Cloud Software Development Techniques for hosting a scale on demand application (web and/or native) with realtime data

- Firebase/GCP is the preferred hosting platform for realtime data
- Document databases and scale on demand platforms usually require vendor lockin
- A “typical” web applications uses authentication, data storage, blob storage and site hosting amongst other application specific features like notifications, dynamic links and other native application specific features
- Storing all code, permissions and deployment details in one project helps simplify non application specific integration and deployment complexities vs multiple projects, repositories and deployments.

Realtime data (socket based) has been something avoided by both AWS and Azure - probably because of cost of active connections at scale. All three major cloud providers now offer realtime database solutions.

AWS vs **Azure** vs **GCP/Firebase**

AWS (Amazon Web Services)

The biggest and offers diverse and well used features for cloud application hosting. Where Firebase has offered better solutions so far is with realtime data, and a more streamlined application development framework for cloud hosting. https://aws.amazon.com/amplify/ is a container solution aimed at competing with Firebase - but the realtime databases and integrated authorization still require much more work. AWS is a solid, reliable and respectable choice for cloud hosting. Dynamo DB https://aws.amazon.com/dynamodb/ is the realtime data solution offered by AWS to compete with Firestore. The authentication integration is much more manual codewise for the various pieces vs Firebase - but Amplify has worked hard to make a “typical” web application much easier to develop.

Azure (Microsoft Azure)

Microsoft has made many iterations and integrated some terrific features over the years. It’s also a continuation of the well liked and heavily used Microsoft development platform. It’s got the scale on demand features you expect from a cloud provider, at the same time the nosql and realtime database options have been late in coming, and I’ve not had first hand experience with Azure since 2015. What I read and gather is that Cosmos DB (https://azure.microsoft.com/en-us/products/cosmos-db) is a solution for hosting realtime data (scale on demand) with a MongoDB option (that can be used for realtime updates). Microsoft has always been a first rate provider of development tools.

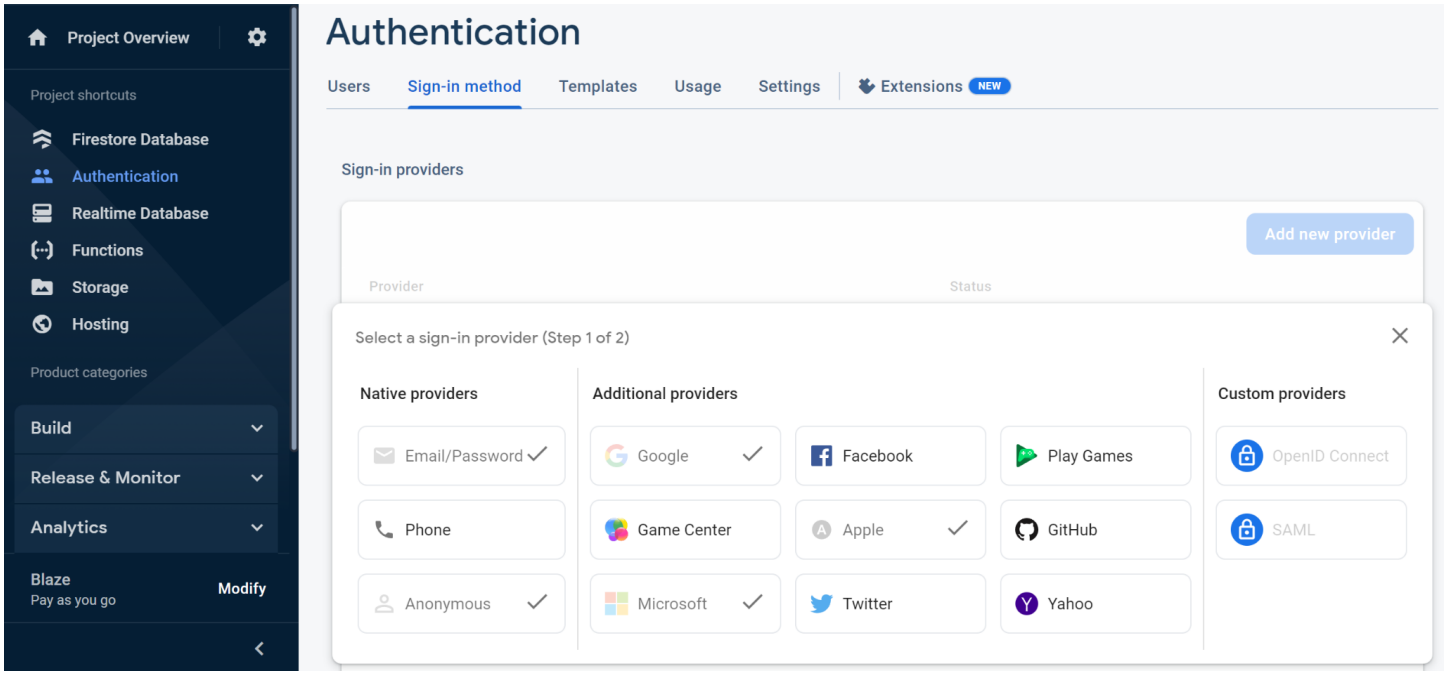
Firebase (Google Firebase)

Firebase was created as a chat service backend for realtime data, and was purchased by Google in 2014. Firebase was just RTDB and authentication at the point, Google integrated cloud functions, cloud storage and Firestore (Google Cloud Storage) in couple years after evolving Firebase into a very streamlined solution to use the overtly complex GCP hosting platform. While GCP is available, especially for integrations with Google’s cloud services, Firebase itself offers the integrated pieces for cloud application development and deployment. Firebase was selected as a focus for projects originally because of it being the only practical realtime data storage solution originally, but they’ve done spectacular by providing integration frameworks, super good documentation and examples and deployment solutions that get the job done without being to complicated. GCP is always there is you want some complication with integration.

Authentication Accounts

https://firebase.google.com/docs/auth

- Solution for both web and native applications
- Scales on demand for massive usage
- Email verification (optional)
- Tokens for passing to cloud functions (server code)
- Webhooks for account create/delete, etc
- Anonymous login (optional)
- Passwordless login with emailed link (works for native apps too)
- Support for most OAuth signin providers provided



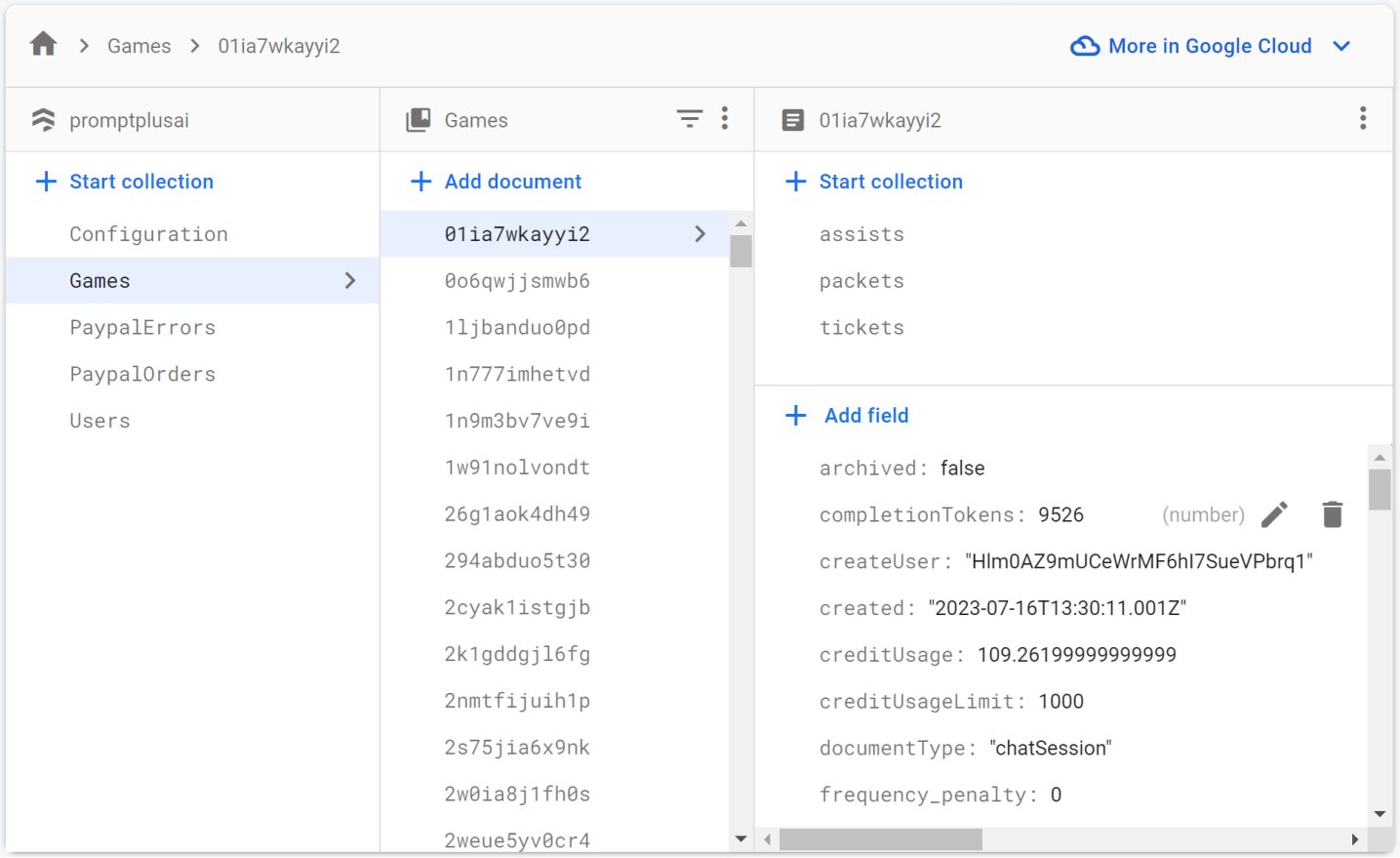
Realtime Data with Firestore or RTDB

https://firebase.google.com/docs/firestore
https://firebase.google.com/docs/database

RTDB has lower latency and is ok to use for games, Firestore has very good realtime performance, but during busy times of the day can have more latency than is useful for games but is fine for business purposes. Firestore offers more scalability, better query options and a document based design with subcollections that make it easier to get only the information you need. RTDB is a JSON database that requires some primitive techniques like storing data summaries to keep size reasonable for querying lists, etc. RTDB includes a special disconnect features that makes it an ideal solution for online presence (the green, yellow or gray dot you see in user lists in realtime applications). Both databases use a rules file to control access, RTDB is more limited, specifically not good for group access; Firestore allows queries to be used to determine access in the rules - providing group and/or data driven access.

- JSON formatted data storage with query and sort abilities (no sql)
- Client side API for realtime/pushed updates via socket based connections
- Integrates with Firebase Authentication
- Scale on demand - pay only for what you use
- Firestore is generally the better option
- Using both isn't rare - i.e. RTDB to monitor user connection status
- Firestore is based on Google Cloud Data Storage and has many advanced utilities
- Neither database is SQL friendly, but single collection/table queries work
- Data must be designed considering the abilities of the datastore
- migrating an existing SQL database will likely require many data and code changes

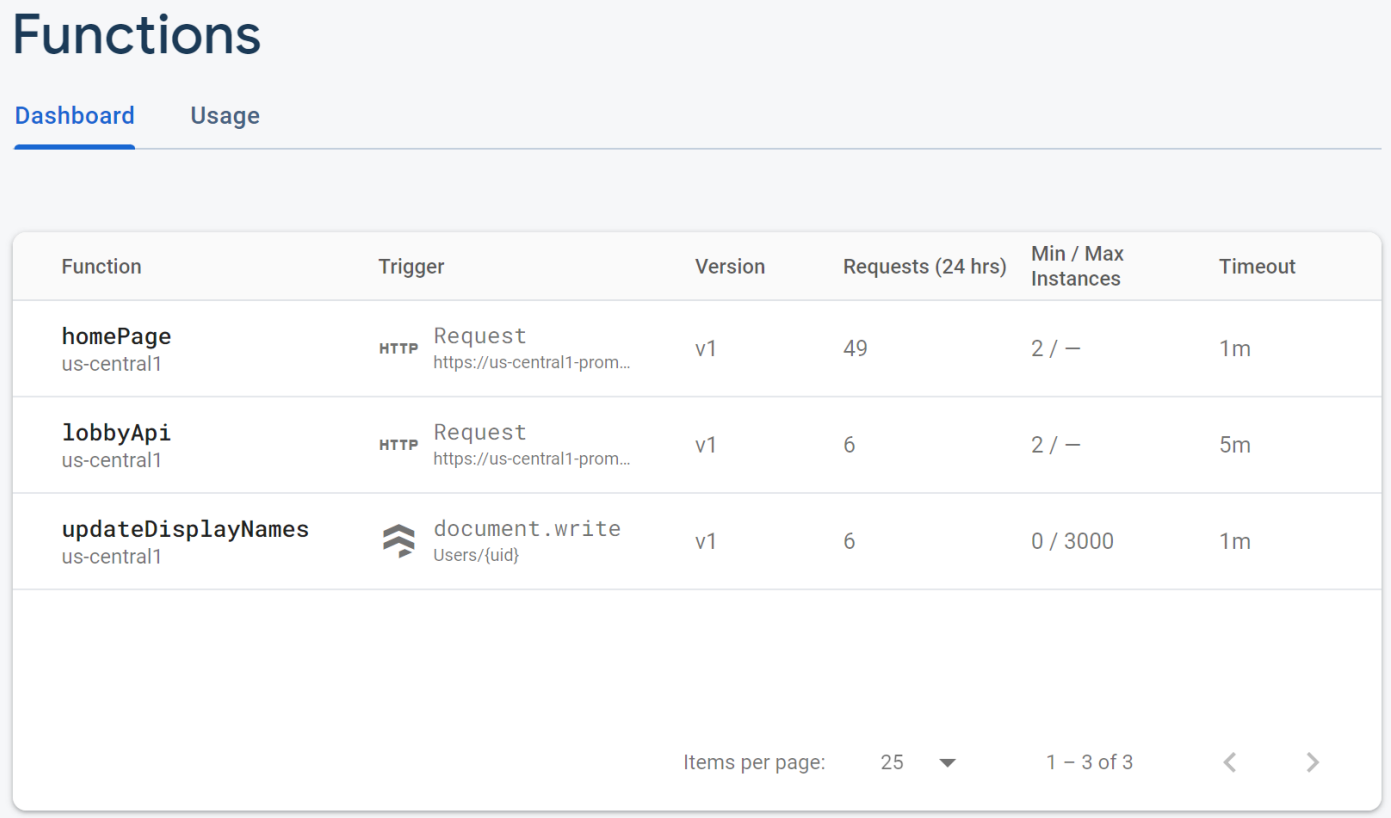
Firestore Console with Collections and Subcollections



Cloud Functions (server side code)

https://firebase.google.com/docs/functions

Cloud functions are most often written in Javascript/Typescript to run on node based servers. Python and others languages are available when using Google Cloud Functions and show up in Firebase the same as Firebase functions, but require additional deployment steps outside of the Javascript centric Firebase platform.



Many operations require data that isn't meant to be read or accessed directly by client applications (user access) - such as API keys or owner information for shared documents. Other operations require that process lots of documents or a large size of data perform faster and cheaper running in the cloud.

In general to access the realtime features of Firebase data, direct access to the datastore is required - but only read access is required for this. When writing data to the cloud (not files/blobs such as images), it's best to use a cloud function on the server to validate the data and write to the datastore. Client side code can be easily manipulated - and as a developer must be expected - so any compartmental security or application secret data (such as API keys for external sources) need never be accessible to the user.

Server side html rendering is possible via cloud functions for firebase, although page generation is a touch slower than static pages in the best (cached) case. Some advantages are all of the code in the same project and the scale on demand functionality inherit to cloud functions. To deliver fresh content to SEO - server side rendering works best, especially considering meta data (for OG previews on social sites, etc). Since client authorization is possible and common on client applications (web or native) - utilizing client side content rendering inside of a static html page is the preferred technique to keep costs down and performance high.

Triggered data, time and authorization events are also passed to cloud functions for processing, such as deleting an account or adding a record or blob can call a function on the server to perform additional

processing - i.e. deleting an authorization account will call a function to delete all associated data, etc - aka “webhooks”.

Cold Boot Considerations

Cloud functions compile and load in a memory cache after being be used (for a short time) - and the base overhead on a call that is cached is about 50ms. Cold boot happens if the application hasn’t been used in a while, so the code must be loaded into server (cloud) memory - and this takes longer and result in a lag - of about 200ms additional time. After the cold boot, the function is cached and will perform decently fast. To get around the cold boot issue, Firebase offers a min instance running feature - which costs a monthly minimum rate - about 1-2 dollars per instance. This eliminates the cold boot delay for small sites - so there is always a cached version. When scaling to many users - the cold boots will be mixed in as the application ramps up to meet user demand - but this is the ideal solution versus not having enough servers (slow for everyone often) - or having too many running not doing anything. Booting entire servers on demand is very slow as it generally takes more a minute to expand computing capacity - although most applications have peak usage during a certain time of day and can boot servers ahead of time. For smaller application that don’t know when their flash mob day will happen (unexpected influencer post, etc), the confidence the application will scale to meet the demand seamlessly and quickly is key. Cold boot isn’t a hard thing to manage, but it’s something to be aware of if your application seems slow at times, and there are a number of affordable solutions to make it work.

NodeJS and Express

Express works fine for routing, so one cloud function - i.e. site.com/function can be used to access different code based operations via route - site.com/function/operation1, operation2, etc. For NodeJS developers this makes it easier and faster to use existing techniques. The downside is each operation won’t be separated out for cost, performance and error handling and tracking by GCP. We’ve had a few clients who want each API route to be it’s own Cloud function so they can use GCP to more closely monitor and manage each operation. Most clients want it done the cheapest now and later - which is using one function for many operations, there is a small amount of coding and deployment overhead that goes with having many cloud functions. Having a function per API operation allows the developer to pick a language of choice to code each function in - so you can mix Python, NodeJS and other languages for each function. In general having all the files in one project allows easier code reuse and code management.

Costing Considerations

Execution prices are generally not a concern, but not the only charge. Runtime is the most critical - and hanging functions (API calls) run the full length of the timeout (usually 60s - but can be set up to 9min). The cost of this is unacceptable and will be detected and need to be fixed if not carefully considered throughout development with quality promise and error handling. Sizing of the instance (done by memory) helps control the cost, in general you can use a very small size - the configuration is adjustable per function, so isolate your longer runtime functions and provide larger instances and longer timeouts to do heavier jobs. Storing keys and using them call external APIs is a regular practice to keep keys private, but cloud functions may not be the most cost effective vs hosting an instance for long jobs - especially long jobs that are just holding a socket open waiting on an external request - as you pay for the compute load whether you use it or not. In general cloud functions do microservice like operations and using data, storage and compute in the same cloud nothing usually runs longer than 30s - a couple seconds is generally a lot of processing if not waiting for an external service to resolve.

Google Cloud Storage (for images, videos and other large files)
https://cloud.google.com/storage

- Works best with files that are cachable - user images, etc
- Cheaper than data storage, affordable way to host videos, etc
- Any data that changes often should be in a datastore - not blob storage
- S3 is AWS's solution and has for a long time been a bedrock in the cloud - Firebase/Google's solutions work well also
- Uploading blobs directly to cloud storage is usually the best choice and for files larger than 10mb, the only choice
- Any data that will be filtered/queried should live in the datastore - if the blob sizes are too large, then the data that files are being filtered/queried on should be stored in a datastore and blob in storage (i.e. a movie file with a title and cast, etc).

S3 is well known and for scalable and affordable blob (file) hosting is extremely heavily used. Google Cloud Storage works very similar and in addition is tightly integrated with Firebase Authorization - including a rules based access system similar to the datastores. Create, delete and update triggers are available. Listing the contents of a directory is possible, but generally not a cheap or fast technique to use in blob storage - so in the regard it's not like a giant disk drive in the cloud, but reading and writing blobs (files) to and from storage is very similar to storage on a hard drive. Public access files are possible. S3 and Cloud Storage have many of the same features and both are modelled around "buckets" - which are similar to a classic computer disk drive. The main reason we usually use Cloud Storage over S3 is it's all integrated into one bill and one project (for rules based access).

Application Notifications
https://firebase.google.com/docs/cloud-messaging

Application notifications are the popups and updates you see on your phone that are not SMS updates. While these work on Android webapps, they do not work on Apple devices (for webapps) - so in general people usually only supply this feature on native applications. A web application can be wrapped up with Apache Cordova (formerly Phonegap) - but most projects I've been they have already opted for a native application - mainly to provide this functionality - at great and unadvised expense and time to market. Firebase provides a framework and platform to deliver this to native and web apps (Android only). It's listed as a bug that these don't work on Apple (nor the camera for web app browser access) - and they haven't been fixed in over 5 years - don't expect them to be.
In general there are several details that are important to get your messages to show, and in addition your application needs to be written to handle these in a very thin way - you don't want to boot the app and initialize lots of data to show a couple lines of text and maybe an image - so this needs to be considered.

Deep Links for Native Applications
https://firebase.google.com/support/dynamic-links-faq
While in the past this has been supported by Firebase - one solution for both Android and Apple, Firebase is hanging up the keyboards on this feature in 2025 - and recommending external providers if you want the same feature set.

They recommend migrating to use a service specific to each platform (Apple or Android) to support dynamic links - where if an application is installed, it will boot the content and show it in the application, if not it'll load the website for you - also these links can install applications, etc.
https://firebase.google.com/docs/auth/android/email-link-auth
https://developer.android.com/training/app-links

Passwordless login to native applications is possible using a link provided in email.