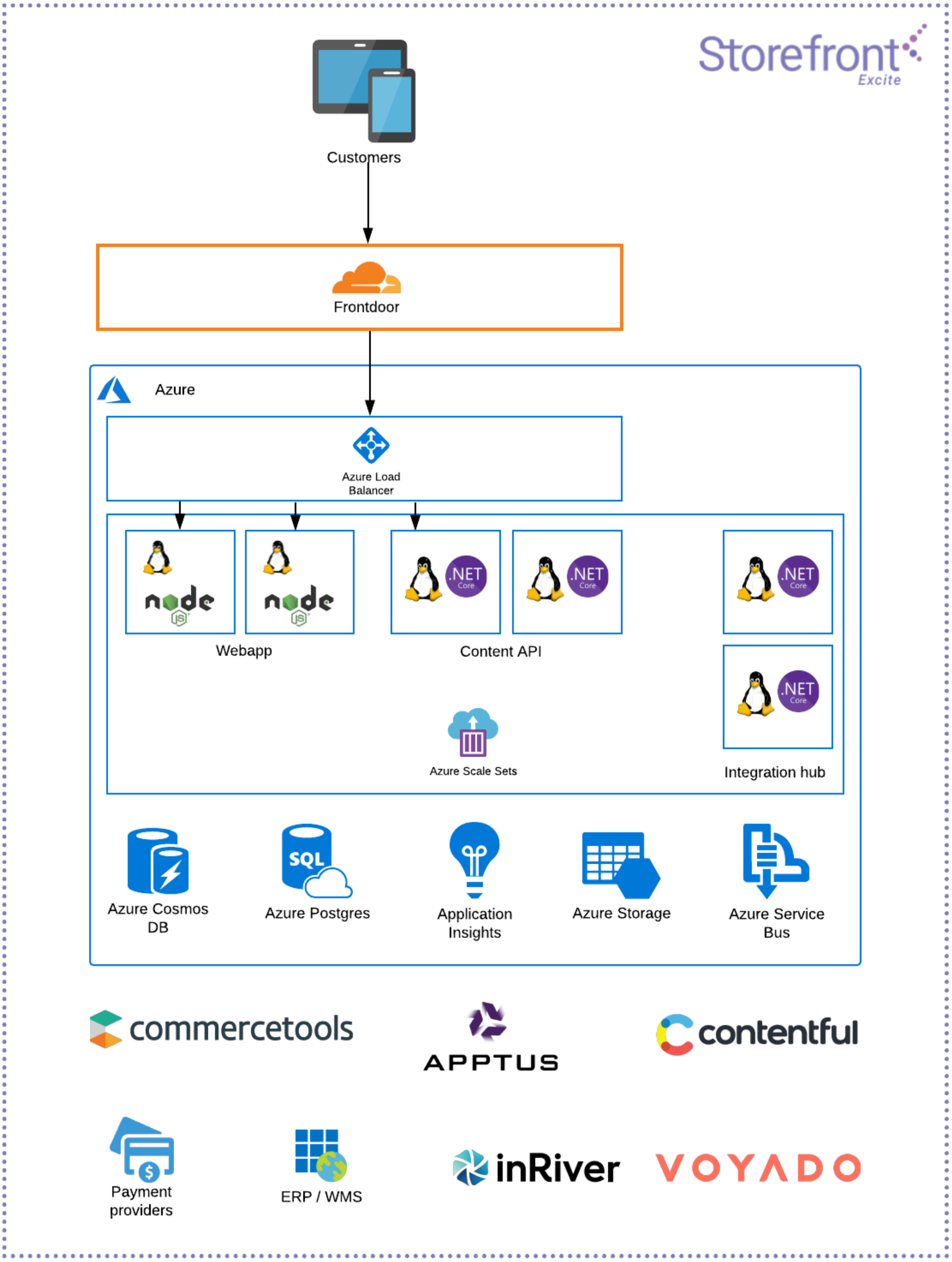
STOREFRONT EXCITE

Architectural documentation

This document is a high-level architectural description of Storefront Excite. The blueprint presented here is made up from components and systems that are modular, parts can be switched out as needed to support a best of breed approach to your specific system landscape.


Storefront Excite
Customers
Frontdoor
Azure
Azure Load Balancer
Webapp
Content API
Azure Scale Sets
Integration hub
Azure Cosmos DB
Azure Postgres
Application Insights
Azure Storage
Azure Service Bus
commercetools
APPTUS
contentful
Payment providers
ERP / WMS
inRiver
VOYADO

1 LANGUAGES, FRAMEWORKS AND CLOUD SERVICES

Each section below goes into more detail about the different services and how they are used. A general list of programming languages, frameworks and services used are:

- TypeScript / Node.js / React.js
- C# / ASP.NET Core / Serilog / Polly
- Azure Cosmos DB / Azure Postgres / Azure Storage / Azure Application Insights

2 EXCITE SERVICES

The storefront Excite solution is composed of multiple services. In its simplest form it is five different services that each can be deployed and scaled individually.

Services hosted in Azure are deployed to Azure VM Scale Sets by default and runs on Linux. By using Scale Sets, Azure will automatically scale out the number of instances needed for each service. Each service is scaled individually.

2.1 **INCLUDED SERVICES**

2.1.1 Webapp

The Webapp service is a Node.js service and its primary purpose is to do server side rendering of the frontend TypeScript code. The webapp responds to all requests that expects HTML as the response type, which is typically the first page load of a session, or if the user refreshes the page.

The only external service that the Webapp talks directly to is the Content API.

2.1.2 Content API

The Content API is a ASP.NET Core service and its primary purpose is to serve the webapp and other touchpoints with content data in JSON format.

All requests where the path starts with "/api/" is automatically routed to the Content API.

The Content API responds both to API calls such as adding a product to the cart as well as routing for content such as CMS pages, category pages, product pages etc.

Routing to content is done by querying a key-value-store where the full url is the key and the value is a small object containing route data. The route data object contains identifiers that can be used to efficiently query other systems. In the case of pages in Contentful it's the entity id and in the case of a commercetools it's typically the product or category key. More data can be added to the route data object if needed.

The building of routing is processed in an event driven manner in the backend/integration hub service described below.

Each type of route data is mapped to a content controller which is an ASP.NET Controller. The typical scenario is that the controller gets the route data object passed in from the framework. It then queries an external system such as Contentful or Apptus eSales, transforms that data into a JSON structure and sends it out to the client.

The Content API serves the Webapp with content but is built to be able to serve other touch points as well such as a mobile app or an in-store display. Customer authentication is done using JWT which can either be sent as a cookie or in an HTTP header.

2.1.3 Integration Hub

The integration hub (also called "backend") is a service that handles integrations with external systems such as commercetools, Contentful, ERP, WMS, CRM, Apptus eSales, etc.

This service also performs as much data processing as possible to minimize the amount of work that the Webapp and Content API needs to perform. Doing this in the backend service offloads work from the other services which means that they are easier to scale and has better response times.

An example of such computation is building full, hierarchical urls for each entity in the system. Neither commercetools nor Contentful has full urls as a concept. To build full urls, all parent entities need to be loaded and their "slugs" needs to be concatenated into a full url. Since this requires multiple API calls to these systems full urls are built in the backend and cached in a way that the content API can use them with very little computation.

The integration hub listens to events in the underlying systems, such as webhooks from Contentful and Azure Service Bus messages from commercetools.

Such events are often placed in internal queues to be picked up by processing jobs. Processing jobs can either be scheduled using a cron schedule or scheduled to run when new entries are placed in a queue.

Internal queues are needed instead of always processing the event directly when the external system calls Excite is because some computation needs synchronization. An entity in Contentful might be changed at the same time as an entity in commercetools and if the computation is performed in the event listeners the same work will be performed twice if those entries are dependent on each other.

Depending on the needs of the Content API the backend will perform content processing which is a core concept in Excite. Both Contentful and commercetools are represented as content providers and the system has one or more content processing jobs which takes data from the providers, processes it and sends it to another system such as Apptus eSales. The content processing infrastructure automatically handles dependencies between entities in different providers. If a category in commercetools has a dependency on an entity in Contentful the content processing for the commercetools category will start whenever the entity in Contentful changes. The infrastructure will also start incremental processing by loading all changed entities and their dependencies in as few API calls as possible to make content processing fast.

2.1.4 Frontdoor

The Frontdoor is a service that runs on a CDNs edge servers. The default is to use Cloudflare Workers but any CDN that allows running code on edge servers can be used. It's possible to use Excite without a CDN that allows this, but the effect is that less data can be cached in the CDN and more traffic will reach the Webapp and Content API.

The Frontdoor service is not hosted in Azure but is deployed directly to the CDN. It will do caching of HTML and JSON responses based on the origins cache-control headers. But instead of just caching based on the url other factors are included in the cache key such as language, country and type of device. What's included in the cache key can be even more granular than that as the Frontdoor service has access to cookies etc on the request.

In the case of Cloudflare the Frontdoor service implements Stale While Revalidate (https://web.dev/stale-while-revalidate/) and Stale If Error which means that the cache is kept up to date by sending requests to the origin servers in the background. This improves latency for the requests that comes in after a cached entry has expired and allows you to have fast expirations (30 seconds or less) on content and still have a high cache hit ratio.

2.1.5 commercetools Merchant Center app

The Merchant Center in commercetools can be extended with custom apps. Excite contains an extendable app for displaying status of running background jobs in the integration hub, the status of internal queues but also extensions for order and customer management with features currently missing in the Merchant Center.

The Merchant Center from commercetools will never contain everything you need. commercetools instead focuses on expanding their API and have building blocks to extend the backoffice functionality. Excite helps you in doing so by making it easier to build such apps.

A Merchant Center app is a React.js based webapp that can be hosted on any static file server and by default Azure Static website in an Azure Storage account is used. The app can talk directly to the commercetools API or any of the Excite APIs.

2.2 ADDING MORE SERVICES

It's expected that more services might be needed depending on the requirements in a project. Having a single service that deals with all integrations might not be feasible if the number of integrations is too high. Another example is having another API service used to serve internal systems that have different needs than public facing traffic.

Adding a new service is very simple in Excite as the process of building and deploying it is already automated.

If the number of services grow it might be time to look at something like Kubernetes for orchestration but Excite does not require Kubernetes by default.

3 SINGLE PAGE APPLICATION

The website is a Single Page Application built using React.js with careful consideration taken to the amount of JavaScript code that each client needs to download. Using the Single Page Application architecture for an e-commerce website is a double-edged sword since it's easy to end up in a situation where too much JavaScript code needs to be downloaded which slows down the application startup in the browser, giving a worse experience for the customer. At the same time the Single Page Application architecture makes it much easier to build highly interactive and app like experiences which makes browsing around much more pleasant.

Excite is built to serve both needs. The build process will perform code splitting and lazy loading to ensure that as little code as possible is downloaded and executed during startup.

The styling uses "CSS in JavaScript" which lets frontend developers co-locate the logic for styling together with their React components, instead of having the styling in a separate file. To mitigate the extra runtime cost of applying styling in JavaScript Excite contains an optimizing compiler that executes the styling code during build time and extracts rules that can be statically evaluated to a separate css file.

3.1 PERCIEVED PERFORMANCE

The use of Single Page Application architecture means that Excite can decouple the user experience from how fast a customers network is. Even if the Content API has fast response times the customer might be on a slow network. To ensure a good user experience regardless of the state of the network the frontend uses granular caching of data.

An example is the navigation between a category page and a product page. The customer might not know exactly what she or he is interested in and will many times go back and forth between products and browse around. To make that experience fast Excite will reuse the product data from the category page to instantly display as much as possible on the product page while loading more data in the background. The customer can quickly go back to the category page without having to wait and will instantly see the category page again as that data is also cached in the browser. This data is cached per customer in memory to avoid cache invalidation problems and is also revalidated as needed.

3.2 **TYPESCRIPT**

A challenge with consuming a JSON API in JavaScript is that it is hard to make changes and hard for developers to know what the data structures that the API responds with looks like. It is also hard to make changes in those data structures because you don't know which parts of the frontend code uses them.

Excite mitigates that by automatically generating TypeScript definitions for all data structures that the APIs involved will use. This helps frontend developers explore the data they get from the API using their editor, but most importantly it will fail the build if a data structure is changed and the frontend code has not been updated. This does not remove the need for testing, but it helps catch a lot of simple bugs and reduce the time required for testing.

Excite also makes use of the TypeScript compiler API to do various compile time optimizations to improve performance. An example is that the build will automatically find all React components that are responsible for rendering different pages such as the start page, the category page, etc. and ensure that the code is split into multiple, optimized bundles.

4 SCALABILITY

Excite achieves excellent scalability by having a layered architecture and a focus on performance in all parts of the architecture. Performance is important because high performance often means doing less computation which is easier to scale.

A big challenge with scalability is to handle large spikes in traffic that comes unexpectedly. For planned events scaling out can be done beforehand to meet the expected new traffic but all spikes are not planned.

The first layer is at the CDN, where as much caching as possible is done of HTML and JSON content as well as static files such as images. By using the CDN cache efficiently a big part of a traffic spike is handled there which gives the origin servers more time to scale up.

The next layer is ensuring that new instances can be started very quickly and that the startup time for a new instance is minimal. By making Webapp and Content API instances as stateless as possible it is very quick to start new instances, and by relying on fast underlying services rather than memory-based caches the risk of bringing in cold instances that needs lengthy warmup is minimized.

A traditional website does full page loads when the customer navigates around the site. Which means that a lot of data that does not change between pageviews are sent with every pageview. A pageview such as navigating to a product page in Excite is not a full page load but instead an API call which only fetches the data needed for that specific product. It does not need to fetch the cart, the main menu, etc for every pageview. This also increases scalability since it minimizes the amount of data needs to be fetched for each pageview.

4.1 MULTIPLE DATA CENTERS

While commercetools doesn't allow a single project to exist in multiple regions it's still possible to host Excite in multiple data centers at the same time. Since the Webapp and Content API services are stateless, instances of those can be brought up in different regions and the Frontdoor service can be used to route to different data centers depending on where the user is geographically.

The Content API is less dependant on the commercetools API which means that latency between the API and commercetools only affect a few parts of the site such as adding something to the cart.

This approach scales well with the services that the Content API is often dependant on such as Contentful and Apptus eSales. Contentful has a global CDN and it is possible to host multiple Apptus eSales instances in different regions.

5 REFERENCES

Azure Scale Sets:
https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview

React.js:
https://reactjs.org/

Node.js:
https://nodejs.org/

TypeScript:
https://www.typescriptlang.org/

Cloudflare Workers:
https://workers.cloudflare.com/

Single Page Application architecture:
https://en.wikipedia.org/wiki/Single-page_application

.NET:
https://dotnet.microsoft.com/

CSS in JS:
https://en.wikipedia.org/wiki/CSS-in-JS

commercetools Merchant Center:
https://docs.commercetools.com/merchant-center/