Technical Platform
Solution Architecture

Prepared for

Markus Wagner
University of Adelaide

taptu

Revision History

Date	Version	Author	Comments
27 Jun 2021	1.0	Luca Gnezda	Initial preparation for 2021 semester 2 University project

Statement of Intellectual Property

Contents

1 Introduction

1.1 Executive Summary

< -- content redacted: What are the key objectives of this software platform -- >

1.2 Purpose

The purpose of this document is to:

1. Define the guiding principles and technical decisions which underpin the solution architecture of the tech platform.

   Ongoing observance and alignment to these guiding principles and technical decisions will allow future design, development, implementation and enhancement to be sympathetic to technical mass and functional momentum of the platform. Doing so will result in fewer bugs, lower total cost of ownership (TCO) and greater speed of feature delivery.

2. Detail of the Solution Architecture, and components therein.

   Ongoing observance will anchor understanding for future technical team members, as well as provide a refresher to team members revisiting previously constructed parts of the system.

3. Highlight known limitations and closing remarks.

   A closing commentary for the MVP will highlight points of note, likely to impact or influence future scopes of work.

1.3 Stakeholder Consultation

< -- content redacted: Who are the people that have been consulted in the development of this architecture and what aare their roles/responsbilities -- >

1.4 Glossary

Term	Definition
API	Application Programming Interface
AuthN	Authentication
AuthZ	Authorisation
Azure	Abbreviation for the Microsoft Azure Cloud Compute Platform.
CMS	Content Management System
IaaS	Infrastructure as a Service
MVP	Minimum Viable Product
PaaS	Platform as a Service
SaaS	Software as a Service
Tech Platform	General term used to define the technical solution and its implementation in its entirety, which in turn supports the goals and objectives of this product.

< -- content redacted: Specific defnitions removed -- >

2 Conceptual Solution

< -- content redacted: Description of the product concept and how this solution architecture supports its delivery. -- >

3 Guiding Principles

In order to arrive at a meaningful solution architecture, technologies must be selected. But to ensure that technology decisions were appropriate fit, we first identify the guiding principles of the Tech Platform.

3.1 Roadmap and Feature Prioritisation

We will:

1. Use a Product Management approach to the development of the Tech Platform.
2. Apply reasoned and described logic for any decisions we make. This will include these guiding principles as well as a questions and decisions log.
3. Utilise a strategic backlog tool for large scale planning and an agile backlog for technical construction.
4. The initial version of the Tech Platform will strive for a minimum viable commercialisation solution that also complies with these guiding principles. This will minimise the level of investment prior before revenue generation can commence.

3.2 Security & Privacy

We will:

1. Preference security and privacy over technical ease. Confidence in the security and privacy architecture of the platform is essential. Once this confidence has been shaken, users will turn away from the solution. Every effort must be made to prevent this from occurring.
2. Only use a well-respected and expert provider, of identity and authentication services. We will not build this as a custom solution.
3. Implement authorisation logic on every endpoint.
4. Separate identifiable information from information to be kept anonymous.
5. Encrypt all network traffic.
6. Implement recommended practices for hosting architecture and software design.
7. Implement DevSecOps principles for continuous, secure development and deployment.
8. Avoid a persistent client device deployed software footprint where possible to avoid long term storage of data on devices.
9. Assess all software against OWASP and any other security assessment frameworks relevant to the technology selected.

10. Strive to meet the relevant level of security compliance, user rights and privacy controls as required in Australia, the European Union and the United States.

3.3 Frameworks & Software

We will:

1. Build all bespoke software using mainstream technologies in order to maximise longevity, supportability and sustainability while minimising costs.
2. Where possible, strive to build bespoke software that is agnostic to hosting. Ideally it should be possible to re-host the solution on premise, or in any mainstream cloud offering.
3. Where possible, avoid reliance on proprietary frameworks with a closed developer ecosystem ethos, thereby minimising vendor lock-in.
4. Seek to leverage frameworks that maximise future flexibility, agility and sustainability.

3.4 Infrastructure & Hosting

We will:

1. Preferentially use cloud hosting over on-premise in order to maximise application availability, scaling and agility.
2. Preferentially use Platform as a Service (PaaS) over Infrastructure as a Service (IaaS), where cloud hosting is possible.
3. Strive to deliver software which removes the need for traditional infrastructure support resources, allowing a limited operating budget to be consolidated on solution and DevSecOps resourcing instead.

3.5 Usability, Supportability & Sustainability

We will:

1. Select technology which is well known by the local talent pool, so future resourcing is easier to find and less costly to engage.
2. Strive to create a tailored and compelling user experience for each of the personas identified.
3. Strive to build future internal technical capacity into the business, supporting self-sufficiency. This will include the education and coaching of young professionals throughout the product lifecycle.

4. Continually upgrade frameworks and underlying product versions as they become available, to maintain an evergreen solution (for the life of the project).

4 Technology Selection

4.1 Overview

Based on the aforementioned guiding principles the following technology solutions were made during the Product Seeding phase (circa June 2021). Please refer to the decisions log for further details:

4.2 Software Development Technology Selection

To underpin software development the following technologies were selected:

- HTML5, SCSS, JavaScript & TypeScript for web layer development, extended through use of Angular 12+ and PrimeNG.
- .NET 5 written in C# for API development, extended through use of Dapper and other Nuget packages as required.
- SQL Server for relational database development.
- Avoidance of non-relational database development technologies such as MongoDB or NoSQL. However, we will use JSON schemas in SQL columns for certain forms of dynamic content where relevant.

The rationale for these decisions were:

1. Technologies align to the experience and capability profile of the team engaged to deliver the work.
2. Technologies are well supported and adopted locally and globally, with rich online communities
3. Technologies are known to scale well
4. Technologies are known to interact well with cloud hosting patterns
5. Technologies are a mix of open source and proprietary. However, all are distributed from sources and vendors with clear commitments to open source and who significantly contribute to that community.
6. Angular provides a rich application centric, single page experience (SPA), which is in line with the user experience expectations of the sponsors.
7. We have elected to not use a framework-based approach to web layer store patterns (eg: NGRX, NGXS). Many of the behaviours expected are unlikely to require this level of complexity. A custom, simpler, lightweight implementation will be produced in its place. We believe this will avoid many of the caching management challenges with these patterns and achieve better agility and sustainability of product development.

8. Java was ruled out due to the poor commercial outcomes resulting from its acquisition by Oracle.
9. NodeJS was considered but discounted. While it would likely deliver comparable outcomes, we believed that scalability, performance and security drivers put .NET ahead of NodeJS for this particular solution.
10. Technologies are well supported across a range of cloud hosting providers, avoiding locking.
11. Non-relational technologies have been avoided as there was insufficient need for loosely typed information and a clear need for well-managed and structured information. Therefore, use of non-relational approaches would likely result in drawbacks outweighing the benefits. Over the life of the project this would impede delivery times, and magnify post project support risks.

4.3 Cloud Hosting Platform Selection

Microsoft Azure was selected as the preferred cloud hosting platform. Amazon Web Services was also identified as the preferred fall-back option, should the future value proposition of Azure change.

The rationale for this decision was:

1. The sponsors indicated a pre-existing affinity for Azure over alternatives. This is based on Microsoft adoption in their market segment, and prior use of Azure and Office 365 in their other businesses.
2. Given the relative maturity of various vendor cloud offerings, two vendors have a clear capability advantage in rich abstract compute services:
    - Microsoft Azure
    - Amazon Web Services (AWS)
3. Both vendors have a rich and mature ecosystem of services with numerous datacentres around the world.
4. Both have security certification to levels required by the South Australian and Australian Governments.
5. Both have substantial support for the Software Development Technologies identified in Section 4.2.
6. From direct team experience supporting production environments in both, Azure was identified as having slightly better support outcomes for PaaS based workloads that utilise the selected software technologies listed in the prior section.

7. Of the two the teams opinion is AWS has the superior IaaS offering, while Azure has the superior PaaS offering. Given AWS's position of implementing PaaS as loosely abstracted IaaS, this would have required investment in traditional infrastructure resources to manage and automate these services effectively. Looking at TCO cost modelling of Azure PaaS vs AWS IaaS, AWS would have required a greater investment in infrastructure staff resourcing for a overall greater cost. Therefore, the decision was made to deploy within Microsoft Azure.

4.4 Software Development Lifecycle Technology Selection

Azure DevOps, Visual Studio, Visual Studio Code and GIT was selected for the SDLC ecosystem.

The rationale for this decision was:

1. Based on the Software Development Technologies identified in Section 4.2. two sets of SDLC technologies emerged as the logical choices:
    - Microsoft ecosystem, Azure DevOps, Git, Visual Studio, VS Code and/or Atom and SQL Server Management Studio
    - Agnostic ecosystem, Atlassian JIRA, Bitbucket, Bamboo, Visual Studio, VS Code and/or Atom and SQL Server Management Studio
2. While both would do an excellent job of supporting the team, the Microsoft ecosystem was seen as the clear front runner, due to its native alignment with the cloud hosting and development technologies selected.

4.5 Specialised Technology Selection

PrimeNG was selected to enhance team capacity in user interface design and construction.

Auth0 was selected to enhance team capacity in customer identity, sign-in and authentication.

Mailjet was selected to enhance team capacity in bulk email message distribution.

Partial use of SurveyJS to enhance dynamic form development.

The rationale for these decisions were:

1. Multiple options were considered for enhancing user interface design and construction. However, PrimeNG is both one of the top 10 component frameworks for Angular, has a rich theming system that will allow us to reskin the application in the future and supports a Fluent like component pattern which the sponsors had a loose affinity for.
2. For customer identity, multiple options were considered. These included:
    - Azure Active Directory B2C
    - Auth0
    - PingOne

- Okta
- RedHat SSO

3. From prior experience we knew that B2C (the native Microsoft solution) offers a weaker user experience than its competitors, less feature rich, evolves in unpredictable ways that often results in code refactoring, and cannot conceal Microsoft base URLs for API endpoints where required.
4. Likewise prior experience has demonstrated that Okta is better suited to enterprise identity aggregation rather than as a standalone identity platform.
5. RedHat SSO only achieves rich flexibility and mature supportability when self-hosted, which would require an IaaS over PaaS approach.
6. PingOne while being a first rate solution, has an opaque cost model with many hidden costs.
7. Auth0, also a first class solution is now owned and backed by Okta, giving future scale and opportunity.
8. For email messaging, several options presented, all with costs and capabilities meeting out needs. Mailjet appeared to have an effective mix of features and pricing for this need.
9. While we have used SurveyJS in the past and are impressed by its capabilities, it does not integrate with custom component libraries and still uses knockout to deliver an angular experience. For the level of form sophistication we require, we intend to use Survey JS online builder to develop JSON templates, then use the new form features of Angular 12+ to achieve a similar outcome in a much lighter way.

5 Hosting Architecture

5.1 Overview

This section describes the hosting architecture for the TaskAlyser Technical Platform.

The majority of the solution is deployed and hosted within an Azure tenancy, with two third party providers, Auth0 and Mailjet providing authentication and end user communication respectively. The solution exclusively uses Azure's Platform as a Service (PaaS) technologies.

Azure services used for the solution include:

- Resource Groups: Encapsulates a set of services which host an environment in a regional datacentre. Four environments exist, which represent the chosen software development lifecycle (SDLC). These are: Continuous Integration (Development or Dev), Quality Assurance (QA), User Acceptance Test (UAT) and Production (Prod).
- Application Service Plans: Scalable serverless compute platforms on which web and API layers are hosted.
- Service Applications: Web containers which hosts the application codebase, either serving up the web layer to clients, execution of the API codebase, or execution of asynchronous web jobs.
- Virtual Networks: Virtual network environments which can contain subnets and have defined communication pathways between them set up.
- Subnets: Subnetwork sections of virtual networks which can have connections made to them by outside resources within Azure. E.g.: private endpoints and regional VNet integrations.
- Private Endpoints: Network interfaces created within virtual networks which can be linked to SQL Azure Databases and other resources within Azure.
- Regional VNet Integrations: Uni-directional communication pathways between app services and subnets.
- DNS: A Domain Name Service which, in this use case, routes authenticated access via a public connection from the public address to the private address assigned to a resource within a VNet.
- VPN Gateway: A Virtual Private Network that can be securely connected to by devices external to the Azure ecosystem and network. Allowing external users authenticated, encrypted access to the hosted environment for miscellaneous development, auditing, and related purposes.
- Storage Accounts: Redundant serverless storage platforms on which to host files, blobs and queues.
- Blob Store Containers: Storage containers that host uploaded documents which can then be retrieved by the application.

- Queue Containers: Storage containers that host asynchronous messages between layers of the application.
- SQL Server: Scalable serverless compute platform on which to host relational databases.
- SQL Server Elastic Pool: Scalable shared compute pool, allowing databases to share resources.
- SQL Server Databases: Databases containing relational application data.
- Key Vaults: Secure repositories for storing secrets such as connection strings, API keys, GUIDs, usernames and passwords.
- Application Insights: Serverless compute solution for monitoring, logging, auditing and exploring usage patterns for the deployed environments.
- Azure AD: TBD
- Azure CDN: A global content delivery network, providing regionally cached, expedient delivery of media assets, static site content, and more.

5.2 Azure Topology

The hosting architecture has been designed in two segments:

- A primary hosting stack for application hosting.
- A supporting hosting stack to facilitate securitisation, deployment, and technical administration.

Resource groupings have been designed in layered segments. This allows public endpoints and key vault resources to be delivered globally, while solution stacks and administrative resources can be regionalised and independently scaled.

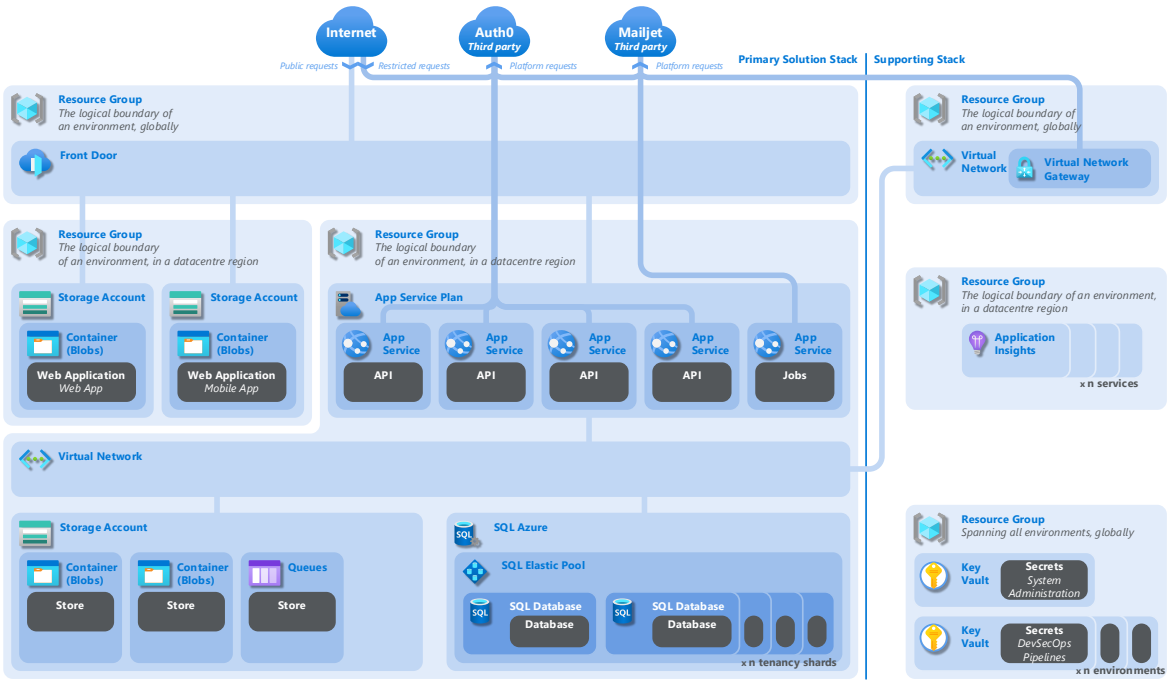

Figure 4: Overall hosting architecture

5.2.1 Primary Hosting Architecture

The primary hosting architecture implements a four-tier model:

- A public facing **access** and routing layer using Front Door.
- A private Web and API **application** hosting layer, using storage accounts and App Service Plans
- A private **network** layer, using a VNet to protect underlying storage resources.
- And a private **data and storage** layer, using SQL Azure and Storage accounts.

Each of these layers is contained within, or spans across, logical resource groups. These define compute boundaries which can be independently managed and scaled.

5.2.1.1 Access Layer

The Access Layer consists of a combination of Azure Front Door. This technology allows for controlled, monitorable access of content and provide reliability, load balancing and regional caching for a quality user experience. This ties directly into the scalability outlined in Section 8.

5.2.1.2 Application Layer

The Application Layer contains, securely, the App Services, within a general App Service Plan, that make up the public facing API/s of the solution. Also contained within this layer are the static file storages that provide the web and mobile applications, logically separated from the storage layer below because they are static and provide only the released applications.

5.2.1.3 Network Layer

The network layer provides the communication channel, and as such control of said communication, between all the services within the hosting architecture, and any external concern. It isolates each section from the others to provide the high degree of controlled security outlined in Section 10. This is all done through careful implementation of Virtual Networks and their Subnets, which are connected into the rest of the hosted components through VNet Peering, Private Endpoints, and Regional VNet Integrations.

5.2.1.4 Storage Layer

The storage layer contains all data aspects of the hosting architecture. It is highly isolated for security, and highly scalable through SQL Elastic Pools.

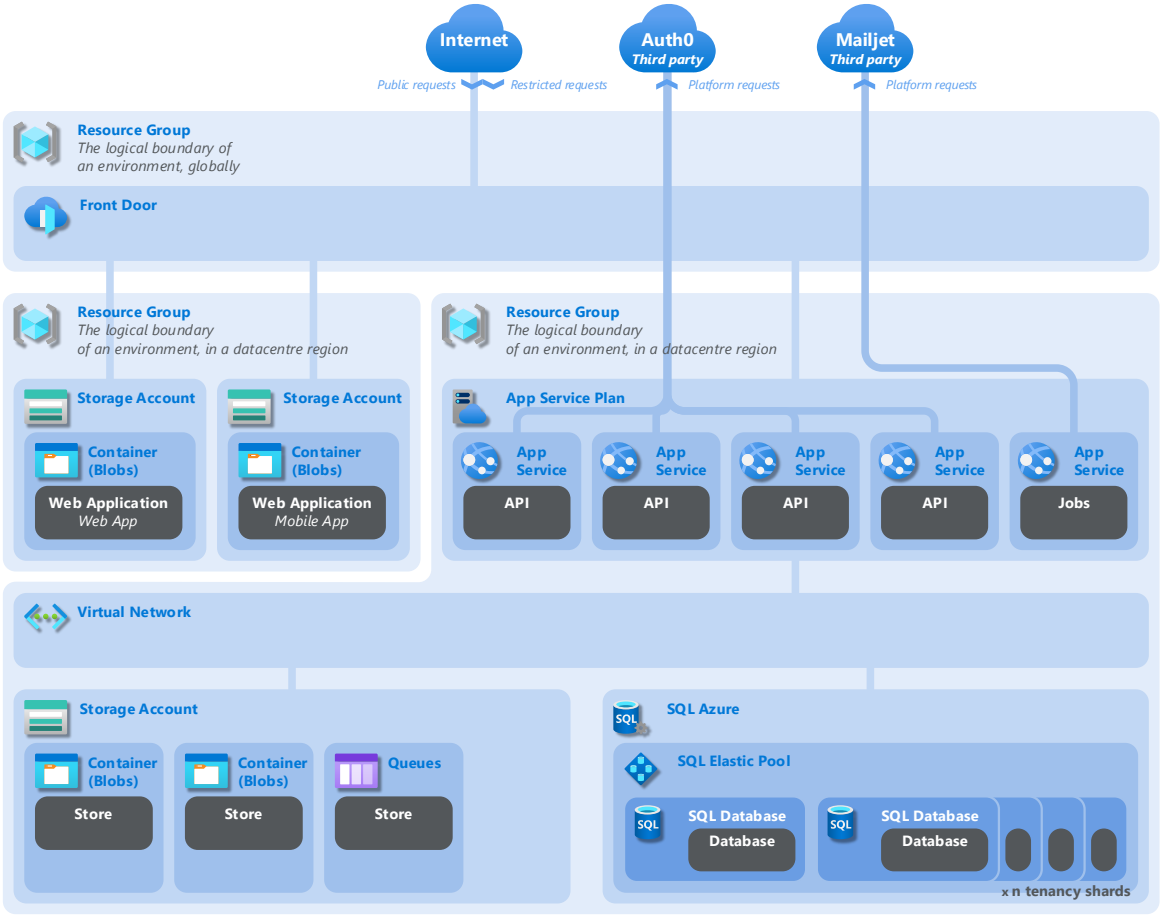

Figure 5: Primary hosting architecture

5.2.2 Supporting Hosting Architecture

A matching set of services then support the host hosting capabilities. These reside in the same resource group as the hosting capabilities they support:

- Vnet and virtual network gateway for secure administrative access to storage resources.
- Application Insight hosting, for monitoring of all web hosts.
- Vault hosting, for storage of system administration secrets.
- Vault hosting, for DevOps automation secrets.

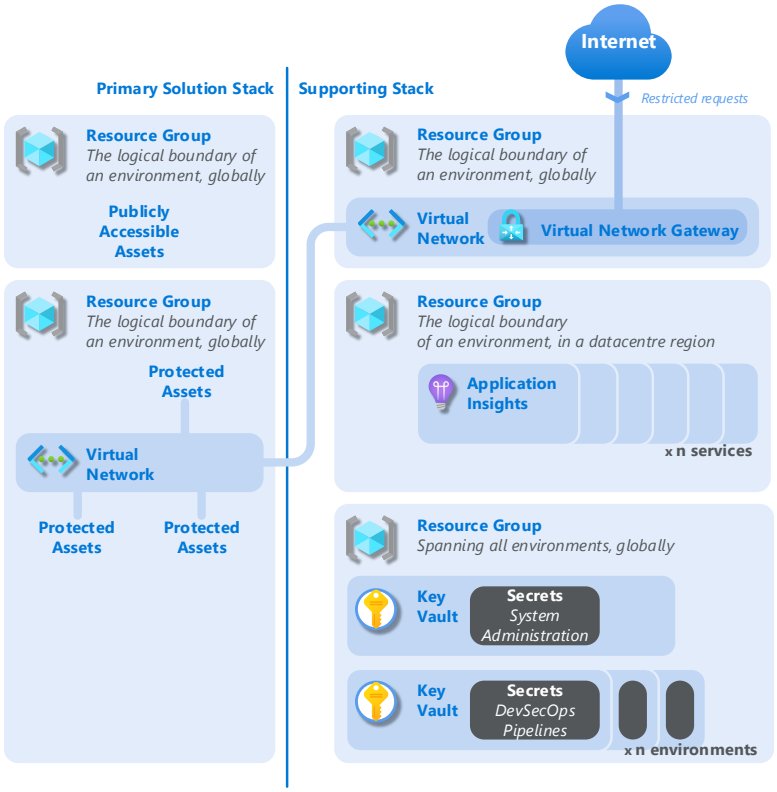

Figure 6: Supporting hosting architecture

5.2.3 Third Party Service Hosting

5.2.3.1 Auth0 – Identity

< -- content redacted: Specific definitions removed -- >

5.2.3.2 Mailjet - Communications

< -- content redacted: Specific definitions removed -- >

6 Software Architecture

6.1 Overview

This section describes the as-implemented software architecture for the Tech Platform solution.

The software itself is broken into five solution layers:

1. A **user interface** layer delivering two discrete experiences implemented independently of each other, tuned to maximise user outcomes of each experience:

    1.1. A large form factor **web layer**, delivering a browser-based user experience. This is built using HTML, TypeScript and SCSS.

    1.2. A small form factor **app layer**, delivering a mobility and field worker data capture experience for android and iOS-based devices.

2. An **API layer**, implementing server-side request-response features and behaviours. This is built using .NET 5 as a C# Web Application. The API layer is then broken down further into four separate API stacks:

    2.1. < -- content redacted: Specific defnitions removed -- >

3. A **data layer**, implementing server-side relational data storage. This is built using SQL Server. Where solution requirements dictate a non-relational approach, we will implement JSON on SQL stored as columns.

4. An **Identity Platform** providing user access control, AuthN and AuthZ, through its API. The chosen platform is Auth0.

5. A **Communications Platform** providing templated email and SMS services for automated user communication. The chosen platform is Mailjet.

6.2 Web Layer Architecture (Desktop Experience)

The Web Layer is implemented using the following key frameworks and technologies:

Angular 12+: A Framework for building rich applications on the web. For more information, please refer to:

https://angular.io/docs

PrimeNG Component Library: A library of predefined Angular user interface components in a range of thematic styles..

https://example.com

The overarching architecture of the Web Layer can be seen in the diagram below.

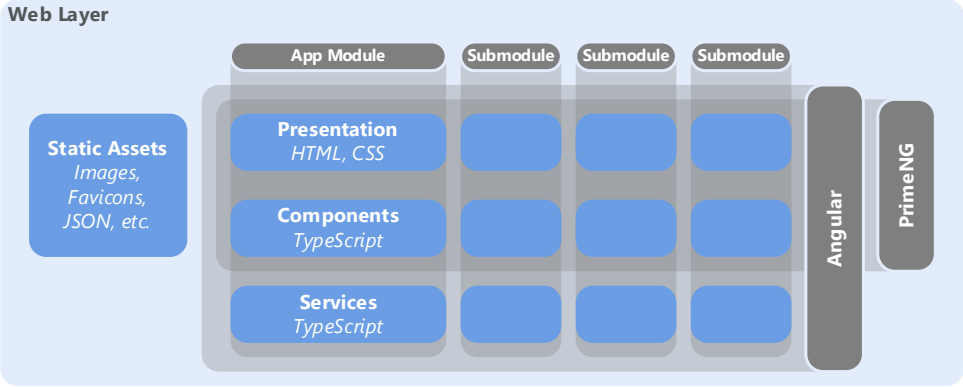

Figure 7: Web Layer Architecture

This layer is managed by SDLC team tooling:

< -- *content redacted: Specific defnitions removed* -- >

and stored in the repository:

< -- *content redacted: Specific defnitions removed* -- >

The implementation of the web layer will use a modular interface pattern. The application will be segmented into:

- Shared static assets
- A core application module, responsible for application entry and submodule loading
- Multiple presentation submodules, representing different aspects of the user interface.

Each of these layers is described in more detail below.

6.2.1 Presentation Layer

The purpose of this layer is to implement the web user interface. This is built on the Angular framework using the PrimeNG component library, and is coded in HTML5, SCSS and TypeScript.

The solution file path for this layer is:

< -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific defnitions removed* -- >

6.2.2 Components / View Layer

The purpose of this layer is to implement UI control behaviour and logic. It is built on the Angular framework and is coded in TypeScript.

Components are the basic reusable building blocks which form an Angular application and are ultimately what get rendered to the DOM and viewed by the user in their browser. Some best practices are:

- Components should follow the Single Responsibility Principle.
- Components should largely be presentational in nature. Taking advantage of content projection where necessary to minimise the number of components exposed to state.
- Components should not be exported from their module unless the component is designed to be shared in multiple contexts, in which case it should be in the Shared Module.
- Components should be strictly typed.
- Components should prefer logic in the component file rather than the template file.
- Components should prefer separation of logic, presentation, and templating.

The solution file path for this layer is:

< -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific defnitions removed* -- >

6.2.3 Services / State Layer

The purpose of this layer is to store state information for the application and manage connections to external APIs, caching and brokering data exchange between the APIs and the user. It is built on the Angular framework and coded in Typescript.

Services provide loosely coupled functionality through a robust dependency injection system. Services exist to abstract business logic or specific implementation of data fetching from components. This centralises changes and aims to make presentation components readily usable across multiple projects or scopes.

Some best practices for services are:

- Services are injectable by default.
- Singleton services follow the current Angular 6+ method.
- Services should be used for configuration of runtime specific variables that are not able to be included in the application at build time. Such as environment variables.
- Services should cache data where appropriate, such as environment configuration, that exists unchanged over the lifetime of a user session.
- Services should provide async functionality for logic such as data fetching or storage.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

> < -- *content redacted: Specific defnitions removed* -- >

6.2.4 Shared Static Assets

Shared static assets are files that live outside the build process in Angular. They are copied to the root of the application at build time and are served statically by the Azure Web App at runtime. Assets typically found in the static folder include:

- Custom Fonts
- Custom Iconography
- Runtime Environment Variables
- Images

6.2.5 App Module

The App Module is the entry point for an Angular Application. It contains the same basic properties of any other submodule in the system. To understand more about its composition please see section 6.2.6 below.

The App module is intended to be lightweight to promote fast application loading time, defines lazy loaded sub modules based on application routing logic, and sets up application wide singleton services known as providers. In addition to providing app level configuration, they will

also contain components that exist in the "core" of the application and persist over its lifetime. Common examples of this are persistent navigation or site-wide footers.

6.2.6 Submodules

Submodules will be defined for key aspects of the user experience. This will include things like:

< -- content redacted: Specific defnitions removed -- >

Given many of these feature sets are optional based on workspace settings and permissions, this will allow the application to limit payload size and client memory footprint to only the elements needed.

The Angular ecosystem is extensive and developers onboarding themselves with this product should also be familiar with their documentation. However, for brevity and consolidation of knowledge this document provides some information about the most widely used elements of a submodule and best practices.

Some best practices for modules are:

- Submodules should be lazy loaded to speed initial app launch and defer downloading of assets only until when needed.
- Modules should be self-contained, except for a Shared Module which exports elements that are explicitly used in multiple contexts.
- Elements of a given module should be scaffolded by the Angular command line utilities (where they exist).

6.3 Mobility Architecture (Mobile Device Experience)

< -- content redacted: Specific defnitions removed -- >

6.4 API Layer Architecture

The API Layer is implemented using the following key frameworks:

ASP.NET Core C# Web Application: At time of writing the current stable release of .NET Core is 5.

https://dotnet.github.io/

Dapper: Framework for automating mappings between objects properties and relational datasets. For more information, please refer to:

https://github.com/StackExchange/Dapper

For security, scalability and data sovereignty reasons, the total API surface area has been divided into four separate compute units (or microservices). These individual API stacks are:

< -- *content redacted: Specific defnitions removed* -- >

To address requirements for future globalisation and distributed scale, APIs each implement one of the following two service host co-ordination patterns:

1. Command Query Pattern
2. Regional Compute with Sharded Data Pattern

These patterns are described in more detail in section; 8 Scalability Architecture.

Within each API layer, the Technical Platform uses a standard four tier architecture, breaking the problem domain into control, business logic, data logic and data access. There is also an optional fifth layer between control and business logic to handle interactions across business logic units (see diagram below). This allows segregation of responsibilities for coded behaviour and helps isolate relevant code when developing.

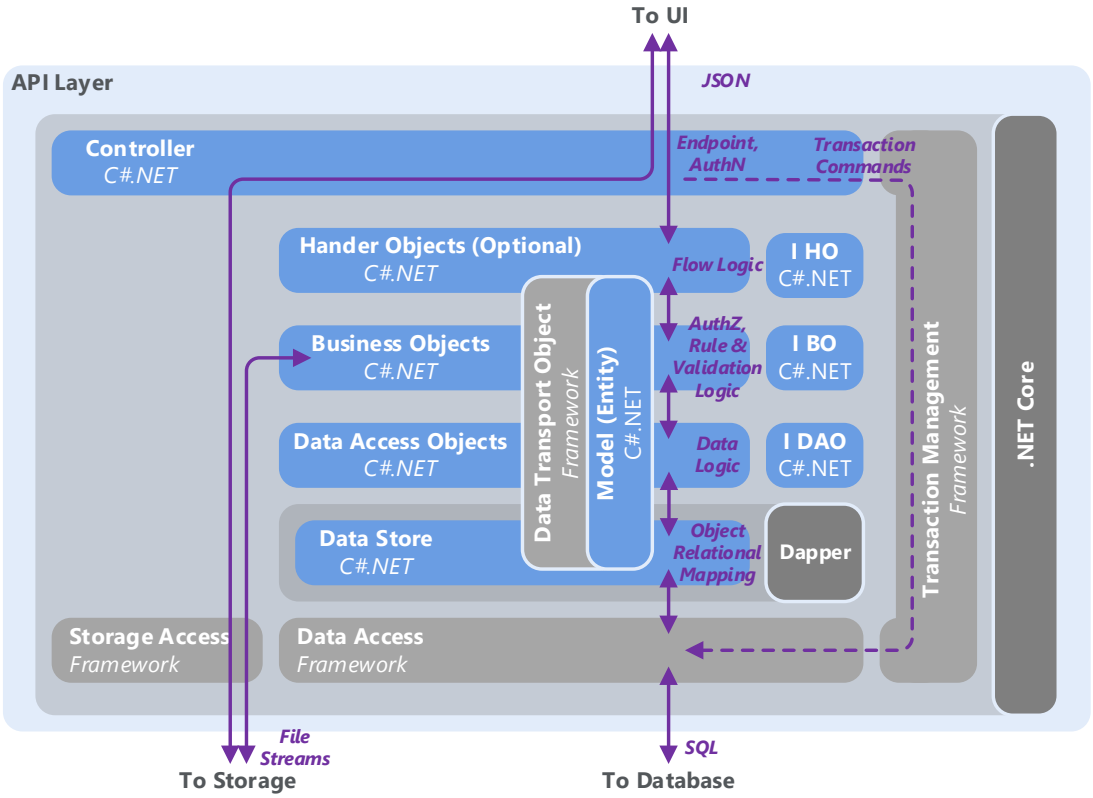

Figure 8: API Layer Architecture

The handler layer, business layer and data layer are implemented using interfaces. This allows the API codebase to also implement test harnesses at these layers for unit test automation during the automated build process.

Data is passed around the various layers of the API in one of two forms. For strict compliance to structured data, entity classes (or models) are used. These are typically generated from the data layer using a T4 template, end extended with partial classes as required. For more dynamic needs, data transport objects (DTOs) are used instead.

The API layer is managed by SDLC team tooling:

< -- *content redacted: Specific definitions removed* -- >

and stored in the repositories:

< -- *content redacted: Specific definitions removed* -- >

Each of these architectural layers is described in more detail below.

6.4.1 Controllers

The purpose of this layer is to define API endpoints, check authentication and execute top level logical flow and routing (if required). It is built on the .NET Core Framework and coded in C#.

Traditionally controllers provide no business logic to the application. Only simple checks are made against incoming variables to ensure they are valid (such as making sure they exist) and calling the correct Business Object to execute the domain specific code. Middleware should be created which intercepts all thrown errors and returns responses as appropriate. This further exemplifies the Separation of Concerns mentioned above, makes code clearer, and typically means most code pathways only need to deal with "Green Light" scenarios. It then becomes the responsibility of the Business and Data Layers to explicitly check for and throw errors as required.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

> < -- *content redacted: Specific defnitions removed* -- >

6.4.2 Models And Data Transport Objects (DTOs)

The purpose of this layer is to define structured reusable entities. In the context of models these entities represent underlying data structures, the tables in the relational database. In the context of DTOs the data structures that correspond to specific views or API calls. These can be created manually or generated programmatically through tools such the T4 templating solution. These are built on the .NET Core Framework and coded in C#.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

6.4.3 Handlers (optional)

The purpose of this layer is to intermediate flow between multiple disparate business objects, i.e.: situations where cross stack actions are required. These objects represent the domain specific implementation of the interfaces defined in 0. These are built on the .NET Core framework and coded in C#.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

> < -- *content redacted: Specific defnitions removed* -- >

6.4.4 Handler Interfaces (where Handlers are used)

The purpose of this layer is to define interfaces for handler flow logic which allow test harness injection during the automated build process. Interfaces are essential in successfully decoupling code and are the core of the Inversion of Control programming paradigm. These are built on the .NET Core framework and coded in C#.

The solution file path for this layer is:

< -- *content redacted: Specific definitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific definitions removed* -- >

Business Objects
The purpose of this layer is to define business logic for application features, including authorisation behaviour. These objects represent the domain specific implementation of the interfaces defined in 0. These are built on the .NET Core Framework and coded in C#.

The solution file path for this layer is:

< -- *content redacted: Specific definitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific definitions removed* -- >

6.4.5 Business Object Interfaces

The purpose of this layer is to define interfaces for business logic which allow test harness injection during the automated build process. Interfaces are essential in successfully decoupling code and are the core of the Inversion of Control programming paradigm. These are built on the .NET Core Framework and coded in C#.

The solution file path in the API for this layer is:

< -- *content redacted: Specific definitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific definitions removed* -- >

6.4.6 Data Access Objects

The purpose of this layer is to define data logic including data transformation and the preparation of SQL statements. These Objects represent the domain specific implementation of the interfaces defined in 6.4.7. These are built on the .NET Core Framework and coded in C#.

The solution file path for this layer is:

< -- *content redacted: Specific definitions removed* -- >

Example of code at this layer:

< -- *content redacted: Specific definitions removed* -- >

6.4.7 Data Access Object Interfaces

The purpose of this layer is to define interfaces for data logic which allow test harness injection during the automated build process. Interfaces are essential in successfully decoupling code and are the core of the Inversion of Control programming paradigm. These are built on the .NET Core Framework and coded in C#.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

> < -- *content redacted: Specific defnitions removed* -- >

6.4.8 Data Store

The purpose of this layer is to define access endpoints to SQL data sources and managing Dapper-based object-relational mappings between entity attributes and database columns. This is built on the .NET Core framework and coded in C#.

The solution file path for this layer is:

> < -- *content redacted: Specific defnitions removed* -- >

Example of code at this layer:

> < -- *content redacted: Specific defnitions removed* -- >

6.4.9 Data Access

The purpose of this layer is to access SQL data sources. This is implemented in full by .NET using System.Data.

6.4.10 Transaction Management

The purpose of this layer is to control transactional boundaries for multi-request database changes. This is implemented in full by .NET using System.Transactions.

6.5 Data Layer Architecture

The Data Layer is implemented using SQL Azure for both relational data and any related JSON document storage. It also implements blob storage for access by business objects. The Data Layer Architecture will be required to work within the Database permissions as defined in 10 Security Architecture.

Data access within the Data Layer will be actioned using one of the following two methods:

- Direct Tabular Access, directly accessing the data as it exists in the tables in the SQL database.

- Stored Procedures, accessing the data in the SQL database through an intermediary channel which abstracts away complexities where needed.

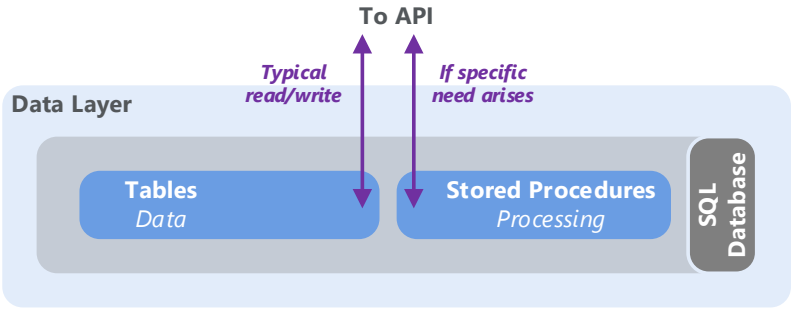

Figure 9: Data Layer Architecture

6.5.1 Direct Tabular Access

Most of the data access to the data layer will be performed from the Data Access Objects outlined above in 6.4.6, through the Dapper framework. This access is direct tabular access where queries are written and actioned against the database directly. This method of access is the simplest and most direct.

6.5.2 Stored Procedures

Where the complexity of required data access is exceedingly high, data access will be performed through stored procedures. These procedures sit between the API and the database and abstract away the potential complexities of the required data access. These will be very uncommon but may be needed in situations such as where multiple tables across multiple databases are being queried together and against each other simultaneously.

6.6 Identity Architecture

< -- content redacted: Specific definitions removed -- >

6.7 Communication Architecture

< -- content redacted: Specific definitions removed -- >

7 Background Processing Architecture

< -- content redacted: Specific defnitions removed -- >

8 Scalability Architecture

8.1 Overview

This section describes the as-implemented scalability architecture for the Technical Platform solution.

The software and hosting topology has been specifically designed and implemented to adapt to vastly differing scalability requirements, ranging from small to massive user bases.

To achieve this, five scalability strategies have been employed:

1. Division of labour, to subdivide the solution into multiple isolated and independent compute units.
2. Use of background processing, to offload long running and complex tasks from the primary APIs.
3. Hosting flexibility, to allow for a variety of configurations, optimising for various load scenarios.
4. Capacity scaling, in three separate dimensions. Scale up (increased compute), scale out (more instances), scale across (additional regional clusters).
5. API patterns for scaling across: Two patterns have been implemented within the software codebase allowing APIs to orchestrate with their peers (in scale across scenarios) in ways tuned to the workload expected for that API. These patterns are:
    a. Command Query for Global Fast Read
    b. Regional Compute with Sharded Data for transactional performance of data with limited scope

8.2 Division of Labour

This strategy divides workload into logical units based on discrete feature sets.

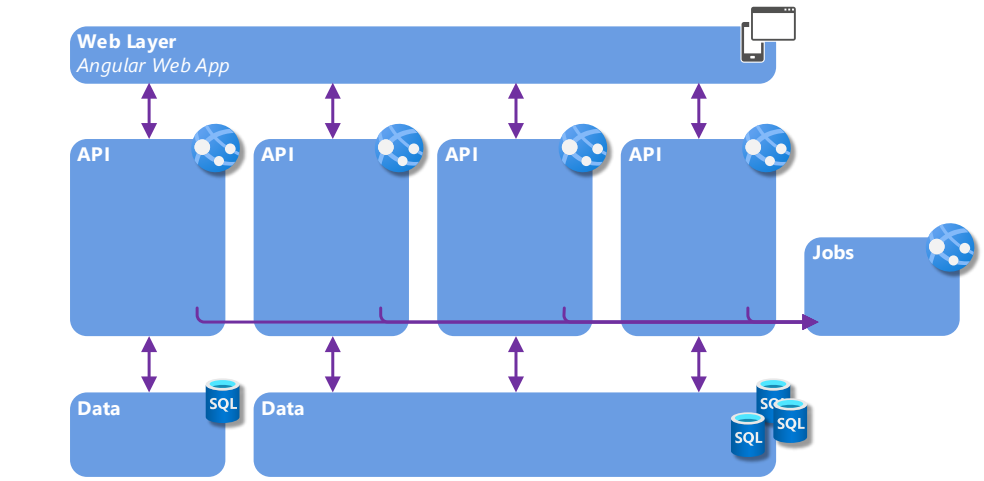

Figure 10: Division of labour

Firstly, the web layer uses a rich web app pattern to push more user experience and user interface rendering workload away from the server and onto the client's web browser.

The API layer is then split into five logical units:

< -- *content redacted: Specific defnitions removed* -- >

8.3 Use of Background Processing

This strategy offloads long running and compute intensive tasks to four groups of background jobs. This is done with defining and scheduling the jobs from the database, covering four broad categories of jobs:

< -- *content redacted: Specific defnitions removed* -- >

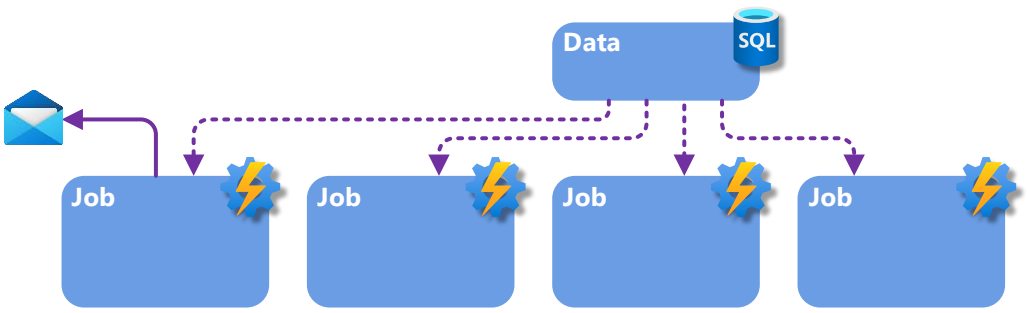

Figure 11: Background processing

This ensures that the APIs can always respond rapidly to requests, and that as the workload requirements increase, these jobs can be further offloaded to separate worker hosts.

8.4 Hosting Flexibility

This strategy employs fundamental design features of the Azure platform to provide deployment and hosting flexibility.

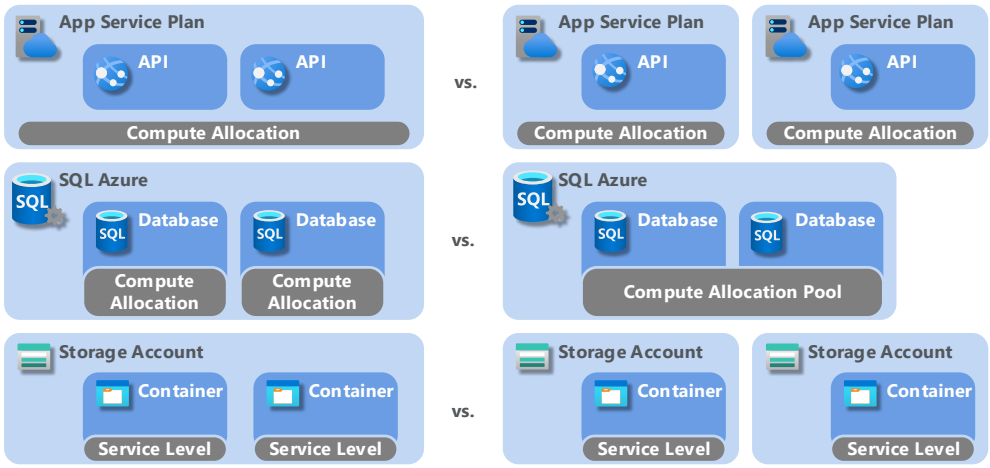

Figure 12: Hosting flexibility

Each of the key server-side host types can be configured in various ways, either consolidating services to share compute, or separate services to isolate and scale various compute needs independently.

8.5 Capacity Scaling

This strategy employs two fundamental design features of the Azure platform: Scaling-up and scaling-out. It also adds a third which has been designed into the codebase allowing the ability to also readily scale across regions.

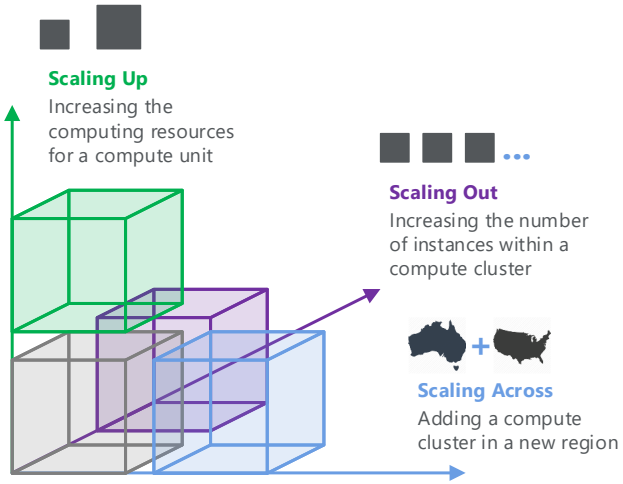

Figure 12: Scaling options

Specifically:

- Scaling up increased the computing resources for a given compute unit.
- Scaling out increases the number of instances of a compute unit within a compute cluster.
- Scaling across adds new compute clusters in other regions around the world.

8.6 API Patterns for Scaling Across

There are two very distinct styles of workload in the API. Therefore, two separate patterns were developed to capitalise on this, thereby maximum effectiveness of any attempt to scale across regions.

Command Query

This pattern is used by the API. This pattern maximises the speed of read requests anywhere around the world. Write requests are potentially slower, but as these will make up a small proportion of API calls, with heavy write workloads being handled from the API. A single write source also ensures that write conflicts due to multi region-deployments regions do not occur.

The key points of this pattern are:

- A single region-based API is declared as the primary API globally.
- All others are secondaries.
- Users assess the API nearest to them
- Read requests are serviced by that region
- Write requests are redirected to the Primary API.
- Data is then replicated globally to secondary APIs using SQL geo-replication, which has an latency SLA of below 5 seconds.

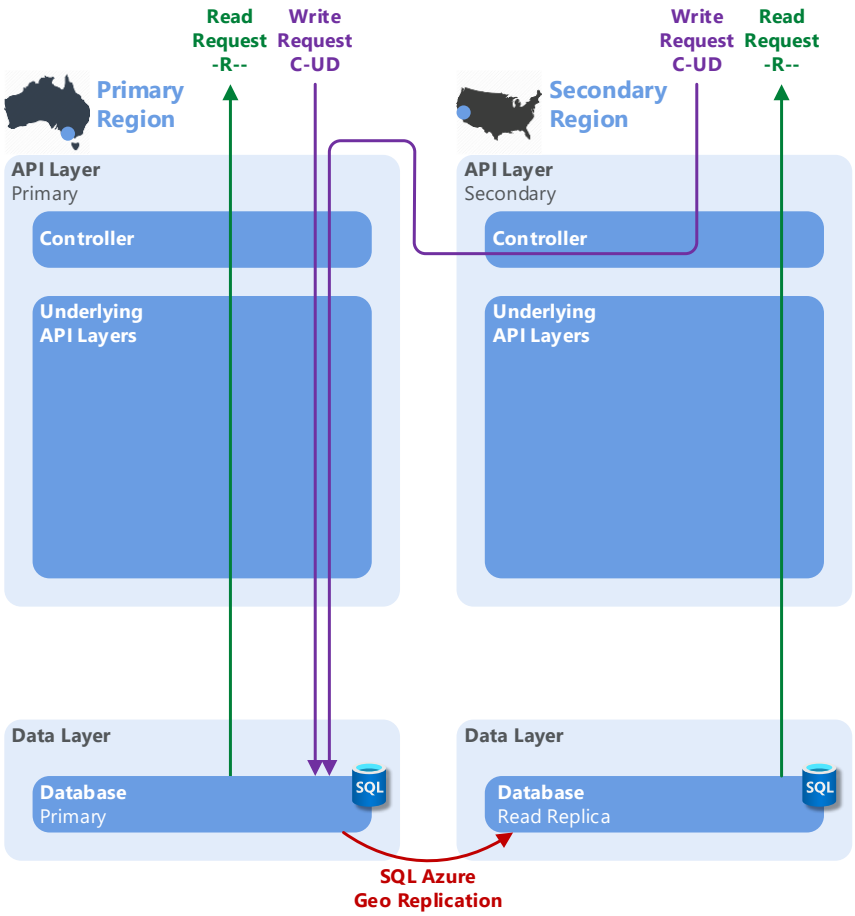

Figure 12: Command query pattern

Regional Compute with Sharded Data

This pattern prioritises data sovereignty, localising data persistence to regions specific to the location of workspaces. Data requests are made directly from a user's location to the region housing the workspace. Data is then further segmented into shared. This allows high performance and sovereignty based on the workspace's preferred region. Workspace data is not replicated beyond the owning region.

The key points of this pattern are:

- Workspace data is localised to its region of creation.
- Data is sharded at a workplace level.
- Requests are made to the directly to the region owning the workspace.
- Performance is high when the user is located in the same region as their data, but may be slower when elsewhere in the world.

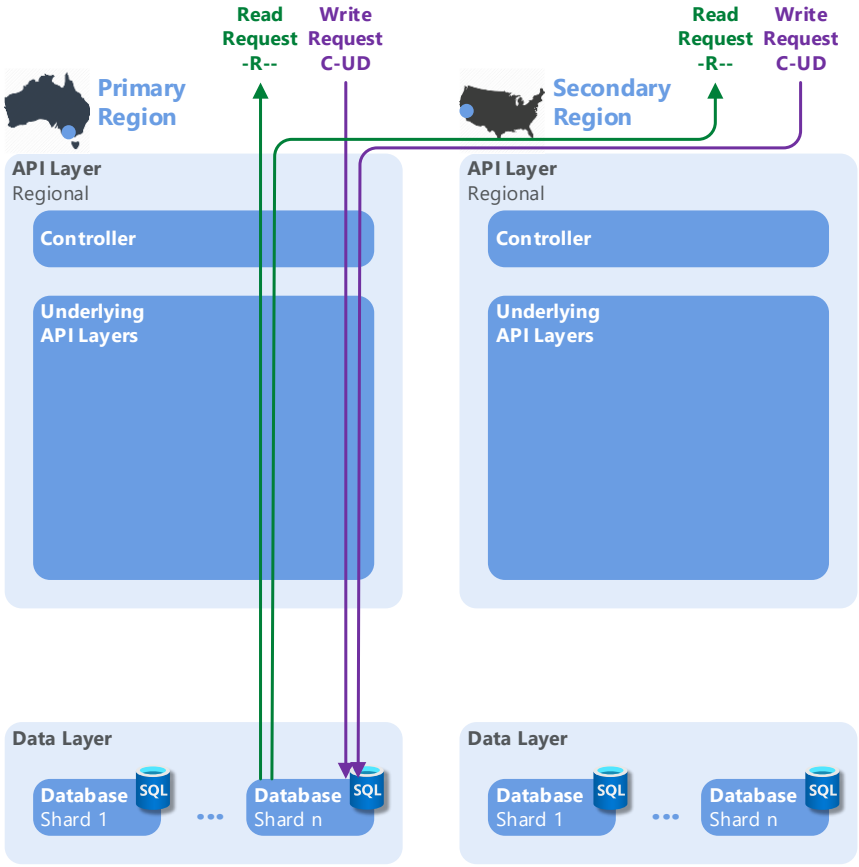

Figure 13: Regional compute with sharded data pattern

9 Software Delivery Architecture

< -- content redacted: Haha … No I'm not going to give you the answer to your project! -- >

10 Security Architecture

10.1 Overview

This section describes the implemented security architecture of the Technical Platform.

This security architecture can be broken into the following concepts:

- Identity: The digital identity of a user.
- Credentials: The proof of identity a user supplies at login.
- Authentication Provider: The process used for a user to supply credentials and receive an authentication token.
- Authentication Token: The token a user receives as ongoing proof of identity, held by the browser for the lifetime of the browser's session and/or token's lifetime.
- Authentication Verification: The points of the solution which verify a user's identify from the Authentication Token.
- User Authorisation: The permitted set of application actions available to an authenticated identity.
- Platform Authorisation: The permitted set of operations available to an unauthenticated platform service, requesting access through use of an API key.
- User Accessible Endpoints: Parts of the application which can be reached by a user.
- Back End Endpoints: Parts of the application which can only be reached by other parts of the application stack.
- Host Configuration: Additional controls and restrictions imposed through platform host.

10.2 Implementation

< -- content redacted: Specific defnitions removed -- >

11 Data Privacy and Sovereignty Architecture

11.1 Overview

If we expect users to trust the Technical Platform and provide sensitive personal information, we need to take proactive steps to earn and retain this trust. For this reason, user privacy and data sovereignty has been deeply considered and designed into the foundations of the platform.

< -- content redacted: Specific defnitions removed -- >

12 User Rights and Protections Architecture

12.1 Overview

In addition to security and privacy controls, the Tech platform includes several user rights and protections controls. These include:

- The right to request personal data stored by the Tech Platform
- The right to change sign in identity
- The right to be forgotten
- Terms of use prevention for persons under the age of 18.

12.2 Implementation

< -- content redacted: Specific definitions removed -- >

13 Known Limitations and Closing Remarks

< -- content redacted: Specific defnitions removed -- >