BuddyToys

Online Pet Shop

Architecture Document

Background

This document describes the BuddyToys web application architecture, the young and promising BuddyToys companys' main product.

BuddyToys sells everything from the pet industry - from pets to toys and equipment.

Because of the pandemic, people started isolating themselves, working from home and lowering their social communications. These things lead to many mental problems, like the increased cases of depression - a dangerous medical illness. Here come in help, pets. They can reduce stress levels drastically and help people.

The website will sell pets, toys, and other equipment they might need. The system should be with very low downtime. It should be prepared to meet a high increase in concurrent users during sales and marketing campaigns without failing because this would cost us money and users trust.

This document describes the system architecture of BuddyToys's application.

The architecture comprises technology and modelling decisions that will ensure the final product, assuming the architecture is followed, will be fast, reliable, scalable, and easy to maintain.

The document outlines the thought process for every aspect of the architecture and clearly explains why specific decisions were made.

The development team must follow the architecture depicted in this document closely. In any case of doubt, please consult the Software Architect.

Requirements

Functional requirements

- The system will have authentication / authorization mechanism
- Provide sales and other important data to BI analyzer
- Users should be able to add items to card
- Users should be able to buy items
- Forum
- Users should be able to search clinics
- Users should be able to rate, comment and see info about clinics
- Payments will be made through an external API
- Deliveries will be made through an external API

Non-functional requirements

- The system should be able to scale for hundreds of concurrent users
- The database will grow by 300GB annually
- SLA: The system should be available as much as possible

Executive summary

This document describes the new BuddyToys web application architecture, an online pet shop, allowing end-users to buy all kinds of pets and toys for them, bringing them comfort and happiness.

When designing the architecture, a strong emphasis was put on two major features:

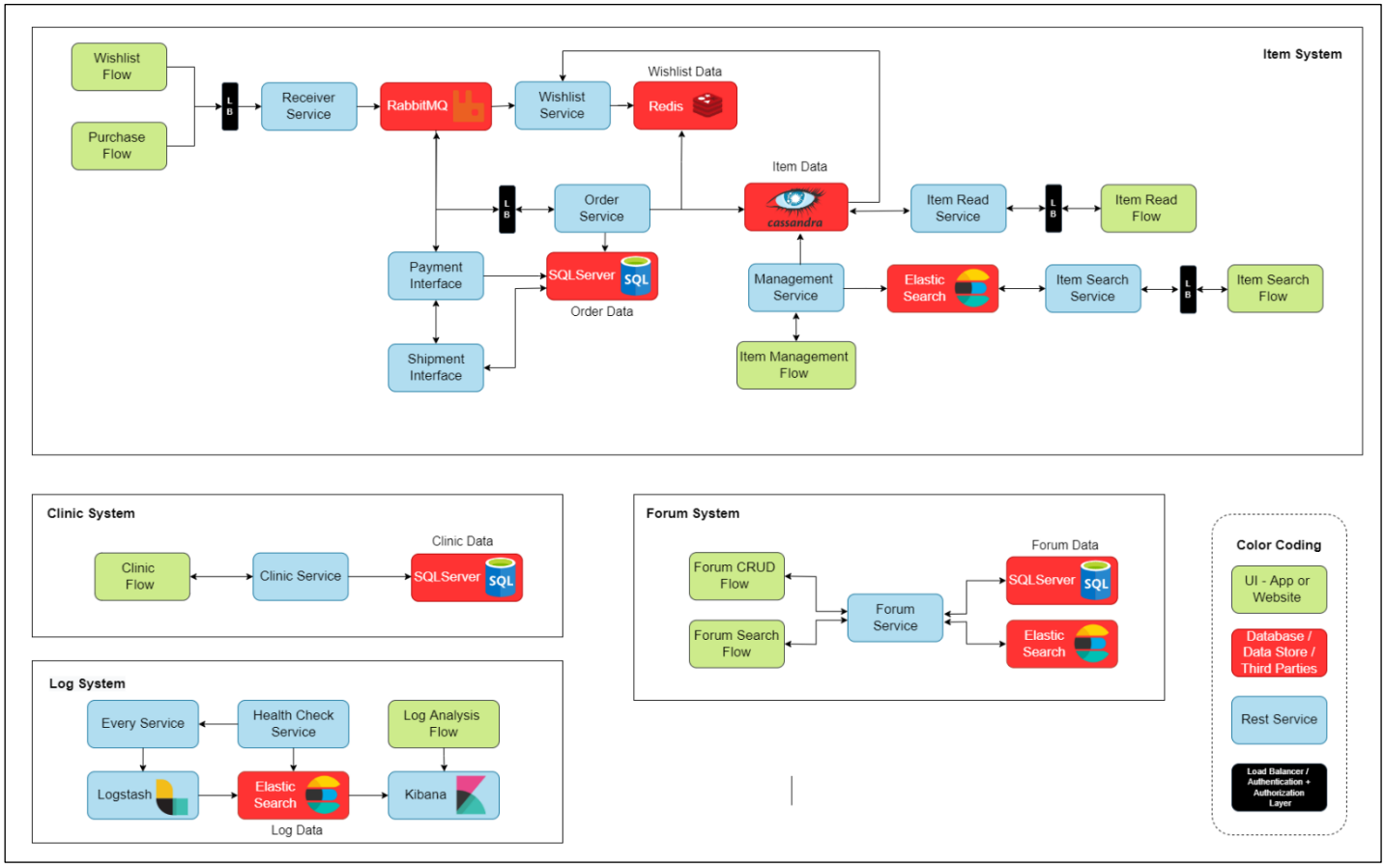
- The application should be reliable
- The application should be scalable

The architecture is based on the most up-to-date best practices, methodologies and technologies to achieve these features, ensuring high-availability and performance.

The application type is going to be SPA (Single Page Application). **React** was chosen as the frontend framework.

For authentication was chosen **Firebase**.

Here is a high-level overview of the architecture:



As shown in the diagram above, the application comprises four separate, independent, loosely-coupled services. Each has its own tasks, and communicates with the other services using standard protocols.

All the services are built as stateless services, meaning – no data is lost if the service is suddenly shutting down. The only places for data in the application are the Queue, Redis, and the Databases, all of them serialize the data to the disk, thus protecting it from shutdown cases.

In conjunction with a modern development platform (**.NET Core**), this architecture will help create a modern, robust, easy to maintain, and reliable system that can serve the company successfully for years to come and helps it achieve its financial goals.

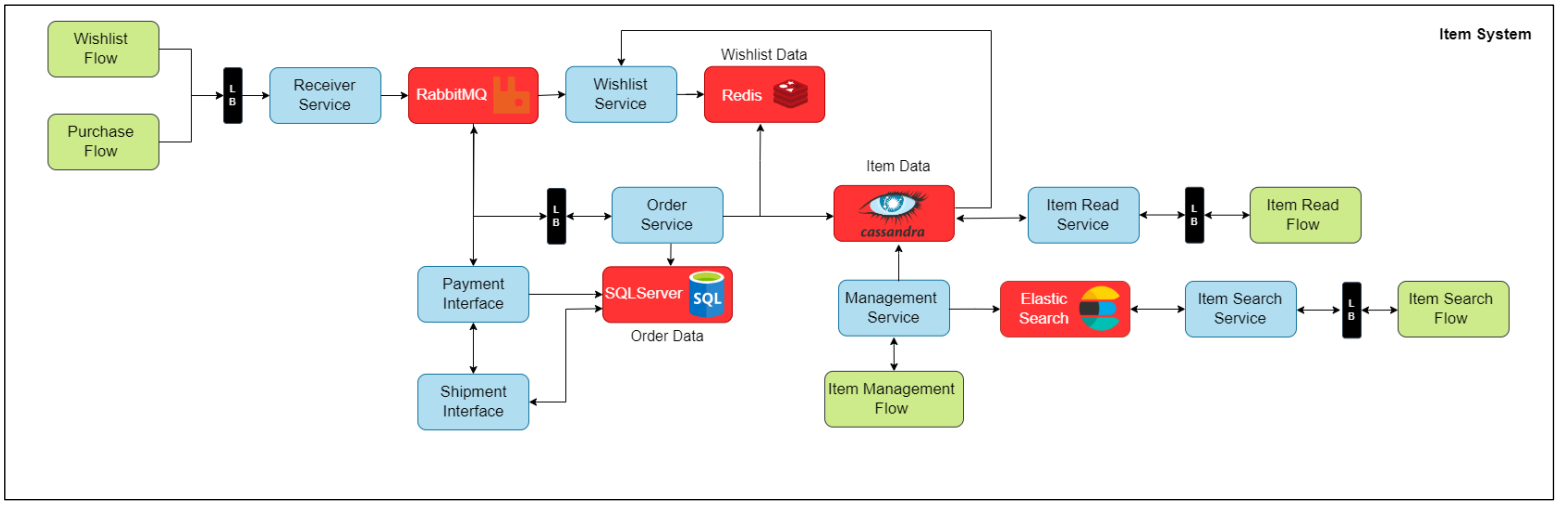
Overall Architecture

Here are the architecture diagrams for the BuddyToys application:

Diagrams from 1 to 4 show a high-level view of the user interaction flows with the system and the interactions between the technologies involved.

0. Legend

1. Item System



That is our product's layer. It is the heart of our business. Our profit comes from it, and users come to us for it. Thus we have to make sure that it is scalable and resistible. We need to invest most of our resources into it.

Services

The system comprised of the following services:

- **Receiver** Service:
  Receives wishlist and order's data and adds them to a Queue for further handling. The receiver puts a strong emphasis on performance.

- **Wishlist** Service:
  Validates and adds products to the wishlist. Pulls data from the Queue (where the Receiver Service placed them).
- **Order** Service:
  Validates and adds orders. Pulls data from the Queue (where the Receiver Service placed them).
- **Payment Interface**:
  Payments are made through an external API, so Payment Interface communicates with it.
- **Shipment Interface**:
  Shipments are made through an external API, so Shipment Interface communicates with it.
- **Item Read** Service:
  Reads Items data and provides it.
- **Management** Service:
  Processes CUD requests (Create, Update, Delete) for Items. We already have a dedicated service for reading. Therefore this one is responsible for the other CRUD operations. Adds every new Item to Casandra Database and Elastic Search Cluster.
- **Item Search** Service:
  Sends search queries to Elastic Search and returns the results.

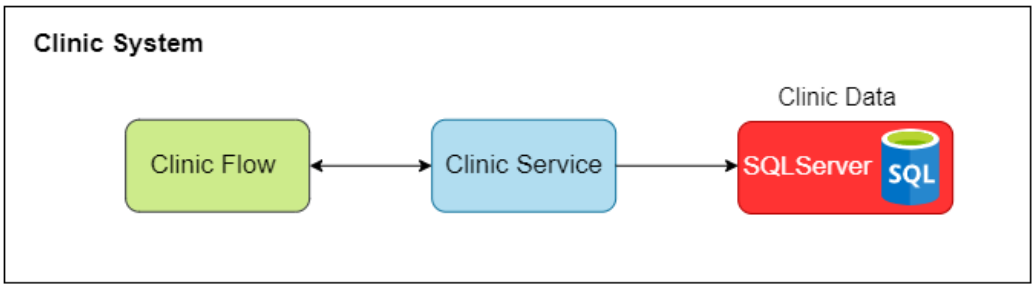
Messaging

The various services communicate with each other using multiple messaging methods. Each method got selected based on the specific requirements of the services. Here are the different messaging methods used in the system:

- **Receiver** Service:
  Receives data through HTTP and forwards it to RabbitMQ**.**
- **Wishlist** Service:
  Receives data from RabbitMQ and forwards it to Redis.
- **Order** Service:
  Receives data from RabbitMQ and sends it to Cassandra Database.
- **Payment Interface**:
  Receives data from RabbitMQ and sends requests to external API through HTTP.
- **Shipment Interface**:
  Receives data through HTTP and sends requests to external API through HTTP.

- **Item Read** Service:
  Communicates through HTTP Rest API.
- **Management** Service:
  Communicates through HTTP Rest API.
- **Item Search** Service:
  Receives data through HTTP and sends requests to Elastic Search. Returns results again through HTTP.

2. Clinic System



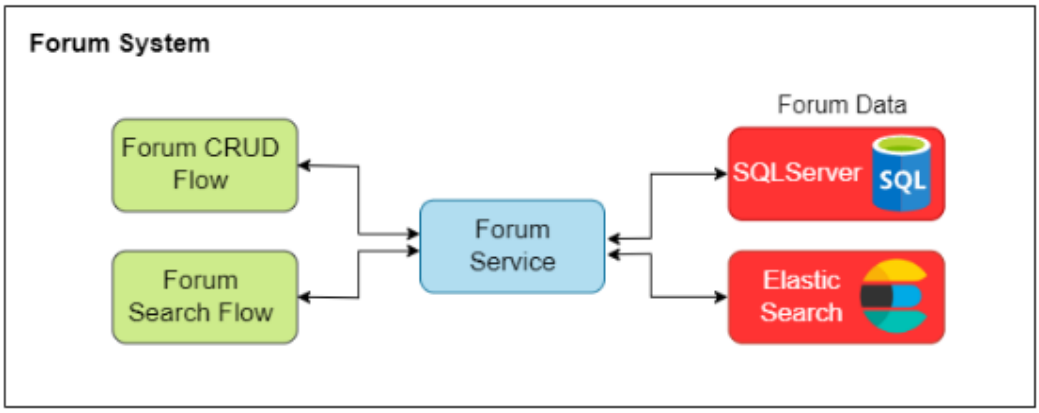
Services

- **Clinic** Service:
  Processes CRUD Request for Clinics.

Messaging

- **Clinic** Service:
  Communicates through HTTP Rest API.

3. Forum System



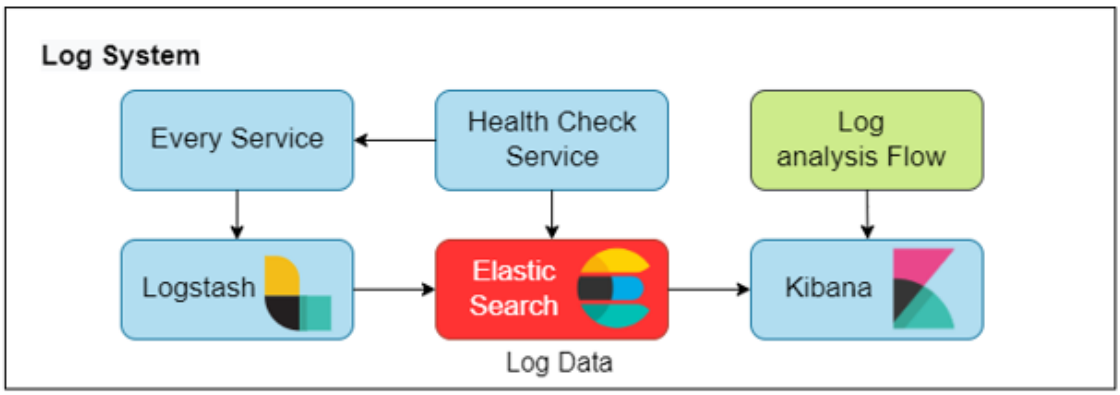
Services

- **Forum** Service:
  Processes CRUD Request for Forum. Adds every new Item to SQLServer Database and Elastic Search Cluster.

Messaging

- **Forum** Service:
  Communicates through HTTP Rest API.

4. Log System



Services

- **Every** Service:
  Not actual service but abstraction over all the other services. It just explains that the other services are expected to send log reports to Logstash.
- **Health Check** Service:
  Sends requests to all services and checks if they are alive. For better abstraction, we can have such a service per each system.

Messaging

- **Health Check** Service:
  Makes requests through HTTP.

Scaling

This architecture allows efficiently scaling the services as needed. Each service can be scaled independently, either automatically (by service managers such as Kubernetes) or manually.

Also, the service's inner code is fully stateless, allowing scaling to be performed on a live system without changing any code lines or shutting down the system.

Services Drill Down

Item System

- ### Receiver Service

Role

The Receiver service is the service that client side applications access to send their purchase or wishlist requests.

To make the Receiver as lightweight and fast as possible, its designed functionality is very focused and very minimal. The Receiver's only task is to receive requests and push them into a queue. Other services are responsible for taking care of the messages afterwards.

Technology Stack

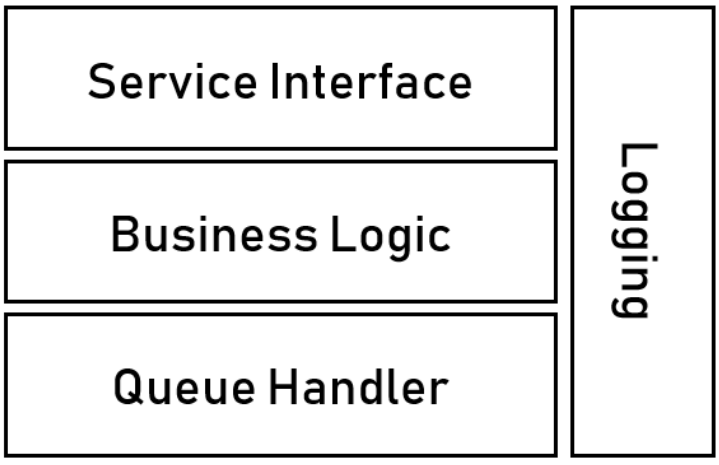
The Receiver is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer, makes sure the messages are valid and passes them to the Queue Handler layer.

Queue Handler – This layer replaces the classic Data Access layer since there is no data store in this service. The Queue handler receives the messages from the Business Logic layer and adds them to the Queue. The Handler service is waiting on the other end of the queue to handle the messages.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

Implementation Instructions

- The Service Interface layer should contain as little code as possible. No logic should occur, and its only task is to receive the messages and pass them to the Business Logic layer.
- Every step in the service must be logged. Use the logging component generously. Since there is no UI for this service, logging is the only way of figuring out what's going on.
- Use dependency injection between the various layers. Implement the dependency injection using `Microsoft.Extensions.DependencyInjection` package.

API Endpoints

The following table displays the API actions available by the Info service:

Functionality	Path	Return Codes
Send wishlist request	POST `/api/wishlist?userIdToken=userIdToken&requestData=requestData`	200 OK 404 Not Found

Send order request	POST `/api/order?userIdToken=userIdToken &requestData=requestData`	200 OK 404 Not Found

- ## Wishlist Service

Role

The Wishlist service is responsible for validating and storing wishlist requests in the data store.

The Wishlist listens to the Queue containing the messages, grabs them, and then handles them. (The messages were placed in the Queue by the Receiver service).

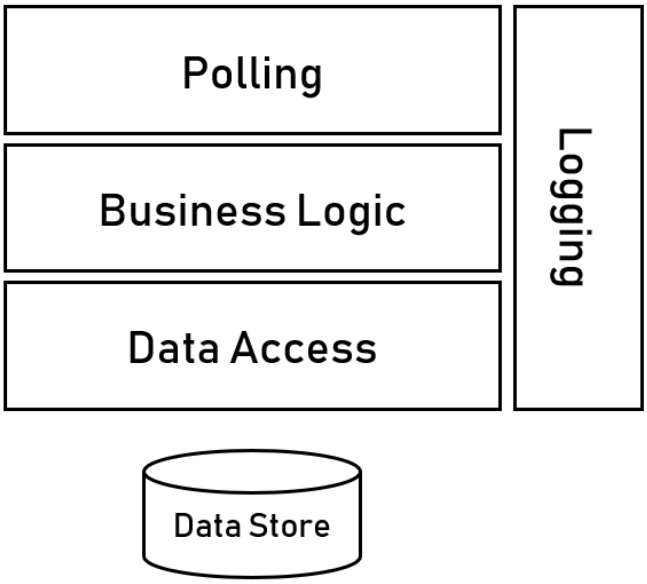
Technology Stack

The Wishlist service is a Worker service, not a Web API or Web app since it's always active and does not wait for requests to return a response but instead polls the queue for new messages, basically initiating the action instead of waiting for one.

There is no real reason to select other technology for this service since there are no special requirements.

Architecture

Here is the architecture of the Wishlist service:



Polling – Manages the work with the messages' queue. This layer polls the queue for new messages, and when such messages are received, it immediately passes them to the business logic layer.

Business Logic – Receives the requests from the Polling layer, validates them and passes them to the Data Access layer.

Data Access – This layer stores wishlist data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible. The log records are added to a queue, which is polled by the Logging Service.

Every step in the service must be logged. Use the logging component generously. Since there is no UI for this service, logging is the only way of figuring out what's going on.

- ## Order Service

Role

The Order service listens to the Queue containing order requests, grabs them, and then handles them. (The messages were placed in the Queue by the Receiver service).

After receiving a request, validates it and then stores order data.

Technology Stack

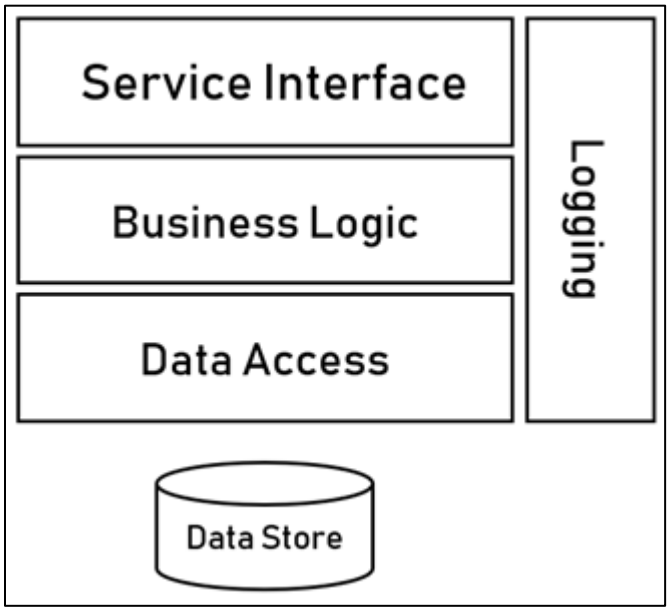
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

- ## Item Read Service

Role

Solely provides Item data. Because reading is much more often a process, we need a service whose only role is reading.

Technology Stack

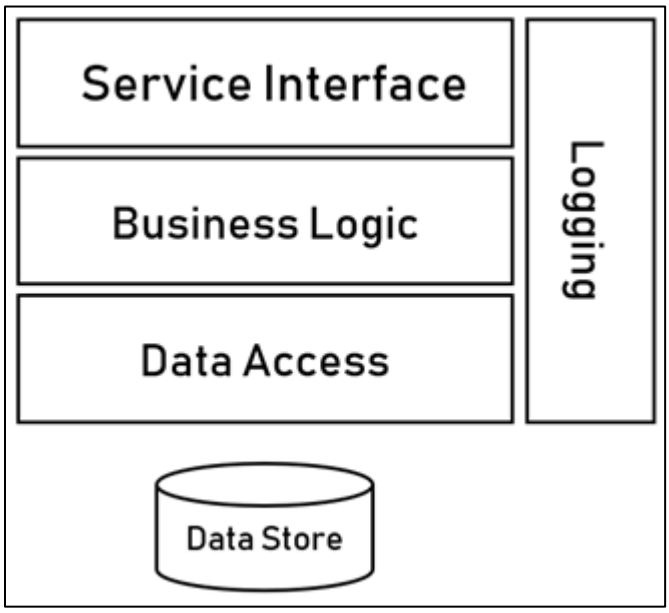
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

- ## Payment Interface

Role

Communicates with external payment API service.

Technology Stack

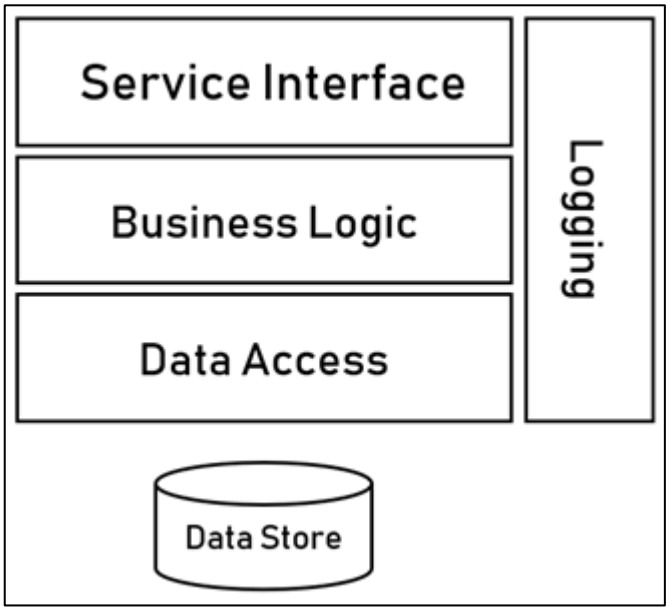
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

- ## Shipment Interface

Role

Communicates with external shipment API service.

Technology Stack

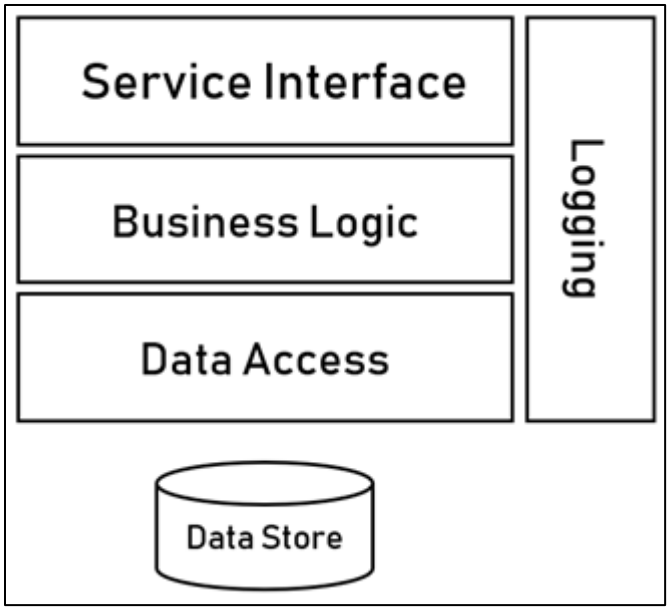
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

- ## Management Service

Role

The Management service processes Item CUD requests. It does not bear single responsibility because Create, Update and Delete requests, are not expected to be very often.

Technology Stack

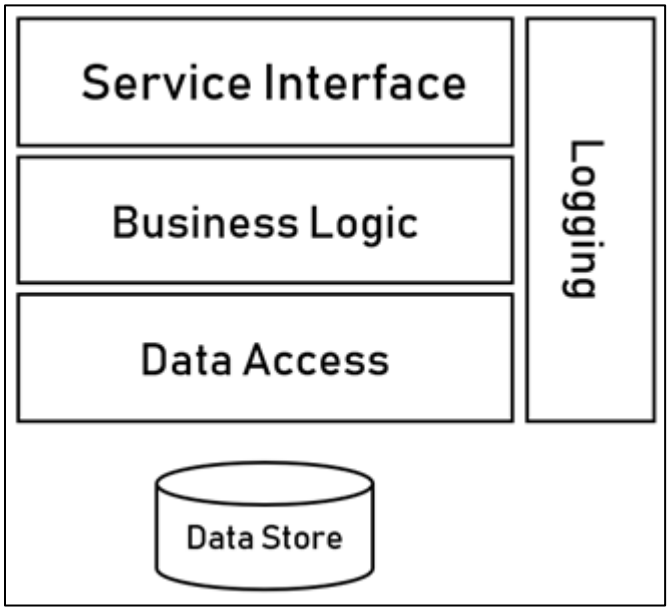
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

- ## Item Search Service

Role

The Item Search service processes Item search requests by sending requests to Elastic Search.

Technology Stack

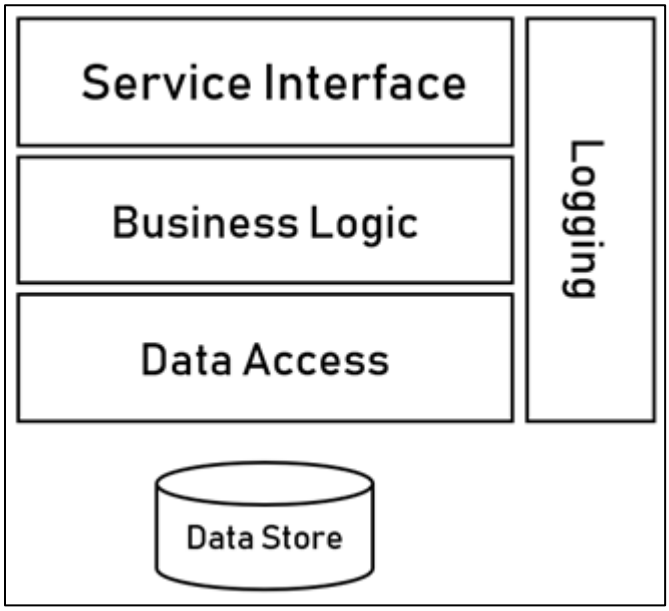
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

Clinic System

- ## Clinic Service

Role

The Clinic service processes vet clinic CRUD requests. It does not bear single responsibility, but this is because it is not business critical.

Technology Stack

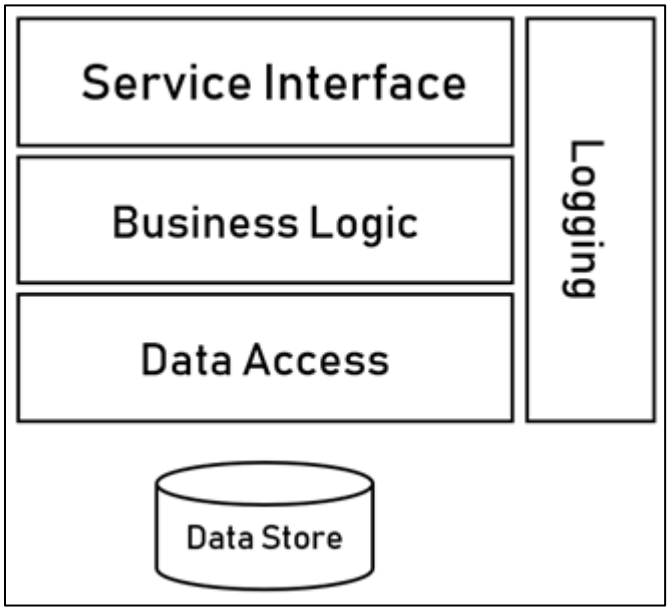
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores clinic data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

Forum System

- ## Forum Service

Role

The Forum service processes forum CRUD requests. When user searches forum post Forum Service's task is to make a request to Elastic Search and return the result. It does not bear single responsibility, but this is because it is not business critical.

Technology Stack

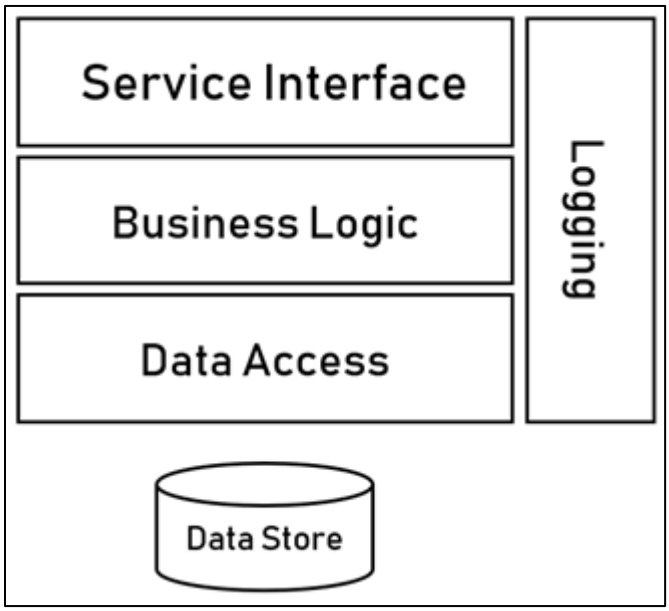
It is a Web API service since clients will communicate via REST API.

So when selecting the technology stack, we must ensure it supports the Web API scenario.

Our dev team is .NET oriented. Therefore our best choice is to base this Service on ASP.NET Core, and we will need an excellent reason to divert from this decision.

Architecture

Here is the architecture of the receiver service:



Service Interface – Exposes REST API. This layer receives the requests and immediately transfers them to the business logic layer.

Business Logic – Receives the requests from the Service Interface layer then processes them and passes them to the Data Layer.

Data Access – This layer stores forum data in the data store.

Logging – This is not an actual layer but a cross-cutting component (meaning – it's accessible by all the layers). This component contains the service's logging library and exposes a simple API to log as fast and straightforward as possible.

Logging System

- ## Logging

Role

For logging, will be used the ELK stack.

All services send Logstash their logs, and he stores them in a central data store (Elastic Search).

Logs, are stored in a queryable data store, and log analytics will be made through Kibana.

This way, the developers can get a unified view of all the system events and are not required to go through different log files in various formats to get a coherent picture of a specific flow or error.

Technology Stack

- Logstash
- Elastic Search
- Kibana

- ## Health Check Service

Role

Sends requests to all other services to make sure they are alive.

In case any service is down must send an email or other notification on which could be immediately reacted.

To make sure there will always be active Health Check Service, we will use the Active-Active architectural pattern.

Both the instances will be up and running, but one of them will be the primary one, doing the job. The other will stay silent and check whether the first one is up and running. If, there is a crash, the second instance will see the problem, become the primary one, and start taking its role. We can configure the operating system to restart automatically, crashed services. After it is back online, the first instance can become the “silent” one, and just health check the other.

Technology Stack

Because the health checking service is always active service and does not need UI, it will be a Worker Service.

As such, there are not many requirements for its implementation technology.

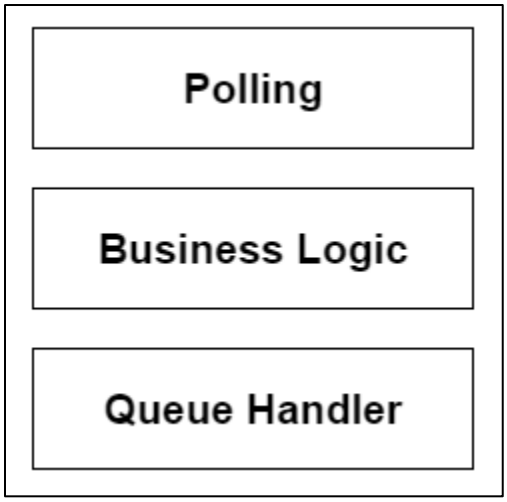
The technology needs to be able to send HTTP requests and store data in a data store. That is nothing special, and any development platform can do that. Also, there are no special requirements for performance. Of course, we want it to be fast, but there is no specific requirement that limits us here.

The development team is familiar with the Microsoft stack, meaning - .NET platform.

Thus this too will be based on the .NET technology.

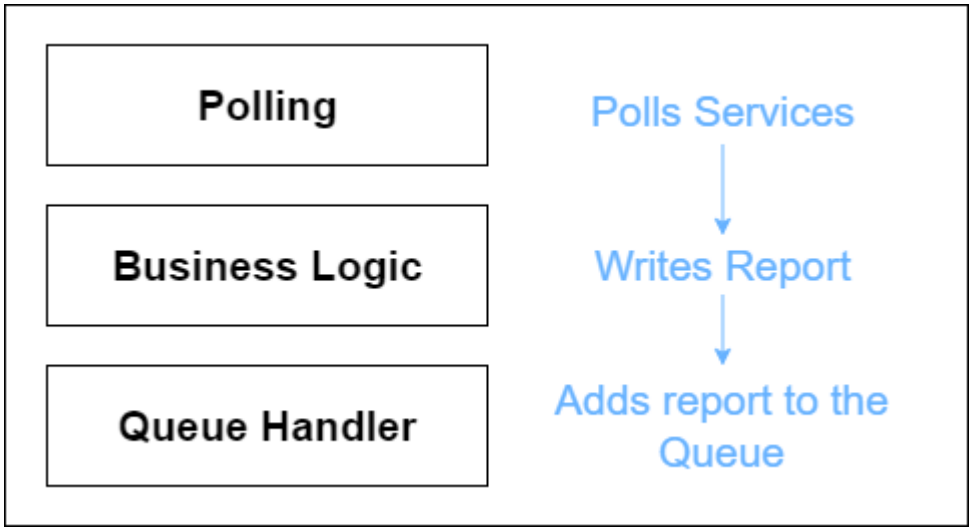
Architecture

Here is the architecture of the health check service:



As you can see, this is a classic layered architecture with a significant change - the top layer, which usually is a service layer, is a polling layer in this service. The reason is that since the service is not a Web API or Web app, it does not expose any interface and therefore does not need a service interface layer. Instead, its polling layer manages the requests to all services. Also, the bottom layer is Queue Handler, which usually is the data layer, now is Queue Handler because it will work with a queue.

Here is the architecture with a description of every layer:



Note that every layer has a well-defined role. It's crucial to keep it this way and make sure no layer interferes with other layers, which will make the service much more difficult to maintain.

Implementation Instructions

- Project type should be Worker Service
- Requests should be made through HTTP

- Use the Elastic.Clients.Elasticsearch package for accessing the database.
- Send an email with Gmail with SmtpClient.