Devlog 3 - November 8, 2025. Session Management, Lazy Loading and Refactoring Pains

This week has been spent working on improving the backend.

Session Management

The GameController, as its name implies, a kind of orchestrator for all the systems of the application, has already undergone several refactors since its creation. It began as the logic and data access script, where everything was put together. Then I added a couple of big services, for scenes, talent, hiring, etc, while the controller still did a lot of CRUD and miscellaneous logic, while connecting the new services to the UI calls. It also had the final say in most of these calls, committing changes to the database, and emitting signals to the UI to let them know there had been a change and they needed to update their state. It also had plenty of logic for methods that required a lot of high level orchestrating between services, like the method that holds all the logic necessary for a turn to advance.

This didn't follow any hard set of rules, and as appropriate, it depended a lot on vibes to decide what went where or who did what. Which was mostly manageable when the amount of logic necessary for the game wasn't that high, but as files started reaching the 1000 lines, it became a real pain to figure out what was going on in any complex method that depended on other services. The main pain source was the session management; the way the live database connection was handled during a running game.

Before the latest refactoring, a single session instance was created on game initialization (whenever a new game started or a save was loaded). This single instance was passed around as appropriate by the controller to whatever service needed it at the time. This worked fine most of the time, but as some methods from services would sometimes commit to or rollback data, it could mean that a different one might be working on stale data, if its session wasn't 'refreshed' after the commit happened. This is usually a problem when a multi-threaded app is doing asynchronous tasks; each thread is working independently until they reach the final point, and they put their database versions together, and they find out that there are conflicts. For a single-threaded app (at least for now), this is quite unforgivable.

Service Cleanup

The individual services had also got quite fat, with each one managing everything related to their domain, and some methods (like the one to release a scene) were huge, with every calculation and query made to complete it wrote in full inside a single monolith. Which makes simply looking at some calculation method

harder than it should.
They also had to know about many other services and loaded the whole game data (i.e. the rules, how much stamina a tag costs, for example).
Breaking both the services into even more specialized ones (dividing querying and calculation methods, for example) and the methods in smaller units (with each individual process inside it a different method) was also necessary then.
It would also help with testing, something I haven't been able to experiment much with yet, unfortunately.

These services now operate under the Single Responsibility Principle: they only handle one particular facet of the game systems (tags, events, hiring, etc); make use of a session factory: they can create their own session instance as they need it; the Unit of Work pattern: every transaction (hiring a talent, shooting a scene, advancing a turn, etc) is committed once, by the creator of the session, after every checkpoint has correctly passed, or rolled back otherwise; and Data Transfer Objects: they don't need to call back and forth to other services because the results of one method are instead encapsulated in a dataclass and then shared with whoever needs it, and they don't need to load all the game data, but instead get custom dataclasses that hold only the rules that they care about (so the hiring service doesn't have to load at what rate the spending willingness of a viewer group depletes and regenerates).

Finally, I also refactored most of the views to strip them off all logic and calls to the controller, which are now handled by their 'presenters', which is the only element the views now know about (aside from the settings manager due to how the geometry persistence works).

So now the flow of the game (in the background, anyway) looks like this: the player clicks on something or modifies some number -> the view emits a signal -> the presenter catches the signal and calls the relevant controller (proxy) method -> the controller delegates to the proper service -> the service handles whatever it is that needs to be done (which might involve other services too) -> the service emits the appropriate signal(s) once it's done -> the presenters catch the signal(s) and tell their respective views to update.

Fuzzing Improvements

I talked about how badly the game was performing when it came to populating and filtering data after adding the skill fuzzing in the last devlog, so once I was done with the session management and service refactoring, I decided to look into it, since I still had some time in the week. The biggest impact on performance happened due to the eager loading of every single talent to the talent view whenever a game was loaded or started, which meant calculating the new 5 fuzzed skills values for every talent at once. This has been changed to instead only calculate the values for the visible talent at any time, which means that the skill ranges are calculated in real time during the scrolling. This has a negative impact on the smoothness of scrolling, but I don't think it is

an unacceptable trade-off, with the performance hit being almost imperceptible unless the scrolling is very fast.

The second improvement was achieved by caching (i.e. saved on the live memory) all the already calculated values. This doesn't help with the first load, but it does with every other one after that, which means it also helps with the filtering.
An extra improvement would be to implement a 'partial cache'. At the moment, every time that a cached value is flagged as 'dirty' (i.e. the value has changed after it was stored), the whole cache is nuked, which means that it has to be built up again. Realistically, this means that every turn, the cache is destroyed, because at least one talent is going to have some change, even in weeks where no scene happens, due to popularity decay.
Partial caching would solve this by making the cache more selective: instead of nuking everything because some talents changed, it would know which talents changed and destroy (and then rebuild) only their cached info, while maintaining the 'clean' cache intact.
To allow for this, the signals system in the game would need to be a lot more informative that they are at the moment. Right now, any time a talent's value changes in some way, the pertinent service throws a 'talent changed' signal, which makes every UI widget that listens to that signal update its view. Instead of this, the game would carry a list of all the talent that has changed during a turn. Then the caching system would take that list, and mark all of those talents as dirty, deleting their info in the cache.

So why didn't I implement this as well? I tried and it completely broke the disconnection system that handles deleting the live-session file that stores the info that is then copied to a save file (if it is). Windows doesn't allow a file to be deleted as long as something is making use of it, so for the file to be able to be deleted, the game has to disconnect the database connection to the file. This is quite straightforward, the issue is that some elements might persist in memory even after the database has been disconnected, which is why explicitly calling Python's 'garbage collection' to get rid of everything in memory that shouldn't be there anymore was necessary when implementing this deletion system.
When I implemented the incremental caching, something with this broke, and I was very much not in the mood to deal with it, getting the file deletion working properly just a couple of days prior had already been a big headache. So I decided to, just like our current caching system, nuke everything to do with the incremental one and perhaps save it for later.

Something that needs to be done and might make the incremental caching kind of superfluous, is to only re-calculate the fuzzing logic every 10 or 15, or perhaps even more turns, depending on what experience gain rate we settle with.

Now, none of this is completely done; there are still a couple of full methods living in the controller that won't get taken out until I change how the talent generator works (probably once I implement talent generation in-game, not just on game start), some views still hold plenty of logic and controller calls (start screen, main window or the scene planner view).

While I think this was still a very positive development, it feels like a sidestep, so I want to get with adding new stuff already.