University of Waterloo

SE464

Section 001

Deliverable 4: Project Architecture & Design

Group 25

December 1, 2016

1 Metadata

Project: ShuttleQL (Shuttle Queueing Logistics)
Team Name: Baddie Boys
Team Members:

- Cheng Dong (c9dong)
- Zhaotian Fang (z23fang)
- Clement Hoang (c8hoang)
- Di Sen Lu (dslu)

2 Introduction

Around the world, recreational badminton clubs host regular sessions at set locations for club members to drop in and play at. Traditionally, many of the tasks related to the operation of the club are done manually with the combined effort of the club's executive staff. Sometimes these tasks are repetitive and tedious and may require a execs full attention for a significant period of time. Examples of these tasks include:

- Registering new club members
- Checking in and out members during a club session
- Scheduling members into courts for each rotation

ShuttleQL is an all-in-one badminton club management platform that aims to automate many of the tasks above. The web-based platform consists of two main components:

1. Mobile web-based dashboard for the club members
2. Desktop web-based administration panel

The online platform will act as a communication hub to increase transparency between the execs and members as well, notifying members of particular events in real time.

3 Architectural description

3.1 Overview

At a high-level, the architectural components can be split into two groups: online and offline. The offline components consist of the matchmaker and pigeon, while online components make up the rest. The online components can be further divided into four different layers, as shown in the table below

Layer	Component(s)
A	Player dashboard, Admin panel
B	API Gateway
C	User, Session, Announcement, Match service
D	User, Session, Announcement, Match database

The online components make up a modified version of the three-tiered pattern. The front, middle, and back tiers are represented by layers A, C, and D respectively. Layer A contains the two web-applications that provide the user interfaces for the players and admins. Layer C contains the application functionality, separated based on the kind of data that is being operated on as described by the service name. Layer D contains the application data access and storage capabilities in the form of different databases where each database contains a specific category of data, which maps 1:1 with a service in layer C. In terms of communication, layer A communicates with C through HTTP and C with D through an FRM called Slick. The responsibility of layer B will be explained later when the report focuses on the architecture of the API gateway.

The online components also make up a layered style. If we assume the layers A to D are stacked from top to bottom, then a layer acts as a server to the layers above and acts as a client to the layers below. For example, layer B, the API gateway provides a client API for the web-applications (layer A) to use. In addition, layer B is a client to layer C, which exposes an internal API for the API gateway to use. The advantage to this pattern is reuse in the sense that a change to a particular layer means that you only have to modify at most two layers, the layer directly above and below.

A client-server style is exhibited between layers A and B. Layer B, the API gateway acts as a server that provides a set of user-facing client APIs. It doesn't know the number or identities of clients that will use its API. On the other hand, layer A, the web applications act as the client in the sense that it needs to know the explicit identity of the server in order to utilize the server's provided API.

3.1.1 Player and Administrator Frontend

The Player and Administrator Frontend are the two interfaces where users can interact with the ShuttleQL. The Player frontend provides users with the ability to view current games, whereas the Admin frontend provides club executives with the ability to start club sessions, manage users, and manage court usage. The main patterns used in these implementations are Flux architecture for data flow in the front-end, as well as multi-layered architecture when communicating with the API gateway. The Flux architecture consists of actions, views, reducers, and stores. It is a one-way data flow pattern where views dispatch actions, which are handled by the reducers in order to update the store, and the changes are propagated back to the views from top to bottom. In the views component, several modules are depended on for rendering. Material UI provides aesthetically pleasing components for the user interface, Router handles the rendering logic based on the URL supplied by the client, and React provides HTML rendering logic. In the Actions component, Axios is a module used to make HTTP requests, and Token Manager is an internal module used to authorize outgoing HTTP requests. Finally, there is a Socket.io client that listens to broadcasts from Pigeon.

3.1.2 API Gateway

The API Gateway serves as the interface that the frontend interacts with the backend services. The individual Scala services do not interact with each other, but with the API Gateway. This ensures a microservice architecture and will be explained further in the design.

3.1.3 User, Session, and Announcement Service

The User, Session and Announcement services are the Scala services that interact with the databases. Each service is a standalone Scala application and has its own Postgres database to preserve a microservice architecture. The services communicate with the database via Slick, an ORM library for Scala.

3.1.4 Pigeon

In a nutshell, Pigeon is the component that notifies interested parties about events that happen in ShuttleQL. For the MVP implementation, Pigeon is used to notify front-end clients when updates to the backend models occur, and when a new announcment is created by a club administrator. It is composed of two parts: the Amazon SNS portion, and the Pigeon server. The services send publish requests to Amazon SNS when interesting events occurs, to which Amazon SNS notifies Pigeon Server in response. Then, Pigeon Server broadcasts the message to all its subscribers (Admin Frontend and Player Frontend).

3.2 Functional Properties

From deliverable 1, 5 mandatory and 4 bonus functional properties were mentioned. The system satisfies all 5 mandatory functional properties and 1 of the 4 bonus functional properties.

3.2.1 Mandatory

Three out of the five mandatory functional properties describe admin-level actions such as registering new club members, checking in/out players, and overriding the matchmaking algorithm. They are all accomplished in a similar way by the system. The admin-panel component on the client-side provides a user interface for the admin to perform these actions. Each action is then proxied through the API gateway server. From that point, depending on the data that's being changed, the gateway will delegate to a specific internal service. For example, registering new club members means that the gateway will delegate to user service in order to persis a new user into the database.

Another functional property is that the system also allows the current matches to be displayed for the user to see. This information needs to be surfaced on both the admin panel and player dashboard. Therefore, it needs to be able to call an API to fetch the matches, which API gateway provides. Internally, API gateway delegates to game service to actually fetch the matches. Once the results are returned to the client, the data is rendered in a visually pleasing manner for the user to see.

Finally, the last mandatory functional property is that the system must have a match making algorithm that allocates players to open badminton courts. The matchmaking background process in the system architecture has exactly this responsibility. It will maintain a queue of players and periodically rotate current players off and allocate players that have been waiting onto the badminton courts. The current state of the queue and the mapping of players to courts is exposed to the internal game service only.

3.2.2 Bonus

One bonus functional property that was implemented was the ability for executives to broadcast announcements to users. The system accomplishes this by first providing the UI for the admin to construct an announcement and broadcast it. In addition, the system handles transferring the announcement data from the client through the API gateway and to the internal announcement service, where it'll be persisted. In addition, the system supports real-time message passing to the client through the component pigeon. Therefore, once the announcement is received by the internal service, it can leverage pigeon to send the announcement in real time to all connected players where it will be shown by the player dashboard UI.

A bonus functional property that wasn't implemented is the ability for admins to send notifications to users once they are able to play. This was never followed through since the system instead supports showing the user's status in the session that updates in real time. This feature accomplishes the problem that the original functional property was trying to solve and more.

The other bonus functional properties, the ability to show a match history and to track match-making-rating of club members were dropped since the level of effort needed to extend the system architecture to support these properties was not realistic given the time constraints of the group.

One functional property that was implemented which wasn't mentioned is the ability to keep the client user interfaces updated in real time. This is supported by the system through pigeon, which would notify connected clients that certain data is stale and must be updated through a fresh fetch from the server. The need to support this property is because data such as a user's status in a session, and the current set of games can all change without interaction from the user. Since there's never a need for the user to trigger a refresh, the system must do the job for the user instead.

3.3 Non-functional Properties

3.3.1 Dependability

The first of the two mandatory non-functional properties that have to be met is dependability, which describes the reliability of the system. This property is satisfied if the system can maintain an uptime of 95% or better for every 24 hours. This is accomplished by deploying all backend services using Ngrok. This third-party service not only deploys any backend service but also provides a user interface as a terminal or web app that exposes crucial metrics related to the service such as uptime, server load, etc. Therefore, we're able to monitor the uptime of all deployed services to ensure that the property is met.

3.3.2 Portability

Portability is the second mandatory non-functional property, which describes the ability of a system to execute on multiple platforms while retaining its functional and non-functional properties. The concrete requirement is for the system to satisfy the following functional properties on both Chrome and Safari for mobile and desktop platforms: player registration, player checkin/checkout, matchmaking, and displaying the current matches. The frontend architecture for the admin panel and player dashboard supports the property through its lack of reliance on Javascript APIs with strict browser support. In addition, the architecture is careful to include only external dependencies that have good browser support across multiple platforms (i.e. material-ui).

3.3.3 Efficiency

Efficiency is the first of two bonus non-functional properties, which describes the system's performance. Since system performance is a very general requirement, it was narrowed down to the performance of the matchmaking algorithm. More specifically, the requirement is that the algorithm should run in less than 5 seconds given a pool of 100 badminton players. In order to maximize the efficiency of the algorithm, it is important to decouple the component away from any services so the component can be run in the background on a separate machine, where the hardware specs of the machine can be upgraded to meet the performance requirements of the algorithm. In addition, the algorithm is structured to fetch all needed data before the algorithm begins in order to prevent the need to make networks calls in the middle of the algorithm, which hinders its performance.

3.3.4 Evolvability

Evolvability is the last bonus non-functional property, which describes how well the system adapts to new requirements to the software. In order to make the requirement more relevant based on the context of the product, the requirement was narrowed down to if the system supports badminton clubs with different number of courts and different game types for each court. In order to fufill this requirement, the matchmaking algorithm is designed to be agnostic to the number of courts and their types. In addition, the information about the badminton club is stored in a configuration file, decoupled from the application code. This allows easy modification by people without needing to understand the system's components and code.

4 System design

4.1 Player and Administrator Frontend

4.2 API Gateway

The API Gateway is written in Node.js using the Koa framework. Node.js was chosen because the purpose of API Gateway is not to do complex calculations, but rather to aggregate and/or proxy calls to the internal micro-services. The benefit of Koa is that performing asynchronous call chains is made relatively simple (for HTTP call aggregation). Also, since the heavy computations are done in the micro-services, then the lower performance of Node.js when compared to a compiled language is insignificant, and still suitable for the application.

4.3 Micro-services

The announcement, user, game, and session services are all written in Scala using the Scalatra framework. Looking into the codebase for the aforementioned services, there will be files with recurring patterns in their naming:

- xServlet.scala
- xDAO.scala
- x.scala

where x approximately refers to the name of the service or model of interest. For example, in the user service, the following files will be present:

- _UserServlet.scala
- UsersDAO.scala
- Users.scala

-technologies used

4.4 MatchMaker

The MatchMaker component is a part of Game Service, which is written in Scala. Scala was chosen because it allows algorithms to be written in a mathematically concise style. In the matchmaking algorithm, a list of checked in players and the court configuration is provided as input, and the output is a JSON representation of the players that will be playing on each court as well as the waiting list. The data structures used in this algorithm is Scala's Mutable.Queue to represent the waiting list, which will be denoted as *queue*, as well as Scala's List to store a list of matches, which will be denoted as *matches*. First, the algorithm calculates the amount of players that need to be placed on the courts to fill up the available court space. This can be calculated via the configuration file: $numDoublesCourts * doublesSize + numSinglesCourts * singlesSize$. Next, the calculated amount of players is popped off from the beginning of the queue, grouped by their levels, and scrambled within their groupings, which will be denoted as *selection*. A random index of *selection* is chosen, and starting from the two outer edges of *selection*, the algorithm iterates through each element towards the chosen index, and assigns court groupings in that order. First, the doubles courts are filled up, and then, the singles courts are filled up. In the first iteration of the algorithm, this portion was done from left to right, but it was observed that there was a bias for a certain level of player to be placed into the singles courts. This was adjusted for by including the random index mentioned above. Therefore, the output of the algorithm would be matches with players that are in similar level ranges. An optimization for this in the future would be to include player preference into the algorithm, since that is currently unaccounted for.

4.5 Gandalf

-talk about HMAC, JWT etc. here

4.6 Pigeon

Pigeon Server is written in Node.js with Express and Socket.io. The choice in language and frameworks is because ShuttleQL required the ability to send push notifications to the front end clients. The only methods to achieve this are by long polling or websockets. The websocket approach was chosen as it has wide browser support, and prevents the need of constantly closing and re-opening connections. Consequently, Socket.io was chosen since it is one of the most supported libraries for websocket integration, and has good support for Node with Express. Pigeon Server is notified by Amazon SNS when actions occur in the micro-services. Amazon SNS was used because it provides an abstraction as well as a security layer for notifying Pigeon of events that happen. Without Amazon SNS, a custom security layer would have to be written for Pigeon.

4.7 System Analysis

4.7.1 Coupling

4.7.2 Adaptability to Future Requirements

5 Participation Journal

Let C, D, J, T represent that Clement Hoang, David (Cheng) Dong, Jason (Zhaotian) Fang, or Tony (Di Sen) Lu worked on a particular component/class.

Below is a table that maps each component/class to a list of 1 or more team members that worked on the component/class:

Component	Sub Component	Team Members	Total hours worked
Admin Panel		C,J	75
Player Dashboard		D,J	50
API Gateway	API	C,D,J,T	10
	JWT	D	15
Gandalf		D	6
User service		C	20
User database		C	4
Session service		T	17
Session database		T	3
Game service		J,T	7
Game database		J	3
Announcement service		J	3
Announcement database		J	1
Matchmaking		C,J	20
Pigeon		C	12

In the table below, the total number of hours worked per team member is shown across the entire system.

Member	Total hours worked
Clement Hoang	86
David (Cheng) Dong	48.5
Jason (Zhaotian) Fang	89
Tony (Di Sen) Lu	22.5