Technical Recommendations

PROJECT NAME: **POGO GAKKŌ**
PROJECT STAKEHOLDER: **Steve Bulgin**

After careful analysis of the requirements in the Creative Brief, the following document lists Team Tomodachi's formal Technical Recommendations for the POGO GAKKŌ project.

Overview

To improve maintainability for the application and to allow for increased ability to improve upon the fundamentals, we recommend splitting this application into a multi-layered service architecture consisting of separate Front-end and Back-end components.

Project-wide

- Source Control: GitHub

Front-end

Our recommended technology stack for the front-end is the Ionic Framework alongside React for styling, layout, and interactive functionalities.

Ionic will allow us to create a multi-platform progressive web application that React will let us develop in a modular fashion. This means we will more easily be able to build a cohesive and uniform application that will work across all platforms (e.g. Windows OS, Android, iOS). Moreover, Ionic and React include interface modules and styles that cater to mobile applications, and will allow us to more easily integrate mobile-specific interactive features.

As a UI toolkit, Ionic provides a framework that will save us overall development time and allow us to quickly create a functional and aesthetically pleasing interface with mobile-specific interactive features. React also specializes in modular user interface development, and so it should pair well with Ionic to meet the challenges of this project. These technologies will handle a good portion of the front-end for us, and will allow us more time to focus on the specific needs of the project.

By default, Ionic utilizes the Angular framework for much of its functionality. As of Ionic 4, support has been added to instead use other frameworks such as React. We have decided to use React over Angular for several reasons; React seems better suited for the development of modular interfaces as envisioned for this application. In addition, the

members of our group either have previous experience with React, or are expecting to become familiar with React in the near future.

The drawback to using React instead of Angular is that there is less documentation on its use within Ionic, as it has only recently become supported. It may require some extra work and research for us to get React working to support certain functionalities. As such, there may still be a need to rely on Angular in specific cases.

An Alternative?

We had React Native as an alternative instead of using Ionic altogether, which would improve speed and performance for individual platforms. The two problems with this, however, are compatibility and time. Mobile-specific code would have to be rewritten for all platforms, which would make development and testing more complicated and time-consuming. If performance becomes an issue in the future, there is always the option to convert the code following the completion of this project.

Back-end

Our recommended technology stack for the back-end is based on ASP.NET Core. Based on the modular nature of this project, we decided that a microservice architecture would be a perfect fit. ASP.NET Core is robust and cross-platform; we can run our microservices in Docker containers on an affordable VPS, and maintain the possibility of easily moving to a larger scale solution like Kubernetes.

Some of the challenges that a microservice architecture presents include the following: Service discovery, which is the process of connecting to other services that are dynamically allocated; interfacing with the client application without making a wasteful number of HTTP requests from the browser; authentication and authorization to ensure that users are logged in and have access to the content they are requesting; and asynchronous communications between services, to allow for data consistency (eventual) without unnecessarily long response times.

To accomplish the objectives in the Creative Brief. The following technologies in this section are our recommendations for the back-end portion of the POGO GAKKŌ Project.

General Technology

- ASP.NET Core Web API
- Entity Framework
- PostgreSQL
- Json/GraphQL

Service Discovery

Service Discovery Registries like Consul enables scaling services horizontally with load balancing, as well as service health monitoring.

Client Application Interface

An API Gateway such as Ocelot enables a client application to consume the collective API of a microservice architecture without excessive numbers of HTTP requests and complicated client-side service discovery. In addition, cross-cutting concerns such as caching, authentication, and authorization are implemented here. GraphQL is a substitute for a REST API that allows a client to request exactly the data it needs, without requiring a new handler on the server-side.

Authentication and Authorization

Identity Server will allow us to quickly get OAuth2 Authorization and Authentication online within the POGO Gakko application. It will also provide us with the ability to quickly integrate external OAuth2 providers including Discord and Google. Ocelot and IdentityServer4 integrate together to keep the application and its users secure.

Asynchronous Interservice Communication

Using RabbitMQ allows us to publish messages when something interesting happens on one service, and other services can subscribe to these messages to keep their data up to date, or otherwise react appropriately. This promotes loose coupling between services, while enabling the application to remain a single system.

Alternatives and Future Development

While the technology recommendations outlined in this document are what the core of the application will be developed using. Due to how the Microservices pattern works, future development is not limited to these technical recommendations.

As a result, additional components added to the back-end portion of the POGO GAKKŌ project can be written using any language or technology stack which are deemed necessary for that component. This is due to Microservices being their own self-contained components which operate independently with each other using a common core interface.