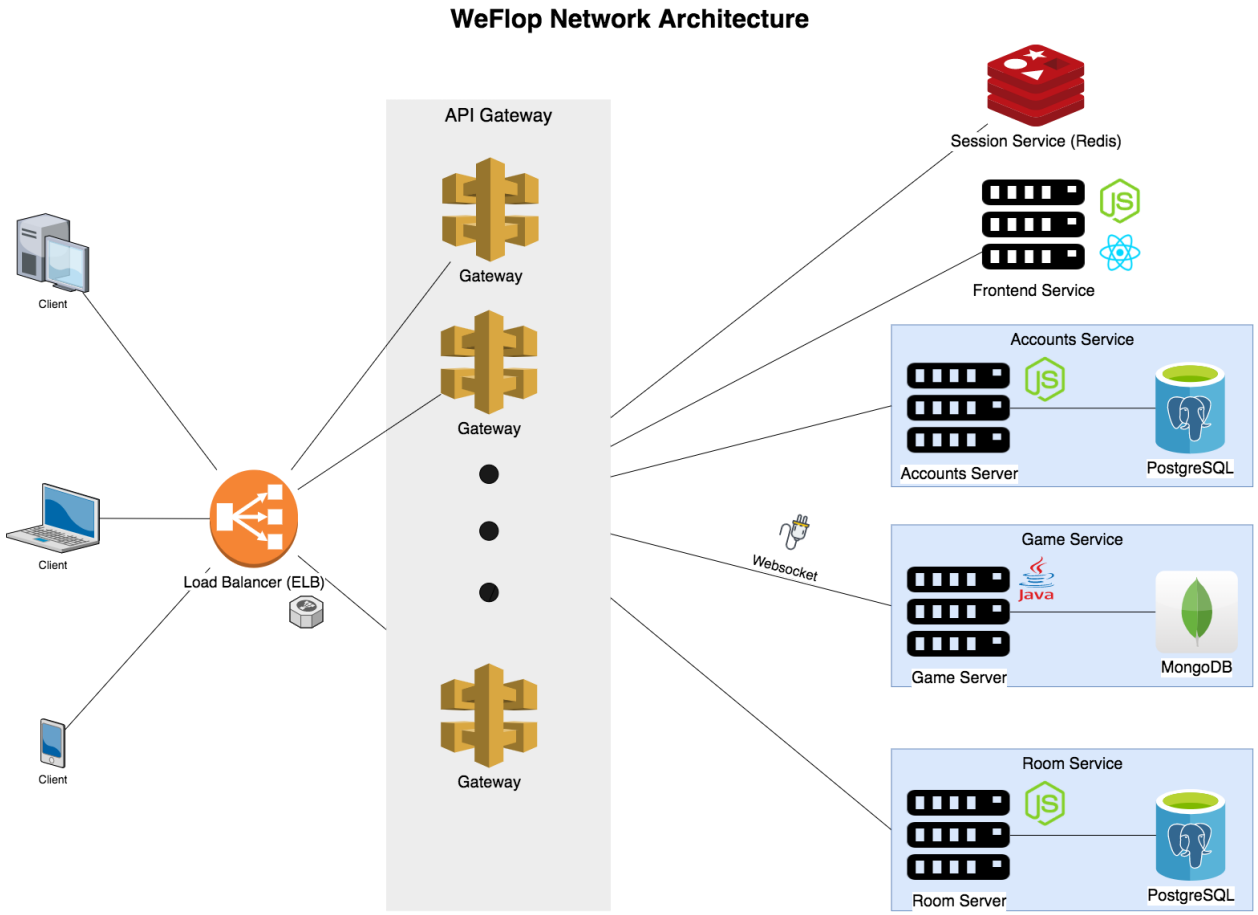
WeFlop: System Architecture

Figure 1.0:



The above figure provides a general outline of our system architecture. As you can see, the proposed architecture is based on microservices that manage Sessions, Frontend, Accounts, Live Games, and Rooms. We will break each of these services out later in the document.

Microservices

Wouldn't it be easier to develop the application as a monolith? Instead of splitting our application up into services, why not just have one server handle everything? This is actually a very common pattern and would be easier in the short term. In fact, in the early 2000s, this would be standard practice (even at scale). However, more and more

companies are switching over to microservice-based architectures for a few reasons. Firstly, when you start to work with teams of developers, having the bulk of your application's code all on one server makes it hard to collaborate. Microservices allow you to give each team their own service, allowing for faster iteration. It also allows you to choose tools and stacks best suited for each service. We may have some data analytics tools that are best suited to be written in python, but still want to write our Game server logic in Java. Breaking up our application into services allows for increased flexibility at scale. These advantages are why Netflix, Amazon, Lyft, and countless other large companies have chosen to use architectures driven by microservices.

However, microservices come with their own set of problems. It requires coordination between services. Some services may be reliant on other services, necessitating the development of communication protocols between services. Furthermore, it begs the question, should clients be able to communicate with each service directly?

In our case, we have elected not to allow that for security reasons (you also make it harder to aggregate requests results if you allow this sort of direct communication) . We instead elect to use an API gateway to manage communication between services and external clients. We go into more detail about our gateway later. Another potential issue is latency. Since we have to make requests to other services over HTTP or some messaging broker, we incur additional network latency. However, it is worth noting that this latency (if managed correctly) should be insignificant compared to database latency. Thus, it should not be a driver for latency in our application.

Now that we have covered our decision to use microservices, refer back to Figure 1.0. We break each portion of this figure down in further depth:

Load Balancer

All clients direct traffic to some form of a DNS load balancer (if we choose to host on AWS public cloud, this will all be handled by Elastic Load Balancer). This is something we will not have to worry about until deployment and will not have a large impact on development.

API Gateway

An API gateway is just a server that proxies traffic to our services and load balances these requests. We do not want clients to be able to directly request our services, as

this presents additional security concerns. An API gateway acts as the middle-man between our services and clients. Technically, gateways are responsible for a number of other features such as throttling, rate limiting, request aggregating (some requests may ask for a number of items spanning multiple services and these features need to be aggregated into one response), throttling, token validation, logging, metrics collection, and more. A lot of these features will not be important until we begin to scale, but choosing an API Gateway that allows us to eventually integrate many of these features is important.

For our MVP, the API gateway will probably be one of the last things I implement. It isn't necessary to develop any of our individual services. Thus, choosing a stack for this gateway isn't imperative to do know. There are three choices of technologies upon which I could build my gateway that I am considering:

1) NGINX: NGINX by itself is just a reverse-proxy and not an API Gateway. However, a lot of the functionality important for an API Gateway may not actually be necessary for our MVP, so NGINX may be sufficient to proxy our traffic to our services. If we wanted to later convert to an API Gateway built on top of NGINX, we could transition to using Kong.
2) Express Gateway: A lot of our proposed services our built on top of Node.js, so it will be convenient to build an API gateway using Node.js. It is straightforward to use. However, it isn't as heavy duty or commonly used at scale as some other options like Kong.
3) Ambassador Edge Stack: An additional layer that integrates with Kubernetes to provide some heavier-duty API gateway features. Much more complicated to use than Express Gateway, as it requires integration with a lot of other tools and has poor documentation.

Session Service

Our session service is just a distributed cache that allows us to quickly lookup session tokens and see which users those sessions correspond to. For services like these, there are really only two choices: Memcached and Redis. Memcached is a bit older and is used by Facebook as well as a number of other large companies. Think of it like a very efficient distributed key-value store. Redis can provide identical functionality as well as a lot of other things. The richness of objects you can store in your cache is much greater than with memcached (that can really only store string values). Furthermore, its performance is as good if not better than memcached in most areas.

Besides just storing session information, we may also end up caching request responses and game information as well (such as which server replica a given game is being hosted on–more on this later).

Frontend Service

There isn't too much to talk about here. React is a very popular framework for designing dynamic frontends, and I personally like its component-based design. Its use of components opens the possibility of breaking up our frontend into many different frontend services if we ever needed to. Choosing Node.js as the backend for this service makes sense because it is easy to use and will not require frontend developers to learn an additional language (as JS will already be heavily utilized on the frontend). Node.js also performs extremely well with simple I/O tasks, making it a good choice for performance reasons.

Accounts Service

This service handles account creation and login. User account information is very structured, so a relational database management system seems like a good option. PostgreSQL scales extremely well and was a clear choice for this. It actually outperforms MongoDB for many tasks, especially transactions. Its performance with transactions is a big reason why I did not just choose to use MongoDB for all of my databases in all of my services. Eventually, we may choose to create a Transactions service to facilitate transactions between users. PostgreSQL is ACID compliant and performs MUCH better than MongoDB with respect to transactions (around 10-100 times better in fact). It would be a clear choice for such a service. If we were not going to eventually introduce features relying on transactions, I likely would have just opted to use MongoDB for every service for simplicity. However, since we will likely introduce such a service and use PostgreSQL for this service, I decided to consider it for my other services as well.

Game Service

This service handles all game logic and game connections. The state of a poker game will be stored on a specific server. Users connecting to a specific game will connect via websockets to the server hosting that game. Each game will have a unique id, and the address of the server hosting a given game will be stored in our Redis cache. Because of this, unlike our other services, the servers in our Game Service are not inherently stateless. As we scale, we will need multiple replicas of game servers to handle many

different concurrent games and websocket connections. However, each game is only hosted on one server. Once we scale to multiple game servers, we can handle potential server failures by periodically flushing game states to our database. Thus, if a server fails, we can propose a new server to host the game and restore it based on our last save. We can also looking into caching game states in a distributed cache.

Choosing MongoDB as the database for this service makes sense in this case as game states may end up being quite unstructured and a JSON representation is the most flexible representation to work with.

Unlike our other servers, this server will be written in Java (using the Spring Framework). I elected to use Java over Node.js because I felt that Java's statically-typed and object-oriented features make it good at representing potentially complex game states with multiple objects such as Cards, Hands, Players, etc… Furthermore, Java is multithreaded at its core (unlike Node.js which is single threaded), making it easier to scale vertically than Node.js.

Note that this server will also be responsible for providing information about game state through REST API calls along with websockets (useful for providing previews of games).

Room Service

The Room Service will handle requests asking about which games/rooms a current user is in. For our MVP, this service won't do a whole lot other than CRUD operations on its database keeping track of user-game information. Since the core functionality of this service is to retrieve relational information about users to games, a relational database made sense, so we chose PostgreSQL as the database for this service.

Future Services

As I have already mentioned, in the future we may consider adding a Transactions service. We also may consider adding a Metrics service that collects metrics on user/game information and can provide analytics. Java (Spring) + MongoDB would be my recommendation for the Metrics service stack. We also can look into adding services that serve group video chat and other features of that nature as we integrate them. If we expand our team, due to this microservice infrastructure, it will be easy to pass off these services to very small teams that can develop and test them in isolation before integrating with our platform using whatever tech stack they deem appropriate.

Production-Related Technical Details

For development, we do not need to create replicas of our API Gateway or replicas of any server or database for that matter. In real life, as we scale, we do want it to be easy to create these replicas and manage them, however. The above architecture as I have proposed it accomplishes that. Most servers are stateless so all we need to do is load-balance requests. In the case of game servers, we can still load balance initial requests to create games and distribute our requests that way. Our caching mechanism will allow us to handle additional replicas.

Once all of the parts of our MVP are built, we will want to glue them together in a way that makes scaling easy. My suggestion is to put all of our services and our API Gateway inside of Docker containers. We can use Kubernetes as a Container Orchestration system to automate-away a lot of the scaling issues we may have when introducing replicas. We can also use Istio to connect and manage our service connections (a common use case is to manage retries on failed service responses), acting as the “glue” between our microservices.

Conclusion

The presented architecture is a microservice-based architecture. All requests are routed and load-balanced through an API Gateway. We have five core services: one that caches (primarily sessions), one that handles authentication and user accounts, one that serves live-poker gameplay, and one that handles poker rooms. Future services may involve transaction handling, analytics, and live-chat functionalities.