Money-Media Programming Assignment Part B

Based on the first money-media programming assignment, please be prepared to describe

how to create a web-based server version that would:

- Allow from two to five people to play the beginnings of a five-card poker game together

- Allow multiple games to be played simultaneously, each with a different set of up to five

people

- Each game would deal five cards to the players participating in the same game

- The server would keep track of and coordinate the player selection, play behavior,

games, cards, results etc

Note this is not an assignment to actually build a web-based poker system. Instead, we are

asking you to consider and explore team, system and architectural design concerns involved

in building a full-fledged system.

Example concerns: team composition, build and testing processes, storage, schema design,

data layers, service layers, web frameworks (streaming? Traditional request response?), etc

1. Planning

Once requirements have been concretely established for product, software development life cycle enters its most critical phase; planning and architectural design.

Team Composition

There are certain givens for a product in terms of initial team composition. In the case of a poker system, there is clearly a backend and frontend. The scope of this specific product is fairly small and as such a single backend developer, a frontend developer, and a tech lead that handles architecture would suffice. A designer to create demos via Invision and supply CSS would be very useful. As a rule of thumb, there should be two QA to one developer. A business analyst or product manager to assist in transparency through JIRA would round off the team. A SCRUM Master and a dedicated devops engineer would certainly be a nice to have.

Research

This is not a cutting-edge product without peer, there are many games in existence that we can conduct research on that can help in creating a product that lasts. Questions that should be asked:

* What is the expected scale? What are the estimated numbers of the competition?
* Are we serving different regions and countries? If so, what are the security concerns we should be way of. GDPR in Europe for instance.
* Do we expect the product to be a first in a new lineup of games? Should we architect our system in such a way that it uses Event Driven Architecture and dumps every action into a queue that can be picked up by other services.

Architecture

The research questions certainly affect architecture profoundly, however, for the purposes of this competency test, I’ll operate under the conditions that they are unavailable or non-issues and the system we build should be scalable to any load demands. Lastly, I believe architecture is something that should be evaluated from multiple personnel and not built in isolation, it should also be something that is questioned. Product, QA, development staff should all scrutinize.

Frontend

In this product, I see the frontend component as being quite straight forward and something that requires the typical modern-day requirements: Event driven, state management and routing for an SPA. Any of the big three (React, Vue, Angular) can satisfy the requirements, however given the small scope of the requirements, Vue would seem to be the best bet.

Backend and Server

There are several approaches that can be taken on the backend, but there are certain consistencies. A CDN would be very helpful to store public facing assets and either a cache aside or write through cache would be ideal regardless of the approach. As a rule of thumb, in terms of long-term viability having a queuing server with an underlying log (such as Kafka) to maintain historical events is always useful. An event queue allows for future microservices to easily pick up and pass events from the application.

One of the biggest questions that should be asked is what is the communication medium? Is a streaming architecture needed? Do we need the malleability of Graphql? Is the REST overhead too much and a MQTT / IoT a better solution? For this exercise, I don’t see any need for streaming or the benefits of graphql and REST will perform just fine. **Decision: REST**

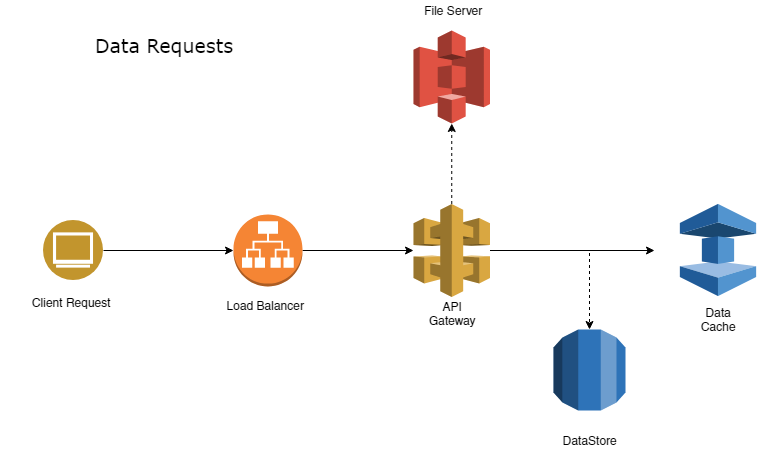
The data store ultimately is the core of any architecture and as such questions that need to be asked is our application write or ready heavy? What is our replication and fault tolerance strategies? Is relational needed or would unstructured data suffice? Perhaps a graph db would be enough instead of full relational? Is ACID compliance needed? Would we benefit from a time series db? My working assumption is that the application would benefit from relational and transactions and it would lean more towards a higher volume of reads. Ultimately, it should be confirmed through a proposed schema mapping that the relations aren’t extensive to the point that the application will need more write throughput than any of the major relational stores (Mysql, MariaDB, Postgres). **Decision: Relational**

Caching should be a major part in any application. As mentioned above, a CDN and a write through data strategy utilizing Redis would be good starting points. Redis is preferable over Memcached in this situation given its data structures. There are certainly other levels of caching that should be considered as well. For instance, is there any benefit at the API level from a reverse proxy cache such as Varnish? Given the cascading and real time nature of the application, varnish would add quite a bit of complication in terms of invalidation strategy. It certainly could add fault tolerance through saint mode but ultimately, I don’t believe it’s worth it. **Decision: Redis, CDN**

One of the biggest and most consequential decisions is the language chosen to write the backend API server. Questions such as the need of asynchronous, threading vs processed and available developer skill need to all be considered. I believe this application would benefit greatly from an asynchronous language, ruling out something like PHP which doesn’t handle async well. I believe the application would benefit more from threading as I imagine it to be I/O bound. I also don’t see the application being heavily calculation based nor does package consistency need to be ensured and as such I don’t believe there to be a need for a compile language. I think Python with Flask works well here. **Decision: Python**

One of the worthwhile concepts to discuss is if the metrics and meta data should be separated into their own database and thereby their own micro-services that govern them. It could go a long way in terms of long-term viability to have statistics available on a reporting database. For the purposes of this product as it’s defined, I believe simply caching our statistics would suffice.

Finally, cost is always a consideration and while it should not dictate architecture it should certainly guide technologies chosen. Decisions such as whether to go serverless, or managed solutions should all be affected by costs. Note, decisions that should be considered are authentication and how the content will be served between regions. Authentication would be best served with its own architecture and service, as well as data store that return user authorization. For the scope of this exercise, neither of these are concerns and as such is ignored.



1. Validation and Testing

I’m a strong believe that decisions should be reviewed, questioned and defended with data. Any proposed architecture should be spun up in the form of a PoC and then validation tests performed to mimic scale.

1. Build strategies

There are a number of build strategies and tools that can be utilized. I believe a lot of this is dictated by the scale of the application, git merging strategies and team composition. Given that the proposed solution is comprised of a backend and frontend, a Jenkins solution supported by docker-compose or Kubernetes to spin up our application servers would certainly suffice. Develop, qa, staging and production environments would go a long way in performing isolated tests, as well as functional ones.

4. Development and Testing

Once an architecture has been settled upon, the next major steps are deciding on testing strategies, the breaking down of tasks and providing visibility into progress from the team to stakeholders.

High priority tasks:

* Creating designs and demos through Invision
* Gathering feedback throughout the process
* Deciding on the deliverables and product milestones
* Establishing epics and understanding who can and will work on what
* Creating test cases that developers can build towards
* Providing documentation and visibility into and during the process

5. Process and Communication

I’ve always found that following agile over waterfall along with the typical principles of SCRUM have always worked best in terms of guiding principles of process. Adherence to a process, regardless of what is decided on, and maintaining consistence goes a long way towards success. I’m also a firm belief that team members shouldn’t be handing things off as it leads to lobbing back and forth, QA, product, development should be involved from the start, middle and end. As such, loosely following BDD has always worked well for me.

6. Testing

Testing is not something that should be considered towards the end of the product, I believe it should be among the first concerns, especially consideration towards building automated end-to-end tests. BDD is often difficult to strictly follow, but having qa staff establish and document test cases goes a long way towards building communication. Testing is also not just the responsibility of qa, the onus is on development and product as well. We should be building applications that can be tested at a unit level and an integrated level.