Celestial TCG Server Rewrite

Kai Geffen

July 9, 2024

Abstract

Explaining the motivation and plan for rewriting the server-side code for Celestial TCG (A card game web-app)

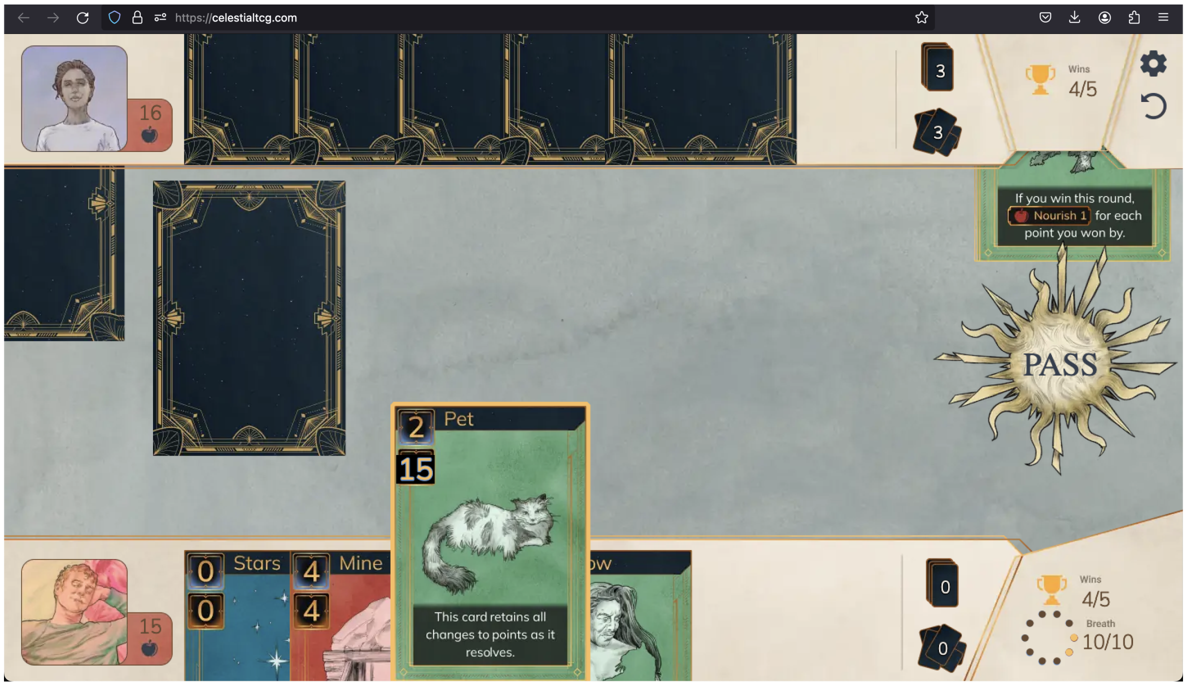
https://celestialtcg.com
16
3
3
Wins
4/5
If you win this round, Nourish 1 for each point you won by.
PASS
2
Pet
15
This card retains all changes to points as it resolves.
0
Stars
0
4
Mine
4
15
0
0
Wins
4/5
Breath
10/10

Figure 1: An exciting game of Celestial TCG.

tl;dr: Server is Python for historical reasons, transitioning it to TS and making the project a monorepo with the server. Separating out shared logic, adding typesafety to RPCs, adding performance tests.

1 Introduction

Celestial TCG is a web-app card game the code for which I solo-developed over the last 4 years which is in use at CelestialTCG.com. The project has 4,000+ hours of work in it spread across 15,000+ lines of code split between the client (Typescript) and the server (Python). Additionally, it has been contributed to by 8 artists, writers, and designers.

Initially, this project was a fan game for the popular title Celeste, and the implementation was peer-to-peer raw UDP connections in Python. This meant that any player needed to install the project and dependencies, specify the IP of their opponent, and adjust their router configuration to allow the connections. This solution was clearly flawed, and so the code for the logic of the game remained in Python while the view shifted to client-side Typescript code. The protocol also changed to websockets, and eventually included OAuth and Postgres for user database management. As it exists now, users can access the game from a webpage. Simple.

The game-logic in Python has persisted and not needed to change significantly over the last 3 years. It did grow to include PyTorch to train more effective AI models, although that has largely been put on pause as existing heuristics are sufficient for computer opponents, and heuristic-based computer opponents is industry standard as best I can tell.

One place the client codebase has grown is to accommodate different connections (Sign-in, opening packs, finding matches). This Python code also must create a consistent interface with the Typescript client code for things like encoding/decoding game-state. Additionally, the server network code is some of the most in need of a rewrite out of anything in the project. Having the language used for the client side of the interface be the same as the server side simplifies the logic, and allows for easier separation of shared logic.

I delayed on doing this rewrite for a while, unsure of what direction to take this project in, but now feels like the time. I expect the process to take about 60 hours of work, and yield the following benefits:

1. Improved client/server interface
2. DRY shared logic between client/server
3. Better scaling for the client code with TS than Python (Ability to grok a growing codebase)
4. Monorepo structure instead of 2 separate repos
5. Performance test coverage

2 Topics

2.1 Shared logic

Currently, some of the details about the cards as well as the game logic exists in both the client and server. This isn't DRY and allows for issues where I forget to update some value consistently between client and server, causing (Largely visual) issues.

While my inclination is to create a repo with shared code files, plus 2 submodules for the client and server (They each have their own repo), discussion with other engineers has convinced me otherwise. Monorepos are more common practice today, even at big tech companies, and certaintly for projects of this scope. The decoupling that my solution would achieve just isn't worth the complication of submitting separate not-inherently-connected commits to the different repos. One loss is if their are a variety of frontends that I want to support (For example, webapp, steam, console, blockchain, whatever), but currently that isn't a priority. Either way preserves the history for the client code, and improves my deployment setup through better docker configuration.

2.2 Testing

The unclear scope of this project has resulted in it being one of the least tested that I have worked on. Even many of my smaller projects have more extensive tests as I tend towards Test-Driven Development (TDD).

While the website is surprisingly bug free, that is not the main purpose of tests! Tests clarify expectations, they make adding or adjusting features easier by giving you assurance that what you've added hasn't broken something else.

Many of the bugs that do exist or show up are related to view (As in Model-View-Controller) and are classically hard to test. For example, on some setups hovering a card glows gold, then moving the mouse away quickly keeps the card glowing even when it should fade out. However, in that paradigm the controller is the easiest part to test, which is what the server is. With the scope of what that code should accomplish much more clear now, writing tests in advance for it is a great opportunity that this rewrite allows for. The process of separating out shared logic also allows for a cleaner divide of view versus controller, which means more of the server side code will also be better tested moving forward.

2.3 Machine-Learning library

My use of PyTorch as a Machine-Learning library came more from a desire to play with ML than a need for the ai opponent to perform better. It did have the potential to make the computer good enough to approximate a human opponent, which could address the issue of a needing a large player-base to get a player-base. That tactic of growth for the game may be considered in the future, but doesn't seem like the best option (Compared to bootstrapping from smaller communities playing the game, or seeking a publisher).

TensorFlow is a popular Javascript library which could accomplish the same task if it was ever important, so that doesn't feel like a major lost opportunity upon shifting away from Python.

3 Proposed Structure

The following is loosely the structure of files I expect for the rework. In this section I discuss how to approach each item and what parts to make dry.

1. Network Interface (Websocket, OAuth, Codecs)
2. Individual Card Logic
3. Game State Elements (Story, recap, controller/model for core game-logic)
4. Assorted Effects (Statuses, sfx, animations)
5. Special Situations (Tutorial controller)
6. AI Logic

3.1 Network Interface

The message literals (Init, mulligan, play_card) Should be dry between client and server. Also the pathing for password matchmaking / special matchmaking.

Much of the Python code should be thrown out, and an implementation handling threads and sockets similar to the existing Typescript should get written instead.

3.2 Individual Card Logic

The actual implementation and heuristics for each card can be exclusively on the server, but much of the other information about each card (Such as its name, effective cost, etc) should be shared with the server.

This has been proposed, but is too extreme to justify the effort:
A card's text should directly derive from its logic, such that changing the implementation of a card automatically updates the card's text without a need to paraphrase. So for instance if a card was "Nourish 2" and changed to "Nourish 3", the text should reflect that without needing to update it as a field literal. To be fully expressive, this would mean creating a DSL for cards, which expresses everything possible for a card to do.

3.3 Game State Elements

The existing Python code organizes the core logic for the game into a controller and model, consistent with MVC standards. The model is the game state that's sent to the client (Partially, since there each player has less information than the full game state). Also, the model is what each card has the authority to affect (Any ways in which an individual card can affect the game state is also exposed through the model). The structure of the model should be shared, but any actions that a card can take on it can be local to the server.

A large part of the model is a module called the Story, which expresses each of the cards that have been played during the current round. This module is in some ways both the model (Expressing that state) and the controller (Handling the story's resolution at the end of round). The majority of this shouldn't be exposed to the client, except the structure in which the model is represented.

Additionally, a state which is preserved is the Recap, which shows essentially the different game states that the game moved through as the story resolved on the previous round (As well as what the resolution of each card caused to happen). This representation of past state is more thorough than in other similar card games, and at the same time doesn't capture events outside of story resolution (Such as Morning, which happens at the start of a round). The client code for displaying recap is also one of the hairiest in that repo, so this aspect of the codebase might need to be changed soon.

3.4 Assorted Effects

Other effects (Like card-specific sound-effects or statuses that affect a player) are relatively simple, and much of them should be shared since they are chiefly just literals that are agreed upon between client/server.

3.5 Special Situations

Currently, the only special situation is the tutorial controller, which causes the game to have deterministic and slightly simplified rules while the player is learning the game. This code is always going to be something of a mess, and is subject to large-scale overhaul. Getting it to function and have basic tests expressing what it should do is good enough.

3.6 AI Logic

The AI logic for handling the heuristics and core decision making for the computer is exclusive to the server. Switching this out with a trained model is currently easy, but preserving that without clarity on if it will get used isn't a priority.

4 Testing

Because this has been MVC, there's obvious opportunity for unit and integration tests for the controller (Server) code. But I also want to highlight a few opportunities for further testing, some of which I'll actualize for this project.

4.1 Performance Tests

Currently the game has very few active users, and at most has had 4 concurrent games at a time. The most popular games (Hearthstone, Magic) have a quarter million, and less successful games that could still be considered "alive" peak at 100. It feels important to test for the current interface solutions' ability to handle a reasonable load (100) both to ensure that events like tournaments that cause a spike and any influx of players can be handled without issue. This is especially important to address before it happens because the interface is some of the hardest to change in a short time-window.

Because of the nature of turn-based games, latency in either a computer or human opponent's actions isn't a big deal. But it's still worth testing for latency there and in other places. Reconnection and user authentication are another place to test.

Tests that I will implement:

1. 100 concurrent users can play the game (Against random opponents, against passworded opponents) without noticeable change in quality
2. 100 users can be signed in at the same time
3. Latency in a match between 2 players meets some threshold
4. Latency for user sign in meets some threshold
5. Latency for user account creation meets some threshold
6. A user can stay signed in for a long time (8 hours)

4.2 Compatibility Tests

With webapps, you get the benefit of it working on any device. And you get the curse of it working on any device.

Because there are so many different canvas dimensions and permissions on different environments, it becomes very hard to ensure things look good and function on all of them. This isn't an issue exposed by the redesign, since the code involved is the server, but now is a good chance to improve the testing if I have the time.

Low hanging fruit are ensuring that it runs at all on all supported environments (Smoke test) and that all assets load and work, but it would also be great to have automated tests ensuring that elements of the different scenes don't overlap in such a way that any become hidden or don't meet accessibility guidelines for clickable elements (Too small).

Unfortunately, after soliciting feedback, it appears that this is a hard problem which doesn't have a good solution. Industry standard is in many cases to either manually check the environments you want to support, or use a service which automates this process and sends screenshots for you to manually review. Chrome's Lighthouse is one tool, services like BrowserStack are another. I will create a list of environments to support, and check a given set of user-flows for each upon major releases, but may not automate further than that.

4.3 Scenario Tests

By this I mean a situation in the game which behaves differently than it should. Some of these require collaboration with playtesters, and may not be in the scope of this redesign.

One example is how the start-of-turn used to work caused an edge-case bug: Nightmare has a conditional trigger based on if you have more cards in hand than the opponent. But start of turn worked like player A triggers their morning, then draws 2, then player B triggers their morning, then draws 2. This resulted in situations where player B wouldn't trigger Nightmare when they should have.

Some of these tests are implemented in the server code currently, but all of them and more need to be added to the new repo. This also helps to clarify complicated or edge situations in the game that the rules don't elegantly resolve.

5 Funding

As a programmer, I require a reliable diet of rice and beans to ensure high-quality code output. I expect the work to take around 60 hours, which is doable in a week.

$$\begin{aligned}
&B : \text{Basal Metabolic Rate (Calories per day)} \approx 1400\\
&T : \text{Time to complete (Days)} \approx 7\\
&R : \text{Cost of rice \& beans (Dollars per calorie)} \approx 0.003\\
&C : \text{Cost of rewrite (Dollars)} = B \times T \times R = 1400 \times 7 \times 0.003 \approx \$29
\end{aligned}$$

This project is thus seeking a generous donor to supply the 29$ USD necessary to make this dream a reality.

6 Feasibility

The python codebase is 4,000 lines, I have existing examples of how to use websockets and OAuth in the client side code, and I'm transpiling from one well-documented language I'm familiar with to another. Writing tests for this code which is largely separated into Model + Controller already should be fairly easy, as well as implementing existing tests for special situations that arise in the game.

Breaking out shared logic into its own repo is quite doable, this is a great opportunity to do it as much as possible. If time runs out, this part is less critical, and can continue to happen over time in response to need.

The effects of this rework should be invisible, but will help the project continue to grow in the future.

The following **stretch-goals** have been highlighted above - most won't get completed at this time:

1. Clarifying and improving the user-flow for the recap system / a way for the player to see what has happened.
2. High-quality AI which learns how to make decks and pilot them.
3. Compatibility tests to ensure correct rendering and performance across devices, browsers, and environments.
4. Writing a DSL for the cards, such that changes reflect in the cards wording, effect when played, and even the Figma containing the card images.