Microservices for Casual Turn-Based Mobile Games

Comp 680 - Advanced Topics in Software Engineering - Prof. Boctor
May 08, 2019

Edgar Lopez-Garcia · Ashot Chobanyan

Abstract

This paper outlines the implementation of a microservices based system for providing services to asynchronous turn-based mobile games. It goes over a high level Reference Architecture (RA) that describes how such a multiplayer system can be configured on the back-end. The reference architecture includes a high-level system design that adapts to a three-tier architecture system. It then goes into the common back-end components required to facilitate such multiplayer system. Finally, a deployment diagram is introduced that shows the physical components of the system. The paper continues with the description of two Implementation Guides (IG). The first IG covers a serverless approach using the Google Cloud Platform. The second IG covers a more standard microservice driven approach utilizing AWS. Using either guide you should be able to setup a scaling set of services to provide for your game with little unneeded overhead. The paper concludes with remarks towards problems encountered and how they were resolved. Improvements on the design and how the system can be extended are also discussed.

Table of Contents

Introduction and Background

Statement of Problem Area

Increasingly many people are approaching mobile gaming as a method to entertain themselves, and most of those games involve various online elements. In order to properly fulfil the needs of large playerbases that often spike and bottom out throughout the day traditional dedicated server approaches become wasteful for many types of games. This is especially true for turn based games where the number of player actions may be relatively few over a long duration of player engagement. In the past, these types of game services have been implemented by developers in which they take charge of the many aspects required such as server provisioning, deployment, and support. With recent advancements in cloud computing, many services have been created to aid the developer in making web applications without having to worry about the logistics of setting these systems up. By using these systems, the developer can focus on application logic rather than back-end system configuration. Asynchronous turn-based multiplayer mobile games closely resemble many aspects of typical web applications. Thus, it is worth investigating how these same cloud services can be leveraged to easily and quickly create these types of multiplayer games with minimal maintenance on the back-end.

Previous and Current Work, Methods and Procedures

Traditionally games would often employ monolithic servers in order to provide services for their users. This would involve spinning up a dedicated server that would accept API calls from the playerbase who are using applications to interface with said server. Seeing as all the static content needed for players is often delivered through application stores, be it iOS or Android, there is an innate reduction in overhead for servers used by games. However, this method is still inefficient.

Many Unity games projects for instance will often rely upon the built-in UNet or Photon multiplayer infrastructures. However, both carry costs that are specifically detrimental to mobile games. Both of those platforms have concurrent user limits on connections and services which increase prices based on peak usage. This is fine for traditional games that have a higher barrier of entry such as high upfront costs and hardware requirements. In the case of mobile games one must provide services to a greater number of players due to the low barrier of entry and low expected return per user. In such an event a technique that has no limit on concurrent users is needed.

There exist methods to support peer-to-peer connections for individual game matches between players as well, but these methods often rely upon the previously mentioned systems such as UNet and Proton to establish the initial connection, and suffer from poor validation of player behavior, allowing cheating to become much easier. Peer-to-peer games also limit the scope of games and are typical only for a small number of players within a match.

Background

A broader industry trend in the server space has been the migration to microservice based architectures to provide high accessibility and reduce server overhead at all levels. We believe that microservices lend themselves especially well to the types of games which do not require a constant connection between users, chiefly among those are turn based games. In the case of turn based games players only provide valuable input into the server periodically, which should require very little bandwidth and not need require a constant connection at any stage.

The most extreme method of microservice implementations involve serverless functions. Doing so reduces the need for servers to the bare minimum and allows your business logic to be run closer to cost. In addition, this should theoretically work to maximize uptime which should help to maintain user engagement. But in using serverless functions there is a need for the player update logic to be stateless, which limits their use in gameplay logic for many types of games since a call needs to be sent out for each player in response to the current game state.

Brief Project Description

Our project implements a microservice based networking infrastructure within a turn-based mobile game project. More specifically it implements a serverless methodology for microservices. Using this methodology minimizes the utilization of servers for each user while fully maintaining expected game functionality. However these patterns do not apply to games that have constant small changes to game state in where a constant stateful connection is needed. These patterns are both well suited to games running in any environment however are designed with the intention to be used by non web-native systems/engines.

Reference Architecture

Three-Tier Architecture

The goal is to design a back-end system to support asynchronous turn-based capabilities. Although not immediately obvious, this system can be modeled with a three-tier architecture design at a high-level. Surprisingly, this system mimics many asynchronous applications that have had success using the three-tier design. Since real-time components are required, the multiplayer system can be treated as a standard asynchronous web application. We propose three layers: Presentation, Business, and Persistence. The client of the multiplayer game is meant to be ran on native mobile devices. As a result this system cannot necessarily leverage the typical web technologies for the view. Therefore the Presentation layer is responsible for preparing and dispatching required data as JSON to the client for rendering. The client should have pre-defined rules to interpret and render the data once it is received. The Business layer includes all services that support multiplayer functionality such as matchmaking, gameplay, and notifications. Finally, the Persistence layer saves data pertaining to lobbies, gameplay state, and player profiles. Since the Presentation layer is only invoked when business logic is required unlike regular web pages, authentication is required on each request. Thus an authentication service is required before each request to validate the user. Refer to figure 1 for a visual depiction of the three-tier architecture pattern proposed as part of the reference architecture.

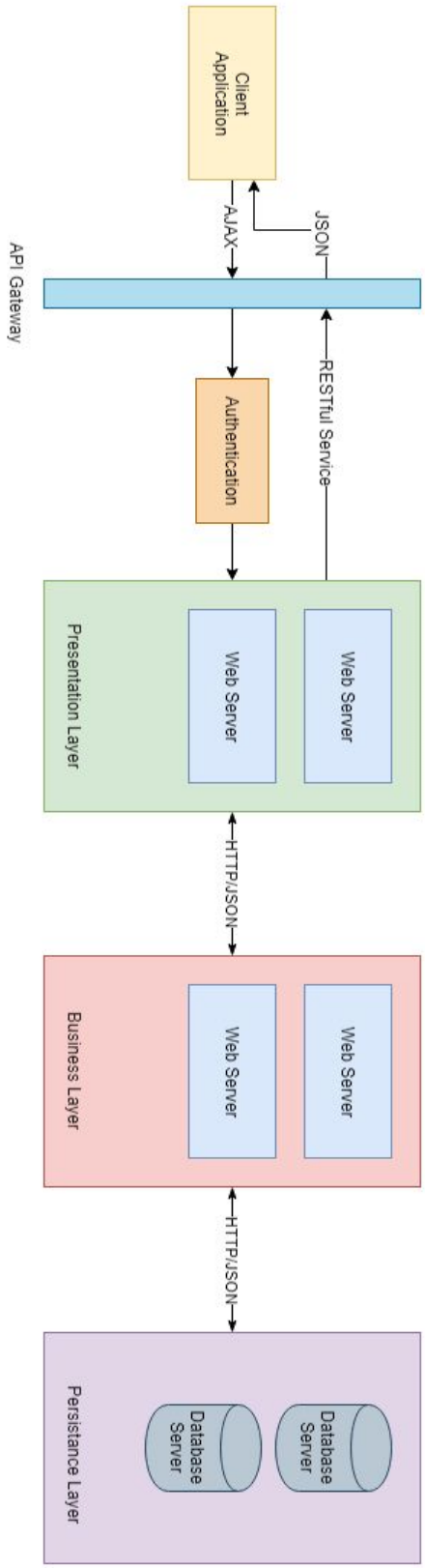

Figure 1: Three-tier architecture pattern

Domain-Specific Three-Tier Components Diagram

Each layer of the three-tier architecture has a specific role. At the presentation layer you will find components that specialize in preparing JSON data to send to the client. At the very minimum, the multiplayer game has three views in some shape or form: the games view, the gameplay view, and the notification view. As such, the presentation layer should contain these three services at minimum. The Games View Data component prepares JSON data that contains information about the games the player is associated with. For example, it contains all the games and lobbies the player is currently subscribed to. Details about the current player's turn is also included. The Gameplay View Data component prepares data for the rendering of the actual gameplay represented by it. In other words it prepares the data required for the client to render the game on the device. This data should be a copy or derivative of the gameplay state, allowing the server to withhold information only accessible to each player. The Notification View Data component prepares data for the notification messages the client(s) receive. This service is mostly used to inform the client if it is their turn for a specific game.

The business layer is in charge of data manipulation within the application. Specifically, it deals with anything gameplay related and logistical tasks required to have a user find a game and complete the game. The Matchmaking service is in charge of connecting a player to an open game lobby. If no such lobby exists, a lobby is created for the player. Once a lobby is full, the game should start. The Matchmaking service communicates with the Lobbies Database in order to find or create lobbies. The Gameplay service is in charge of moving forward a game. It manipulates the Games Database based on the action requested by the client and modifies it appropriately. It also handles examining for win conditions upon each turn. The gameplay state is always stored within the Games Database. The Game Creation service creates a game once a lobby is full. It accesses the Games Database, which stores all the games, in order to add a game to the database in response to user requests. The Notifications service creates messages to notify the client of something. Most likely, this will include messages notifying the client it is their turn for a particular game. Figure 2 summarizes the layers and components within each layer in the three-tier architecture.

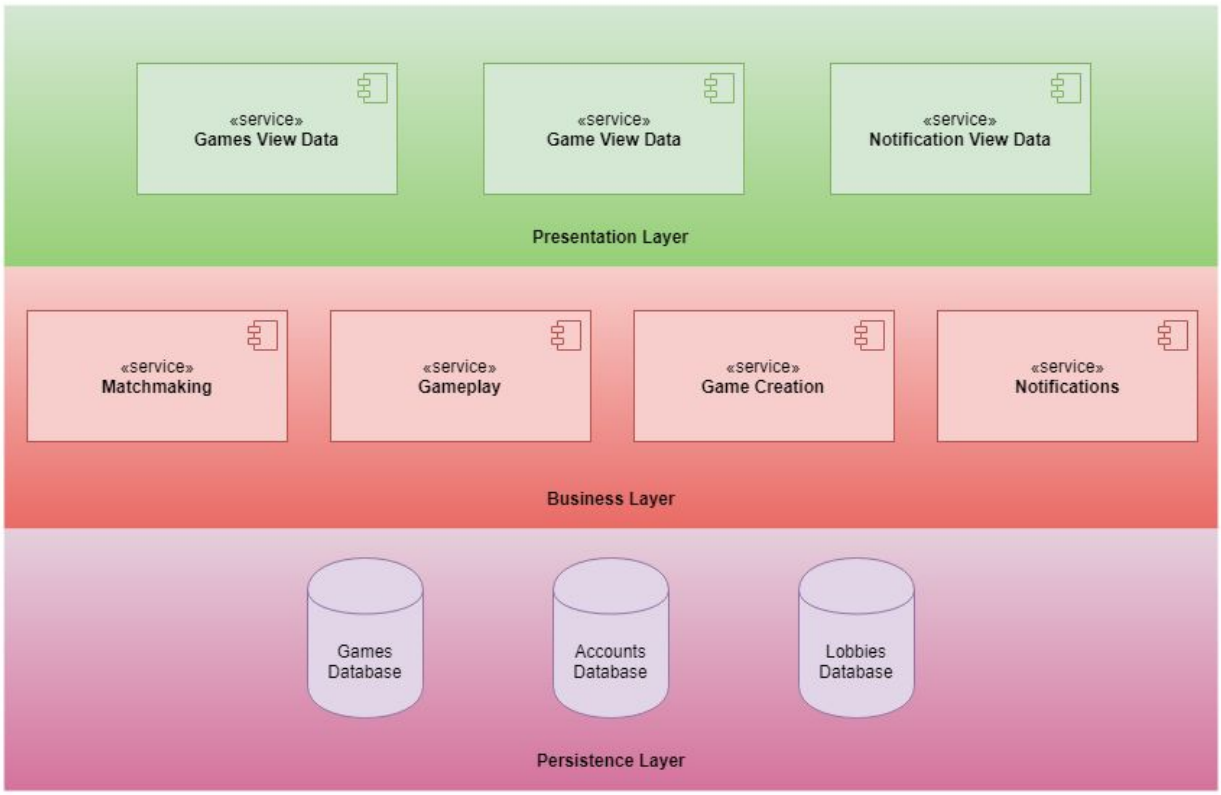

Figure 2: Domain-specific three-tier components

Deployment Diagram

The deployment diagram indicates the physical layout of the system and its components required to run the system on a network. The deployment diagram follows the three-tier architecture with inter-component communications. In shows the physical layout of the system and also how components should communicate between one another. Although the deployment diagram shows the physical layout of the system, it is still open for interpretation during the implementation phase. That is, it does not explicitly state how this asynchronous turn-based multiplayer system should be configured using actual services offered. The system can be deployed in a managed environment commonly seen in cloud providers, but is not restricted to this; you can deploy your own physical hardware and manage your own servers that implement this deployment guide. Whereas the domain-specific three-tier diagram shows the components required for a asynchronous turn-based multiplayer system, the deployment diagram shows the physical layout of these components in relation to one another and their inter-communications. Refer to figure 3 for the deployment diagram.

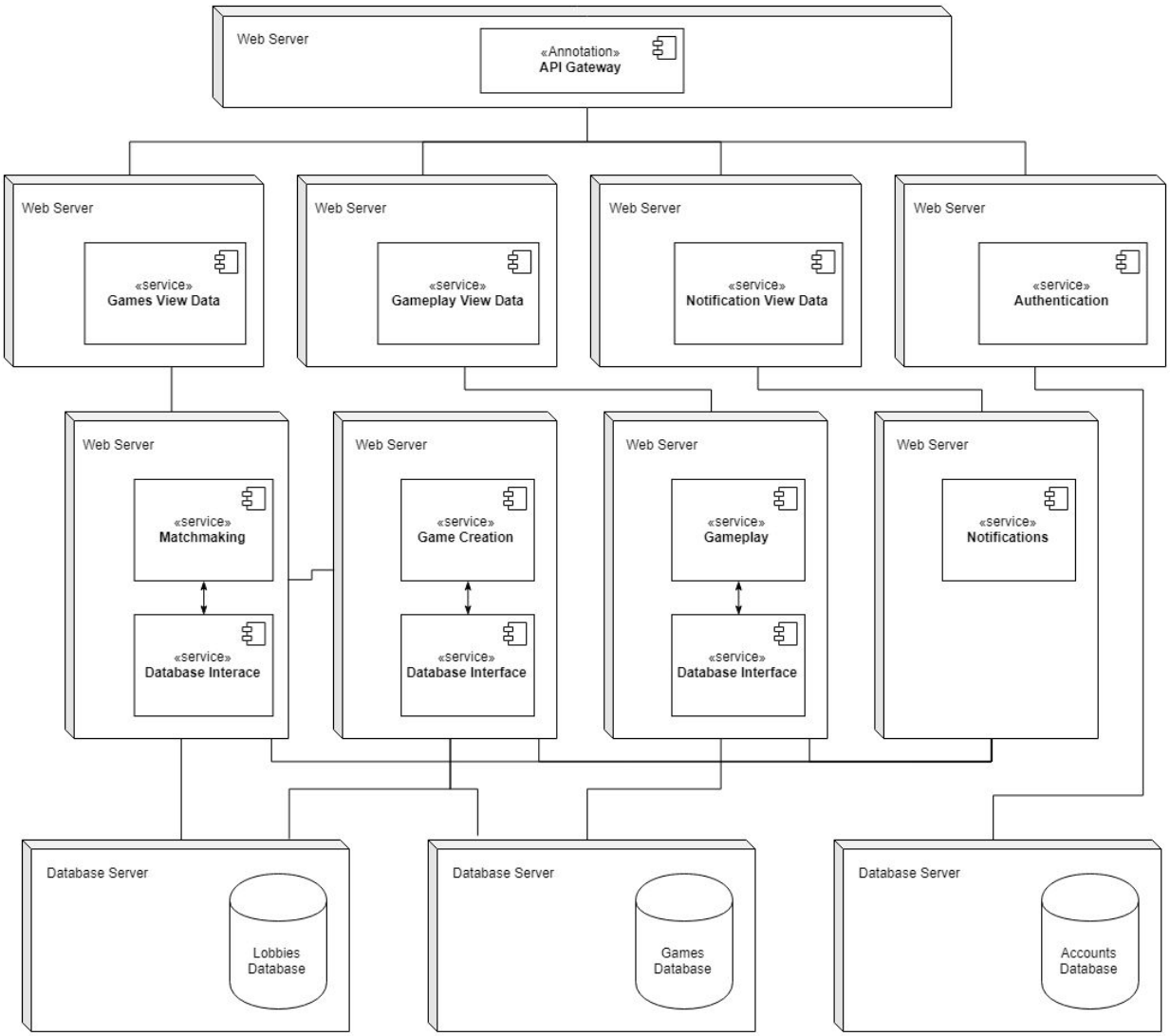

Figure 3: Deployment diagram

Implementation Guides

Implementation Guide 1

The first implementation revolves around using Google Firebase services. Google Firebase services expedite web application development by providing out-of-the-box services for many common problems required in web application development. For instance, most if not all web applications required authentication, static asset storage, database storage, and web servers the business logic. Firebase provides ready-to-use solutions for all of these problems. It is wise to leverage as many as these products as possible to get a web application running as quickly as possible. The user-friendliness and price points (free under the Spark plan) of Firebase make it a very attractive suite of products for developing. Thus, it is worth examining how a asynchronous turn-based multiplayer game can be implemented in Firebase. Figure 1 depicts the implementation guide with Firebase.

Although not explicit, the implementation does follow the three-tier system proposed in the Reference Architecture system. The View layer is implemented using Firebase Functions. Since these Functions are ad-hoc and short-lived they do not require maintenance and can be scaled horizontally to the demands of traffic. These View components make HTTP requests to the Business layer components where they perform the operations described in the Reference Architecture section. They are also implemented with Firebase Functions for the same reasons listed above. Furthermore, all the Firebase Functions are implemented using NodeJS since it provides a direct translation from JSON to JavaScript object and vice-versa, allowing for quicker development. Since any game or game-related operation is purely data-driven, NodeJS is good enough for most cases to manage data flow. One caveat to Firebase Functions is that they have a lifespan limit of 60 seconds. Since most of the operations required for casual turn-based games (ex. tic-tac-toe) are not resource intensive, this limit should suffice to run gameplay logic. The lobbies, games, and accounts are stored using NoSQL Firestore. The Firestore interfaces well with Firebase Functions, allowing the developer to trigger a function when a record is changed in the Firestore database. The Business layer components interact directly with these databases to manipulate the data. Since Firestore is a managed database service, it scales with demand so you do not have to worry about provisioning additional databases; the replication is done for you.

Firebase does not provide an api gateway. Although there are many managed api gateway services to choose from, Google Cloud Endpoints is chosen so that it easily

interfaces with Firebase services. The device performs an AJAX request indication the user wants to take an action, whether it be joining a game or making a move. The Cloud Endpoints service consumes this request along with the access token sent to it. The access token is required to authenticate the user for a particular action. The reason for this is to make sure the user making the request is requesting data only from their own sessions. The actual user validation occurs in the Authentication component implemented in a Firebase Function that gets invoked before any other Firebase Functions. It is part authenticator and part router since it has to delegate the request to the proper service once validation is successful. The Authentication component is different from the Firebase Authentication service. The Authentication component authenticates requests from the client while the Firebase Authentication services signs-in a user. Since Firebase Authentication is a managed service, a user can sign-in using multiple SSO providers such as Google, Facebook, and even email. If a user signs-in for the very first time, an account is created for that user by the Account Creating Firebase Function. Firebase Functions can be triggered via Firebase Authentication services. Thus the Account Creating component leverages this functionality by creating an account when a user signs-in for the first time.

Finally, the Notification Function interfaces with all of the Business layers components to notify the client about changes occurring within the system. Although AJAX requests are expected to provide responses to the client, there are some situations where information must be provided outside of AJAX calls. For example, one player may take days to respond to an action made by another player. AJAX calls cannot wait for this long for a response. The Notification component alleviates this issue; it sends information to the client whenever the data is ready whether is days or weeks later. In this implementation, the Notification Function interfaces with other components to prepare the data. Once the data is ready, it is sent to Firebase Cloud Messaging, which interfaces directly to the client's mobile phone. By supporting these features, the player will never miss out on important events occurring within the games they are in.

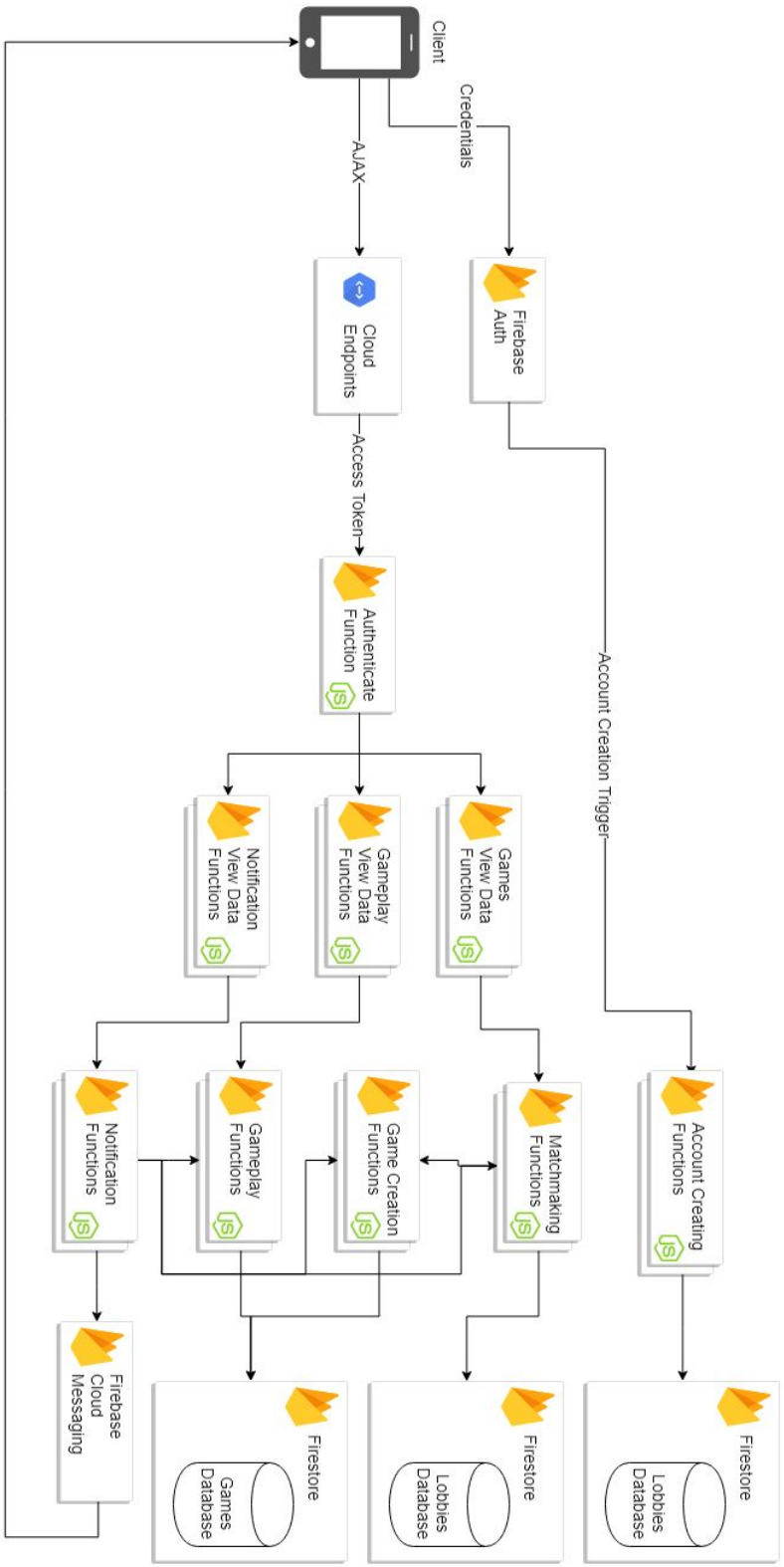

Figure 4: Firebase Implementation Guide (IG 1)

Implementation Guide 2

Another implementation exists within AWS that adheres to a more standard non-serverless microservice approach. This approach assumes you will be delivering your game on an app store such as the Play Store which curtails the need for a separate binary data store. Most users prefer to download games from these trusted sources so the use of an external data store for your application data is both unnecessary and punished.

On the Presentation layer you need to leverage a REST based API to communicate out to your servers and render the data that is sent back. Since our game relies upon a stateless architecture the REST based API alone is enough to power all the services needed by this layer.

On the Business layer you need to establish an EC2 instance to respond to the API requests from your Presentation layer. You should set up an Elastic Load Balancer to autoscale services based upon inbound traffic, the load balancer that works best with EC2 instances for our use cases would be the Application Load Balancer. You should configure your load balancing to be based upon CPU utilization, the relatively light but frequent workloads produced by turn based games do not require any more complicated metrics to act as triggers for auto-scaling operations. Use Puppet to install your application code upon the startup of a new instance. On this layer all the previously mentioned services (figure 4) should be created individually and ideally set to scale separately within availability zones and regions as necessary for your game.

If you desire for users to be able to notify one another of game state changes based on the actions of their opponents, trigger Amazon SNS as part of whichever services require attention on behalf of another player. SNS will automatically deliver your notifications across all major mobile platforms so there is no need to provide any device specific logic.

For authorization and authentication, using Cognito is the most straightforward choice. Your client app will directly contact Cognito service before reporting back to the rest of your app with its credentials assigned, ensure that no microservice is allowed to act on behalf of unvalidated or actions by the incorrect users.

On the Persistence layer any reasonably quick choice of database would suffice however MySQL would be the most straightforward in the context of storing game data. The multiple databases mentioned in the previous implementation guide have been compressed down to a single database for the sake of brevity in the below diagram.

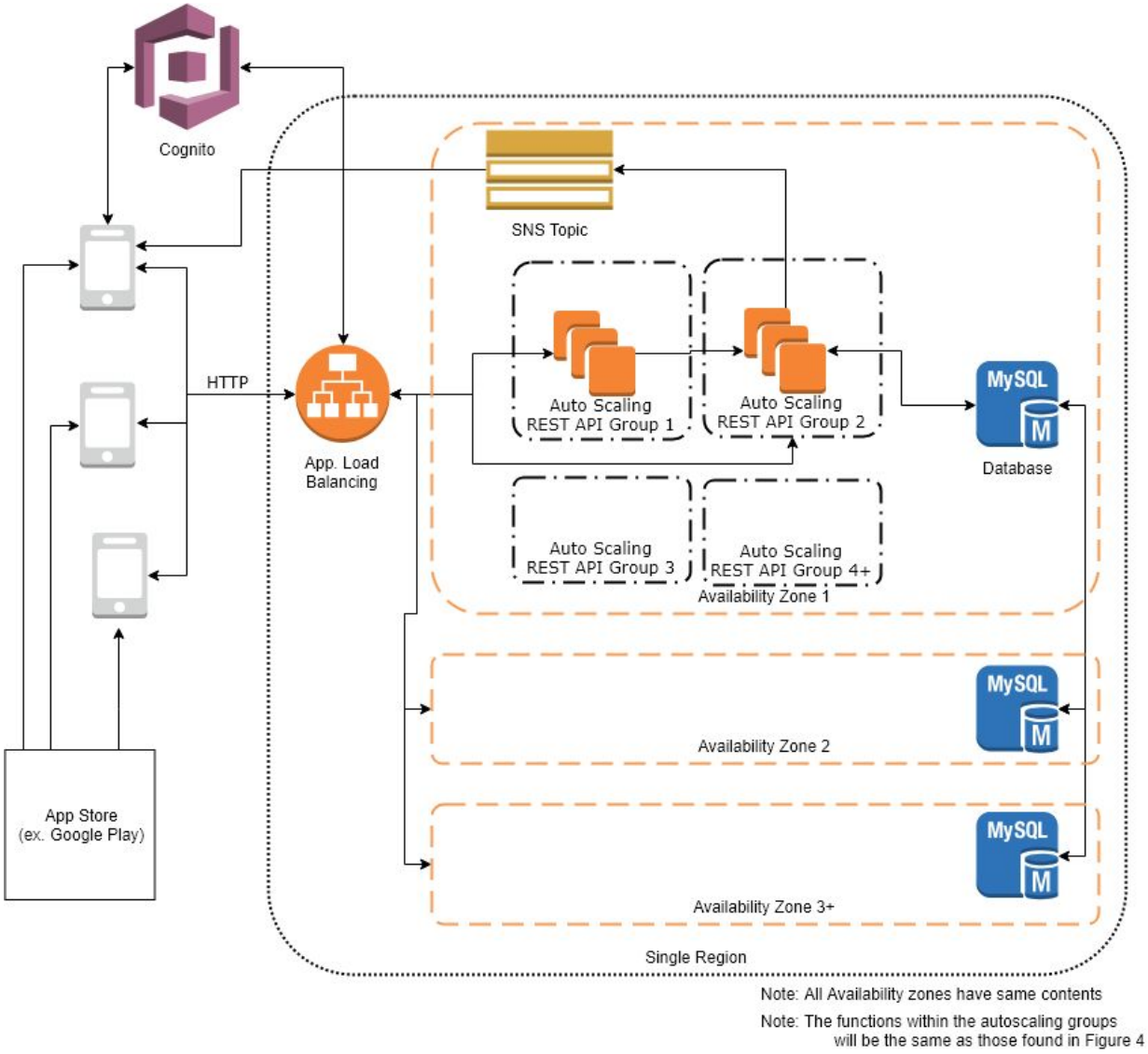

Figure 5: AWS Implementation Guide (IG 2)

Conclusion

Summary

The use of microservice backend services for mobile games is a natural fit and will drive your server costs down substantially while being able to sustain many concurrent users. Microservices perfectly suit the rapid demands of mobile games and will ensure that sufficient resources are available throughout the day to rapidly meet user demand. The use of a serverless approach is highly recommended however a standard microservice based approach is also good.

Problems Encountered and Solved

Firebase offers two types of database services: Real-Time Database and Firestore. The differences between these two services is not immediately apparent when first deciding which to select as your database service. It was not until after implementing the database functionality that we saw the fundamental difference. Firestore provides a typical noSQL database solution. Meanwhile, Real-Time Database provides the same features are Firestore with the addition of a real-time component. This means that with Real-Time Database, you can automatically sync data to the client without having to do any additional REST calls. This can speed up the development process and makes data syncing a fool-proof functionality with offline syncing support.

We left Firestore as the default database service to use in implementation guide 1 because it does not assume anything about your environment. Real-Time Database will only work if you have the proper SDK setup on the client. This might not always be the case, especially if the SDK is not supported in your platform. As a result, Firestore is a saver bet to use since it can be invoked with Firebase Functions. Although Real-Time Database can simplify data retrieval for the developer, its pros might outway its cons in some situations.

Suggestions for Better Approaches to Problem/Project

Our approach takes rolling out your own backend solution a step further by using cloud service providers to manage these aspects for us. In return, we can focus on developing core functionality of the game without having to worry about provisioning machines and configurations. Yet, many common problems within gaming platforms still need to solved by the developer. For instance, matchmaking is a common functionality seen across any multiplayer game. Our approach suggests you implement your own matchmaking service. Although the matchmaking service is managed by the cloud service provider, the developer still has to implement the explicit logic to match players together. This only gets even more complicated when you start adding special rules to the matchmaking such as player lever and win/lose streak considerations. This can be a daunting task and require significant time to implement such a common problem seen throughout multiplayer games. We acknowledge the shortcomings of our design for these such aspects and recommoned the reader to take managed services a step further and use a cloud service platform specifically geared towards multiplayer games. Google offers a cloud-based open source matchmaking service called Open Match. Open Match provides managed matchmaking for your game. Additionally, Google' Agonese platform provides specialized cloud solutions for deploying and scaling dedicated game servers. These specialized platforms solve common problems faced in multiplayer game development that can save development time and energy. We recommend the reader to look into these services and consider using them if their needs align with what these services offer for a better development experience.

The second implementation guide offers up a system that is more flexible to development demands however it suffers in that it lacks the relative simplicity as found in the first implementation guide. Serverless is the ideal state for the turn-based style of gameplay to reduce its server footprint, but IG2 is more open to change in case one wishes to add more real-time elements to game at a potential future date.

Suggestions for Future Extensions

These implementation guides are heavily tailored to mobile turn based games but can just as easily apply for or between any platform where games can be played as the presentation layer only requires a device to be able to make API calls. The resources available across platforms (eg. Desktop games, game consoles, TV Apps) may vary so a set of common resources needs to be established and any vital services will need to avoid using non-common resources.

There may be some methods to further extend our implementation to other types of games with the introduction of some stateful layer which can maintain more complex interactions for certain types/portions of games. In doing so cost will substantially increase but use of the microservices elsewhere in the game would still be useful for reducing the overall costs regardless. Implementation guide 2 is more flexible to accepting these changes for the introduction of state as serverless functions are especially weak with handling stateful connections.

Another, perhaps trivial, extension could be to map out implementation guide 1 within the AWS set of serverless tools. While Firebase is perfectly suitable for the task and is in many ways faster to implement from the ground up, a wider pool of talent exists for AWS based services which is especially useful when considering relatively new advances such as serverless compute.

Amazon GameLift is a separate service designed by Amazon specifically for games, but it seems to be (even by their own admission) ideal for realtime games that have fixed runtimes. Further analysis may deem it useful for use in turn based games like what is proposed here however our initial research suggests this may not be the case.

References

1. https://microservices.io/
2. https://microservices.io/patterns/monolithic.html
3. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63
4. https://medium.freecodecamp.org/monolith-vs-microservices-which-architecture-is-right-for-your-team-bb840319d531https://medium.freecodecamp.org/monolith-vs-microservices-which-architecture-is-right-for-your-team-bb840319d531
5. http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html
6. https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59
7. https://ieeexplore.ieee.org/abstract/document/7030212
8. https://www.thorntech.com/2017/12/microservices-vs-monoliths-whats-right-architecture-software/
9. https://dzone.com/articles/5-steps-to-successfully-prepare-for-microservices
10. https://www.slideshare.net/TriNimbus/chris-munns-devops-amazon-microservices-2-pizza-teams-50-million-deploys-a-year
11. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/
12. https://blog.armory.io/how-microservices-help-continuous-delivery-in-the-sdlc-part-one/
13. https://www.devbridge.com/articles/a-6-point-plan-for-implementing-a-scalable-microservices-architecture/
14. https://aws.amazon.com/microservices/
15. https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec
16. https://www.apiacademy.co/lessons/2016/06/api-design-304-api-design-for-microservices
17. https://dzone.com/articles/patterns-for-microservices-sync-vs-async
18. https://cdelmas.github.io/2015/11/01/A-comparison-of-Microservices-Frameworks.html
19. https://d1.awsstatic.com/whitepapers/aws-scalable-gaming-patterns.pdf
20. https://unity3d.com/unity/features/multiplayer
21. https://www.photonengine.com/en/pun
22. https://cloud.google.com/blog/products/open-source/open-match-flexible-and-extensible-matchmaking-for-games

23.
https://cloud.google.com/blog/products/gcp/introducing-agones-open-source-multiplayer-dedicated-game-server-hosting-built-on-kubernetes
24. https://aws.amazon.com/gamelift/
25. https://docs.aws.amazon.com/cognito/