[Skip to content](https://chatgpt.com/share/68e3ffcb-e18c-8007-83d4-ff5866c05890#main)

**Chat history**

This is a copy of a conversation between ChatGPT & Anonymous.

**Report conversation**

**You said:**

After going through a few iterations of this idea, I've come to a better understanding of what I want to do. The goal is to make an engine (I'm calling it GrimEngine) to play (terminal console) and support play in other UIs/platforms of solo D&D 5e Campaigns - we can use open ai api where needed / where it makes sense, but I want to minimize the amount we call to that since it costs money. further to that end, I want to cache responses where it makes sense. - We can use the D&D 5e api for most of the other things - also caching responses we get where it makes sense. I know that it is free, but the less we can call apis, the less the app has to deal with that overhead. Previous attempts at this and stored a bunch of downloaded json. I've attached the D&D basic rules PDF to this chat gpt project to help us get started. The very first thing I want to do before roadmapping or planning anything is to decided on what system to code this in so I can establish a repo on github (i do have that connector). Let me know if you have any questions or concerns.

**ChatGPT said:**

Thought for 31s

Awesome—love the scope and the constraints. Given your goals (solo 5e engine, terminal-first, easily pluggable into other UIs, minimal/cheap API usage with smart caching), here’s my strong recommendation:

**Pick this stack**

**TypeScript + Node.js in a monorepo** (pnpm + Turborepo), with a **headless core engine** published as a package.  
Why this wins for GrimEngine:

* **One language, many surfaces.** The same TS core can power: CLI (Node), desktop (Tauri), web (Next.js/React), Discord/Slack bots, and even service backends—no duplication.
* **Type-safety for complex rules.** Modeling 5e mechanics, combat flows, and state machines benefits hugely from TypeScript’s types and discriminated unions.
* **Great DX + ecosystem.** First-class tooling for testing, linting, bundling, and API clients. Plenty of caching libs and SQLite/embedded DB options.
* **Future-web ready.** If you ever want a browser UI, TS is the natural fit. (We can even compile rules to WASM later if needed, but you likely won’t need that.)
* **LLM control.** Node has robust middleware patterns for **prompt caching**, **response dedup**, **rate limiting**, and **offline fallbacks** to keep OpenAI costs low.

Second-best choice: **Python** (Typer/Rich for CLI, FastAPI for services) if you anticipate heavy local NLP work or want quick scripting ergonomics. It’s excellent too, but TS keeps UI paths simpler and unified.

**High-level architecture (monorepo)**

grimengine/

apps/

cli/ *# Terminal game (Node). Uses @grimengine/core.*

sandbox/ *# Example playground, scripts, profiling.*

packages/

core/ *# Headless rules + game loop + state engine*

adapters/

dnd5e-api/ *# Client for D&D 5e API, with aggressive caching*

rules-srd/ *# Local SRD/basic-rules data extract, normalized JSON*

openai/ *# Optional: LLM helpers with prompt/response cache*

persistence/

cache/ *# SQLite-backed cache (HTTP + prompt + asset)*

storage/ *# Save files, world state, journal entries*

tooling/

types/ *# Shared TypeScript types*

test/ *# Test utils, fixtures*

tools/

scripts/ *# Codegen, data importers (e.g., PDF→JSON processors)*

.github/

workflows/ *# CI: lint/test/build, cache warmers*

turbo.json

package.json

pnpm-workspace.yaml

tsconfig.base.json

**Key design choices (keeps API calls and cost low)**

* **Headless core**: All game logic (initiative, rolling, conditions, rests, exploration/clocks, oracle tables for solo play, etc.) lives in @grimengine/core. No I/O in core—pure functions/state machine—so you can reuse it in any UI.
* **Two data sources with caching-first policy:**
  1. **Local SRD/Basic Rules JSON** in rules-srd (preprocessed once from your PDF and/or public SRD datasets).
  2. **D&D 5e API adapter** that only fires when a resource is missing or stale; results are **normalized + written** to local cache for future offline use.
* **SQLite cache** (via better-sqlite3): unified layer for:
  1. HTTP GET cache for D&D 5e API (ETag/Last-Modified aware)
  2. **LLM prompt/response cache** (hash prompts incl. system/user/tool context; store completion + tokens used)
  3. Asset cache (stat blocks, items, spell blocks)
* **Deterministic randomness**: seedable RNG (e.g., seedrandom) so sessions are replayable/debuggable.
* **Zero-cost mode** for LLM: pure-rules fallback where the “oracle” is a table/procedure (Solo/GM emulator). You only enable LLM for flavor text or non-critical adjudication.

**Packages in a bit more detail**

@grimengine/core

* State model: campaign → scenes → encounters → actors (PCs/NPCs) → stats/conditions → inventory.
* Subsystems:
  + Dice & advantage/disadvantage
  + Initiative & turn engine
  + Movement/position zones (tactical-lite for terminal)
  + Checks/saves/attacks/damage
  + Resting, travel, clocks, random encounters
  + Compendium lookups via injected providers (SRD, API, custom)
* Pure API: (state, event) => newState + selectors. Great for testing.

@grimengine/adapters/rules-srd

* Prebuilt JSON datasets for classes, spells, equipment, monsters (SRD-compatible).
* Versioned data with schema migrations.

@grimengine/adapters/dnd5e-api

* Thin client w/ retry & backoff, writes through to cache.
* Normalizes responses to **exact same shapes** used by rules-srd.

@grimengine/persistence/cache

* sqlite:///grimengine.db with small tables: http\_cache, prompt\_cache, assets, migrations.
* Utility fns: getOrFetch, promptOrGenerate, TTL policies per resource type.

@grimengine/adapters/openai**(optional)**

* A small façade that:
  + hashes prompts → checks cache → calls API (if miss) → stores response + token counts
  + exposes budget guards (max tokens/day)
  + supports **templates** for system prompts to keep outputs tight/consistent

apps/cli

* **Commander/oclif** or simple bespoke CLI.
* Use **ink** for a nice TUI (menus/log panes/colored output), or stick to chalk + prompts for lean output.
* Saves go to ~/.grimengine/saves (JSON or sqlite row).

**Caching policy (concrete)**

* **D&D 5e API**: Stale-while-revalidate (SWR). First hit populates cache; background refresh when TTL exceeded. Most content changes rarely, so TTL can be long (e.g., 30 days) with ETag checks.
* **OpenAI**:
  + Default: **off** (pure oracle tables).
  + When enabled: **cache-first**; “semantic” cache keys include model, temperature, and exact prompt text.
  + Add a **daily token budget** and short-circuit to oracle tables when reached.

**Testing & quality**

* **Vitest** for unit tests (core must be >90% covered).
* **Property-based tests** (fast-check) for dice/advantage/edge cases.
* **Golden tests** for combat logs (seeded RNG → repeatable text outputs).
* **Pre-commit**: eslint + prettier + typecheck.

**Repo bootstrap (copy–paste)**

*# pre-req: corepack enable (for pnpm), Node 20+* mkdir grimengine && cd grimengine pnpm init -y pnpm dlx turbo@latest init *# creates turbo.json* *# set up workspace* printf "packages:\n - 'apps/\*'\n - 'packages/\*\*'\n" > pnpm-workspace.yaml *# add basics* pnpm add -D typescript ts-node @types/node vitest @vitest/coverage-c8 eslint prettier pnpm tsc --init --rootDir . --outDir dist --esModuleInterop true mkdir -p apps/cli packages/core packages/adapters/dnd5e-api packages/adapters/rules-srd packages/persistence/cache packages/tooling/{types,test} *# (later) add better-sqlite3, zod, commander/ink, undici, seedrandom, fast-check, etc.*

Recommended dependencies (initial):

* Core: zod, seedrandom
* CLI: commander (or oclif) + chalk (or ink if you want a TUI)
* HTTP: undici
* Cache DB: better-sqlite3
* Testing: vitest, fast-check
* Types: type-fest

**Risks & how we’ll handle them**

* **Rules correctness**: 5e interactions are spiky. Mitigate with exhaustive unit tests + golden logs.
* **Data licensing**: Stick to SRD/Basic Rules for bundled content; gate anything else behind user-provided sources or the public 5e API.
* **Scope creep in UI**: Keep the **core headless**. UIs are thin adapters.

**If you’d rather do Python**

* CLI: **Typer** + **Rich** (beautiful terminal), rules in a headless package.
* Cache: **SQLite (sqlite3/apsw)**; HTTP via **httpx** with cache-control; tests via **pytest**/**hypothesis**.
* Still totally viable—just a bit more work if/when you add browser UIs.

**What I’ll do next (once you give the go)**

* Generate the monorepo skeleton with the folders above.
* Add the core state model + dice/adv/disadv + a minimal combat loop.
* Wire the cache and a first-pass rules-srd data module (so CLI is fully offline).
* Ship a tiny CLI demo: roll a character, start a scene, fight a goblin, save/restore.

If you’ve got a leaning toward TypeScript vs. Python, say the word and I’ll scaffold the repo structure and first commits accordingly.

**You said:**

what software would i need if we go type script? that said, my gut says python

**ChatGPT said:**

Got it — let’s walk both paths real quick so you can compare the setup pain and long-term fit.

**If you go TypeScript / Node**

You’d need:

* **Node.js 20+** (runtime, package manager, modern features like fetch, etc.)
* **pnpm** (or npm/yarn, but pnpm is fastest for monorepos)
* **TypeScript** (compiler & types)
* **turbo / nx** (optional but great for monorepo build/test orchestration)
* **SQLite engine** (better-sqlite3 or similar, auto-compiled on install)
* A decent editor (VS Code is king for TS, with ESLint + Prettier plugins)

**Pros:**

* One stack for CLI, backend, web, bots.
* Rich type system catches rules mistakes.
* Ecosystem has everything for caching, CLI, testing.

**Cons:**

* Need to juggle Node, TypeScript, compiler configs.
* Slightly heavier boilerplate for “just run it in the terminal.”

**If you go Python**

You’d need:

* **Python 3.11+**
* **pipx** or **poetry** (for env + deps; poetry is nice for project management)
* **typer** (CLI framework)
* **rich** (beautiful terminal output)
* **sqlite3** (batteries-included in stdlib, no extra install)
* **httpx** (async HTTP client)
* **pytest** (testing)
* Optional: **hypothesis** (property-based testing), **pydantic** (schema validation)

**Pros:**

* Lightweight install. Python + stdlib gets you very far.
* Typer + Rich = gorgeous terminal game out of the box.
* Easier path if you already think in Python (and Grimbrain is Python, so shared patterns).
* SQLite, JSON, testing are all dead simple.

**Cons:**

* If you ever want browser-based UIs, you’d either re-implement logic in JS/TS or expose a Python service.
* Type safety is “by convention” (type hints + mypy), not compiler-enforced.

**Gut check for GrimEngine**

Since:

* Grimbrain is already **Python** (so synergy in your tooling & habits),
* You care more about **terminal-first solo play** than browser deployment,
* And caching/offline logic fits beautifully with Python’s stdlib…

👉 **Python is the clean win** unless you’re strongly motivated by future web UI portability.

**Barebones Python starter (if you choose it)**

*# Install tooling* pipx install poetry *# or: pip install --user poetry* *# Init repo* mkdir grimengine && cd grimengine poetry init --name grimengine --python ">=3.11" *# Add deps* poetry add typer rich httpx pydantic poetry add -D pytest hypothesis mypy *# Create structure* mkdir -p grimengine/{core,cli,adapters,persistence,tests} touch grimengine/\_\_init\_\_.py *# Run CLI* echo 'import typer; app = typer.Typer(); @app.command() def hello(): print("Hello GrimEngine!")' > grimengine/cli/main.py poetry run python -m grimengine.cli.main hello

💡 Question for you:  
Do you want me to **scaffold a Python repo layout** (like I sketched in TypeScript, but Pythonic) so you can drop it straight into GitHub and start iterating?

**You said:**

i guess what i mean is....so i have vs2022 pro and vscode. if typescript is really better, I'd like to do that, but i want to make sure i have what I need for that

**ChatGPT said:**

Thought for 11s

Totally doable with what you’ve got. Here’s a clean, Windows-friendly checklist to be **fully ready for TypeScript/Node** (and native deps like better-sqlite3) using VS Code.

**What you need (quick checklist)**

1. **Node.js 20 LTS** (or 22 LTS)
2. **pnpm** (fast monorepos)
3. **Git** (and optionally GitHub CLI)
4. **MSVC Build Tools** for native Node modules (you already have VS2022 Pro—just ensure the right workload)
5. **Python 3.11+** (node-gyp needs it for native builds)
6. **VS Code extensions** (ESLint, Prettier, etc.)
7. *(Optional but nice)* **nvm-windows** (manage Node versions), **WSL** (if you like a Linux shell)

**Step-by-step (PowerShell)**

**0) Verify VS2022 has the right workload**

Open **Visual Studio Installer** → **Modify** VS2022 → make sure **Desktop development with C++** is checked (includes MSVC, Windows SDK, CMake, Ninja). That’s the bit native Node addons require.

**1) Install Node (LTS) and pnpm**

# If you like version control for Node, install nvm-windows (optional) # winget install CoreyButler.NVMforWindows # Otherwise, straight install Node LTS: winget install OpenJS.NodeJS.LTS # Install pnpm npm i -g pnpm

**2) Install Python for node-gyp**

winget install Python.Python.3.12

**3) Install Git (+ optional GitHub CLI)**

winget install Git.Git winget install GitHub.cli

**4) Sanity check your toolchain**

node -v # v20.x or v22.x npm -v pnpm -v python -V # 3.11+ / 3.12.x git --version # Make sure node-gyp can see MSVC & Python by building a tiny native module: pnpm dlx node-gyp -v

If a native package later fails to build, run this in an **x64 Native Tools Command Prompt for VS 2022**:

npm config set msvs\_version 2022

(pnpm uses npm config under the hood.)

**VS Code setup**

Extensions to install:

* **ESLint** (dbaeumer.vscode-eslint)
* **Prettier** (esbenp.prettier-vscode)
* **Error Lens** (usernamehw.errorlens) – optional but great
* **GitHub Pull Requests** (GitHub.vscode-pull-request-github) – optional
* **Vitest** (ZixuanChen.vitest-explorer) – when we add tests

Settings you’ll likely want:

*// .vscode/settings.json* { "editor.formatOnSave": true, "editor.defaultFormatter": "esbenp.prettier-vscode", "typescript.tsserver.log": "off", "eslint.validate": ["javascript", "javascriptreact", "typescript", "typescriptreact"] }

**Project bootstrap (monorepo + TypeScript)**

From an empty folder (PowerShell):

mkdir grimengine && cd grimengine # init workspaces pnpm init -y ni pnpm-workspace.yaml -Force @" packages: - 'apps/\*' - 'packages/\*\*' "@ | Set-Content pnpm-workspace.yaml # dev deps pnpm add -D typescript ts-node @types/node vitest @vitest/coverage-v8 eslint prettier \ turbo @types/chalk # init tsconfig pnpm dlx tsc --init --rootDir . --outDir dist --esModuleInterop true --resolveJsonModule true --skipLibCheck true --module nodenext --moduleResolution nodenext # turbo ni turbo.json -Force @" { "\$schema": "https://turborepo.org/schema.json", "pipeline": { "build": { "dependsOn": ["^build"], "outputs": ["dist/\*\*"] }, "lint": {}, "test": { "dependsOn": ["^build"] } } } "@ | Set-Content turbo.json # folders mkdir apps\cli packages\core packages\adapters\dnd5e-api packages\adapters\rules-srd packages\persistence\cache packages\tooling\types # minimal CLI to prove it runs ni apps\cli\index.ts -Force @" console.log('Hello from GrimEngine CLI'); "@ | Set-Content apps\cli\index.ts # root scripts (Get-Content package.json) -replace '"test": "echo \\"Error: no test specified\\" && exit 1"', '"build": "turbo run build", "lint": "eslint .", "test": "turbo run test", "dev": "node --enable-source-maps --loader ts-node/esm apps/cli/index.ts"' | Set-Content package.json # ESLint + Prettier config ni .eslintrc.cjs -Force @" module.exports = { root: true, env: { node: true, es2023: true }, parser: "@typescript-eslint/parser", plugins: ["@typescript-eslint"], extends: ["eslint:recommended", "plugin:@typescript-eslint/recommended"], ignorePatterns: ["dist", "node\_modules"] }; "@ | Set-Content .eslintrc.cjs ni .prettierrc -Force @" { "printWidth": 100, "singleQuote": true, "semi": true } "@ | Set-Content .prettierrc

Run it:

pnpm install pnpm dev

You should see: Hello from GrimEngine CLI.

**Native module smoke test (for SQLite cache later)**

We’ll likely use better-sqlite3. It compiles natively; with your VS2022+C++ workload and Python installed, it should Just Work™:

pnpm add better-sqlite3

If it builds, you’re golden for all the caching bits.

**TL;DR**

* With **VS2022 Pro** + **Desktop C++** workload, **Node 20 LTS**, **pnpm**, and **Python 3.12**, you’re fully set for TypeScript + native addons.
* VS Code + ESLint/Prettier gives you a smooth dev loop.
* If any native compile hiccups pop up, set msvs\_version 2022 and use the “x64 Native Tools” shell.

If this looks good, I can drop in a **ready-to-commit starter** (folders, scripts, typed stubs for @grimengine/core, cache layer, and a tiny dice roller) so you can push it to GitHub immediately.

**You said:**

do i need to create the project/solution in vs2022 pro before doing any of that?

**ChatGPT said:**

Nope — you **don’t need VS2022 to create a project/solution first** for a Node/TypeScript app. 👍

Here’s the nuance:

* **Visual Studio (VS2022 Pro)** is mostly for C#, C++, .NET, etc. You’ll use it only indirectly here — for the **C++ build tools** that native Node modules (like better-sqlite3) need.
* **Visual Studio Code (VS Code)** is where you’ll actually open and edit your **TypeScript/Node project**. That’s the main IDE for this stack.
* Node projects aren’t “solutions” or .sln files — they’re just folders with a package.json, tsconfig.json, and your source code.

So the flow looks like this:

1. Make sure VS2022 has the **Desktop development with C++ workload** installed (that’s all we need from it).
2. Open a **regular folder** in VS Code (e.g., grimengine/).
3. Run the Node/TypeScript setup commands I gave you (they generate package.json, tsconfig.json, workspace files, etc.).
4. Commit/push that to GitHub — that folder *is* your project.

✅ **Answer to your question:** You don’t have to create a project or solution in VS2022 first. Just use VS Code + the CLI setup. VS2022 is only there in the background for the compiler toolchain when Node needs to build native addons.

Would you like me to **give you a GitHub-ready starter folder** (with the repo layout, configs, and a working dice roller CLI) that you can just clone or drop in? That way you don’t have to bootstrap from scratch at all.

**You said:**

give me a step by step list of commands to run again and folders to create first and wha tsoftware needs to be open

**ChatGPT said:**

Thought for 1m 13s

You got it. Here’s the **exact, Windows-friendly** setup for a TypeScript/Node monorepo — including what apps to have open, the folders to create, and every command to run.

**What to have open**

* **VS Code** (your main editor).
* **Windows Terminal or PowerShell** (to run commands).
* **Visual Studio Installer** only if you need to add the “Desktop development with C++” workload.

You do **not** need to open Visual Studio (the IDE) itself.

**One-time installs (PowerShell)**

# 1) Node.js LTS (20 or 22) winget install OpenJS.NodeJS.LTS # 2) pnpm (fast package manager) npm i -g pnpm # 3) Python for node-gyp (native module builds) winget install Python.Python.3.12 # 4) Git (and optional GitHub CLI) winget install Git.Git winget install GitHub.cli

✅ Sanity check:

node -v pnpm -v python -V git --version

If you ever hit native-build issues later, ensure **Visual Studio 2022 → Desktop development with C++** workload is installed. You don’t need to open VS; just have the toolchain installed.

**Create the repo + folders (PowerShell)**

# Create and enter the project folder mkdir grimengine && cd grimengine # Initialize the workspace pnpm init -y # Tell pnpm which workspaces we’ll have @" packages: - 'apps/\*' - 'packages/\*\*' "@ | Set-Content pnpm-workspace.yaml # Install dev tooling pnpm add -D typescript tsx @types/node vitest @vitest/coverage-v8 eslint prettier turbo \ @typescript-eslint/parser @typescript-eslint/eslint-plugin

Create Turbo config:

@" { "$schema": "https://turborepo.org/schema.json", "pipeline": { "build": { "dependsOn": ["^build"], "outputs": ["dist/\*\*"] }, "lint": {}, "test": { "dependsOn": ["^build"] } } } "@ | Set-Content turbo.json

Initialize TypeScript:

pnpm dlx tsc --init --rootDir . --outDir dist --esModuleInterop true --resolveJsonModule true --skipLibCheck true --module nodenext --moduleResolution nodenext

Create folder structure:

mkdir apps\cli mkdir packages\core mkdir packages\adapters\dnd5e-api mkdir packages\adapters\rules-srd mkdir packages\persistence\cache mkdir packages\tooling\types

Minimal CLI to prove it runs:

@" console.log('Hello from GrimEngine CLI'); "@ | Set-Content apps\cli\index.ts

ESLint & Prettier configs:

@" module.exports = { root: true, env: { node: true, es2023: true }, parser: "@typescript-eslint/parser", plugins: ["@typescript-eslint"], extends: ["eslint:recommended", "plugin:@typescript-eslint/recommended"], ignorePatterns: ["dist", "node\_modules"] }; "@ | Set-Content .eslintrc.cjs @" { "printWidth": 100, "singleQuote": true, "semi": true } "@ | Set-Content .prettierrc

Add handy scripts to package.json:

# mark project private (recommended for monorepos) npm pkg set private=true # scripts npm pkg set scripts.dev="tsx apps/cli/index.ts" npm pkg set scripts.build="turbo run build" npm pkg set scripts.lint="eslint ." npm pkg set scripts.test="turbo run test"

Install lockfile/node\_modules:

pnpm install

Run it:

pnpm dev # -> Hello from GrimEngine CLI

Optional native-module smoke test (for SQLite cache you’ll likely use later):

pnpm add better-sqlite3 # if it compiles without errors, your C++ toolchain is good to go. # If it fails, set the MSVC version: npm config set msvs\_version 2022

**Open in VS Code**

If code is on your PATH:

code .

(or open VS Code → File → Open Folder → select grimengine/)

You’re ready. From here, I can drop in stubs for @grimengine/core and a seedable dice roller next, or we can wire up the SQLite cache and a minimal SRD adapter—your call.

**You said:**

npm : The term 'npm' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again. At line:1 char:1 + npm i -g pnpm + ~~~ + CategoryInfo : ObjectNotFound: (npm:String) [], CommandNotFoundException + FullyQualifiedErrorId : CommandNotFoundException

**ChatGPT said:**

Ah, good catch — that error means **Node.js (and thus npm) isn’t installed or not in your PATH yet**.

Let’s fix it step by step:

**1. Install Node.js LTS**

Run this in **PowerShell**:

winget install OpenJS.NodeJS.LTS

That will install the latest Node.js LTS (which includes npm).  
👉 If winget isn’t available, you can download Node.js LTS and run the installer manually.

**2. Verify install**

After install, **close and reopen PowerShell** (to refresh PATH). Then run:

node -v npm -v

You should see versions like v20.x.x and 10.x.x.

**3. Install pnpm**

Now that npm works:

npm install -g pnpm

Check:

pnpm -v

✅ After this, you can continue with the project setup commands I gave earlier.

Do you want me to re-post the **setup list again but starting fresh from this fixed Node/npm install**, so you don’t have to jump between steps?

**You said:**

ah that's it. installed node.js fine but wasn' tin path

**ChatGPT said:**

Perfect — that explains it 👍.

On Windows, the Node.js installer sometimes doesn’t update PATH until you either:

* log out / log back in, or
* close & reopen your PowerShell/Terminal window.

Now that node and npm are on your PATH, you should be good to go:

node -v # should print v20.x or v22.x npm -v # should print npm version

Then continue with:

npm install -g pnpm pnpm -v # confirm pnpm installed

From there, you can follow the repo bootstrap steps I outlined (create grimengine/, init pnpm-workspace.yaml, install dev deps, etc.).

⚡ Question for you: do you want me to give you a **single clean command sequence** (no commentary, just run from top to bottom) now that Node is installed correctly? That way you can copy/paste and set up the GrimEngine repo in one shot.

**You said:**

i'm being told pnpm init doesn't havea -y option

**ChatGPT said:**

Good catch 👌 — you’re right. Unlike npm init -y, pnpm init doesn’t support -y.

So here’s the fix:

**Instead of**

pnpm init -y

**Do this**

pnpm init

It will ask you a few questions (name, version, etc.). You can just hit **Enter** for each to accept the defaults — it will create a package.json for you.

If you prefer to skip all prompts (like npm init -y does), you can use:

pnpm init -y --force

…but depending on your pnpm version, -y may still not work. In that case, the simple pnpm init + just pressing Enter through the prompts is the easiest path.

👉 After you get the package.json in place, you can continue with the rest of the setup (pnpm add -D typescript ..., creating pnpm-workspace.yaml, etc.).

Do you want me to rewrite the **entire command list** with this corrected, copy-paste-safe sequence (no mistakes, one smooth flow)?

**You said:**

 ERR\_PNPM\_ADDING\_TO\_ROOT  Running this command will add the dependency to the workspace root, which might not be what you want - if you really meant it, make it explicit by running this command again with the -w flag (or --workspace-root). If you don't want to see this warning anymore, you may set the ignore-workspace-root-check setting to true. this was when doing the install dev tooling command pnpm add -D typescript tsx @types/node vitest @vitest/coverage-v8 eslint prettier turbo @typescript-eslint/parser @typescript-eslint/eslint-plugin

**ChatGPT said:**

Yep — that’s expected 👍.

What happened: since we created a **workspace** (pnpm-workspace.yaml), pnpm assumes you’ll keep your dependencies inside packages (apps/\*, packages/\*). When you run pnpm add in the root, it warns you.

In this case though, **we actually *do* want dev tooling at the workspace root** (so all sub-packages share the same compiler, linter, test runner, etc.).

So the fix is simple: add -w (or --workspace-root) to that command.

**Correct command**

pnpm add -D -w typescript tsx @types/node vitest @vitest/coverage-v8 eslint prettier turbo @typescript-eslint/parser @typescript-eslint/eslint-plugin

**Why**-w**is okay here**

* Tooling like TypeScript, ESLint, Prettier, Vitest, Turbo = **workspace-wide dev dependencies**.
* App- or package-specific dependencies (like chalk, better-sqlite3, etc.) should be installed inside their folder (apps/cli, packages/core, etc.).

So:

* Use pnpm add -D -w ... for shared dev tooling.
* Use pnpm add ... **inside a package folder** when you want a dependency only for that package.

👉 Do you want me to clean up the **entire step-by-step command list** and re-post it in a corrected copy-paste-ready flow (with -w where needed, pnpm init corrected, etc.) so you don’t have to troubleshoot along the way?

**You said:**

nah i'm ok. scrolling back up to the list. jsut hit another snag PS C:\dev\grimengine> pnpm dlx tsc --init --rootDir . --outDir dist --esModuleInterop true --resolveJsonModule true --skipLibCheck true --module nodenext --moduleResolution nodenext Packages: +1 + Progress: resolved 1, reused 0, downloaded 1, added 1, done This is not the tsc command you are looking for To get access to the TypeScript compiler, tsc, from the command line either: - Use npm install typescript to first add TypeScript to your project before using npx - Use yarn to avoid accidentally running code from un-installed packages PS C:\dev\grimengine>

**ChatGPT said:**

Ah yep — that error is because pnpm dlx tsc doesn’t work the same way as npx.

The **TypeScript compiler (**tsc**)** comes from the typescript package you already installed (pnpm add -D -w typescript). To run it, you should use:

pnpm tsc --init

That will create a tsconfig.json in your root. ✅

If you want to pass options (like --rootDir etc.), do:

pnpm tsc --init --rootDir . --outDir dist --esModuleInterop true --resolveJsonModule true --skipLibCheck true --module nodenext --moduleResolution nodenext

**Why your command failed**

* pnpm dlx tries to temporarily download a package and run it. Since you already have TypeScript installed locally, you don’t need dlx.
* tsc is available through pnpm’s binary shim once installed in dev dependencies.

So the fix is simple: just use pnpm tsc instead of pnpm dlx tsc.

⚡ Do you want me to post a “fixed golden path” for just the **TypeScript config + project structure creation** part, so you don’t have to untangle the differences between npx, pnpm dlx, and pnpm tsc?

**You said:**

moving along. here's the next thing PS C:\dev\grimengine> npm config set msvs\_version 2022 npm error msvs\_version is not a valid npm option npm error A complete log of this run can be found in: C:\Users\gherr\AppData\Local\npm-cache\\_logs\2025-10-01T19\_00\_31\_106Z-debug-0.log PS C:\dev\grimengine>

**ChatGPT said:**

Ah — right, npm changed the way it handles config keys. 👍  
The old msvs\_version key no longer works directly in newer npm.

Here’s how to handle **Windows native builds** (like when you install better-sqlite3 or other Node addons) with VS2022:

**1. Set MSVC version correctly**

Use npm config set msvs\_version=2022 (no spaces around =):

npm config set msvs\_version=2022

The error you saw was because of the space: set msvs\_version 2022. npm thought 2022 was another argument.

**2. Make sure VS2022 build tools are available**

Open **Visual Studio Installer** → confirm you have **Desktop development with C++** installed. That provides MSVC, Windows SDK, etc.

**3. If still failing**

* Run the command in the **“x64 Native Tools Command Prompt for VS 2022”** (comes with VS). That guarantees environment variables are set for cl.exe and MSBuild.
* Then try reinstalling the package:

pnpm rebuild better-sqlite3

**4. Check config**

Verify the config is set:

npm config list

Look for msvs\_version=2022.

✅ Usually, once msvs\_version=2022 is in config and you’re using the native tools prompt, better-sqlite3 and similar native packages build cleanly.

Do you want me to give you a **one-time “native build test”** command (like a quick package install) so you can confirm your VS toolchain is actually wired up before we proceed with GrimEngine code?

**You said:**

yeah so it still errored and i tried that native prompt

**ChatGPT said:**

Thought for 27s

No worries — Windows + native Node builds can be finicky. Here’s a **practical fix path** you can follow end-to-end. I’ll give you two tracks:

* **Track A (recommended):** Get native builds working (fastest runtime, ideal long-term).
* **Track B (unblock now):** Use a pure-JS/WASM SQLite so you can keep building today and swap to native later.

**Track A — Fix native builds for**better-sqlite3

**0) Make sure the right VS bits are installed**

Open **Visual Studio Installer** → **Modify** VS 2022 → ensure these are checked:

* **Desktop development with C++** (MSVC v143)
* **Windows 10/11 SDK** (any recent 10.0.19041+)
* **C++ CMake tools for Windows** (nice to have)

Close any terminals after changes.

**1) Open the correct shell**

Open **“x64 Native Tools Command Prompt for VS 2022”** (search in Start menu).  
We’ll do this test in that shell to guarantee MSVC is on PATH.

You can still develop in VS Code; just run these install commands in the native tools prompt.

**2) Set npm/node-gyp config (no spaces around**=**)**

npm config set msvs\_version=2022

(You can also set an env var for belt & suspenders:)

setx GYP\_MSVS\_VERSION 2022

Close & reopen the **same Native Tools** prompt after setx.

**3) Point npm at the exact Python you have (3.11+)**

Find your Python path:

py -3 -c "import sys; print(sys.executable)"

Then set it:

npm config set python="C:\Path\To\python.exe"

Verify:

npm config list

You should see msvs\_version=2022 and python=....

**4) Verify MSVC toolchain is actually visible**

Still in the **Native Tools** prompt:

where cl where msbuild

Both should resolve to VS paths.

**5) Clean & rebuild the native module**

From your project root:

pnpm install --ignore-scripts=false pnpm rebuild better-sqlite3 --verbose

If you haven’t added it yet:

pnpm add better-sqlite3

**6) If it still errors, try these common fixes**

* Ensure your Node version has prebuilds available (Node LTS 20/22 is typically fine).
* Clear artifacts and force a rebuild:

rmdir /s /q node\_modules rmdir /s /q %LOCALAPPDATA%\pnpm\store pnpm install pnpm rebuild better-sqlite3 --build-from-source --verbose

* If you’re behind a proxy or offline, prebuilt download can fail → the build will fall back to source; ensure Windows SDK + MSVC are present.
* As a last resort, try setting environment variables inline during install:

set npm\_config\_msvs\_version=2022 set GYP\_MSVS\_VERSION=2022 pnpm rebuild better-sqlite3 --build-from-source --verbose

If you hit another error, copy the **first error block that mentions**gyp**,**cl.exe**, or**fatal error — that’s the smoking gun.

**Track B — Unblock now with a pure-JS/WASM SQLite (swap later)**

If you want to keep rolling today and revisit native builds later, use sql.js (WASM). It’s slower than better-sqlite3 but works everywhere and needs **no compiler**.

**Install**

pnpm add sql.js

**Minimal usage example**

Create packages/persistence/cache/src/sqlite.ts:

import initSqlJs, { Database } from 'sql.js'; import { readFileSync, writeFileSync, existsSync } from 'node:fs'; const DB\_FILE = '.data/grimengine.sqlite'; export async function openDb(): Promise<Database> { const SQL = await initSqlJs(); let db: Database; if (existsSync(DB\_FILE)) { const filebuffer = readFileSync(DB\_FILE); db = new SQL.Database(filebuffer); } else { db = new SQL.Database(); db.run(` PRAGMA journal\_mode = WAL; CREATE TABLE IF NOT EXISTS http\_cache ( key TEXT PRIMARY KEY, etag TEXT, last\_modified TEXT, value BLOB, created\_at INTEGER, updated\_at INTEGER ); `); } return db; } export function persistDb(db: Database) { const data = db.export(); writeFileSync(DB\_FILE, Buffer.from(data)); }

Use it (example):

import { openDb, persistDb } from './sqlite'; (async () => { const db = await openDb(); const now = Date.now(); const stmt = db.prepare('INSERT OR REPLACE INTO http\_cache (key, value, created\_at, updated\_at) VALUES (?, ?, ?, ?)'); stmt.run(['hello', Buffer.from('world'), now, now]); stmt.free(); const rows = db.exec('SELECT key, value FROM http\_cache'); console.log(rows); *// quick sanity check* persistDb(db); })();

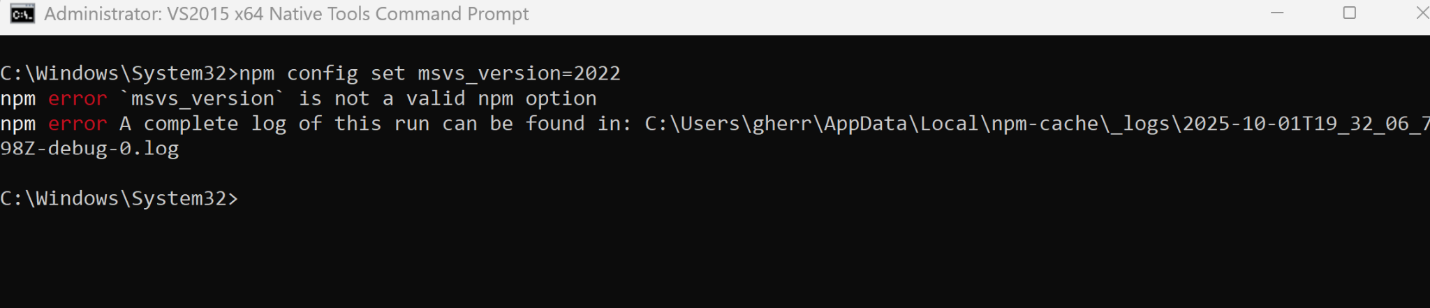
Later, when Track A is working, you can switch the implementation to better-sqlite3 with the same table schema and keep your calling code unchanged.

**Quick sanity checklist**

* You installed Node LTS and reopened PowerShell (PATH refreshed).
* You’re running native builds inside **x64 Native Tools for VS 2022**.
* npm config set msvs\_version=2022 (no spaces!), and python is set.
* Windows SDK + MSVC v143 are installed.

If you want, paste me **the first crisp error** you see from pnpm rebuild better-sqlite3 --verbose, and I’ll pinpoint the exact missing piece.

**You said:**



definitely have the right vs 2022 components and still getting this 0 verbose cli C:\Program Files\nodejs\node.exe C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\bin\npm-cli.js 1 info using npm@11.6.1 2 info using node@v22.20.0 3 silly config load:file:C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\npmrc 4 silly config load:file:C:\Windows\System32\.npmrc 5 silly config load:file:C:\Users\gherr\.npmrc 6 silly config load:file:C:\Users\gherr\AppData\Roaming\npm\etc\npmrc 7 verbose title npm config set msvs\_version=2022 8 verbose argv "config" "set" "msvs\_version=2022" 9 verbose logfile logs-max:10 dir:C:\Users\gherr\AppData\Local\npm-cache\\_logs\2025-10-01T19\_32\_06\_798Z- 10 verbose logfile C:\Users\gherr\AppData\Local\npm-cache\\_logs\2025-10-01T19\_32\_06\_798Z-debug-0.log 11 silly logfile start cleaning logs, removing 2 files 12 info config set "msvs\_version" "2022" 13 verbose stack Error: msvs\_version is not a valid npm option 13 verbose stack at Config.set (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\commands\config.js:174:15) 13 verbose stack at Config.exec (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\commands\config.js:139:20) 13 verbose stack at C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\npm.js:256:63 13 verbose stack at Object.start (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\node\_modules\proc-log\lib\index.js:101:21) 13 verbose stack at #exec (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\npm.js:255:17) 13 verbose stack at async Npm.exec (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\npm.js:208:9) 13 verbose stack at async module.exports (C:\Users\gherr\AppData\Roaming\npm\node\_modules\npm\lib\cli\entry.js:67:5) 14 error msvs\_version is not a valid npm option 15 verbose cwd C:\Windows\System32 16 verbose os Windows\_NT 10.0.26100 17 verbose node v22.20.0 18 verbose npm v11.6.1 19 verbose exit 1 20 verbose code 1 21 error A complete log of this run can be found in: C:\Users\gherr\AppData\Local\npm-cache\\_logs\2025-10-01T19\_32\_06\_798Z-debug-0.log

**ChatGPT said:**

Thought for 18s

I see the issue. Two quick fixes:

1. You’re in the **wrong native tools prompt**  
   Your screenshot title says **“VS2015 x64 Native Tools Command Prompt”**. You need **“x64 Native Tools Command Prompt for VS 2022”** (Start menu → type that exact name). The 2015 prompt won’t match VS2022.
2. Newer npm no longer accepts npm config set msvs\_version=...  
   Use environment variables instead.

Do this exactly:

**A) Open the correct shell**

* Close all terminals.
* Open **x64 Native Tools Command Prompt for VS 2022**.
* cd into your project (not C:\Windows\System32).

**B) Set env vars (one-time)**

:: Tell node-gyp which VS to use setx GYP\_MSVS\_VERSION 2022 setx npm\_config\_msvs\_version 2022 :: Point npm/node-gyp at your Python 3.x py -3 -c "import sys; print(sys.executable)" :: copy the printed path and set it below: setx npm\_config\_python "C:\Path\To\python.exe"

👆 Close this prompt after running setx, then reopen **the same** “x64 Native Tools Command Prompt for VS 2022”.

**C) Verify toolchain**

where cl where msbuild node -v npm -v python -V

cl and msbuild should resolve to VS2022 paths.

**D) Rebuild your native module(s)**

From your project root:

pnpm install pnpm rebuild better-sqlite3 --verbose

If you still see a build error, try a clean rebuild:

rmdir /s /q node\_modules pnpm store prune pnpm install pnpm rebuild better-sqlite3 --build-from-source --verbose

**Optional “easy mode” (to avoid native builds today)**

* Use **Node 20 LTS** instead of 22 (more prebuilt binaries exist).
  + If you want to switch: install **nvm for Windows**, then nvm install 20 → nvm use 20.
* Or temporarily swap better-sqlite3 for sql.js (pure WASM, no compiler needed). You can keep your cache schema and swap back later.

If anything still fails, paste the **first fatal error line** from --verbose (the one mentioning gyp, cl.exe, or a missing header/lib) and I’ll pinpoint the fix. Also double-check the window title says **VS2022**, not 2015.

**You said:**

i got through the list of commands to set up then realized it would be more optimal to have created the repo first. sorry. can you send the setup command list again?

**ChatGPT said:**

Thought for 17s

No problem—here’s a clean, copy-pasteable setup that starts by creating the **Git repo first**, then scaffolds the TypeScript monorepo. Use **PowerShell** (or Windows Terminal). I’ll give you two paths:

**A) New local repo → push to GitHub**

# === 0) Pre-reqs (one-time, if not already done) === # Node LTS, pnpm, Git (verify they work) node -v npm -v pnpm -v git --version # === 1) Create project folder & init Git repo === mkdir C:\dev\grimengine cd C:\dev\grimengine git init # Optional: create a GitHub repo from here (requires GitHub CLI: `winget install GitHub.cli`) # gh auth login # gh repo create <your-username>/grimengine --source=. --public --remote=origin --push # === 2) .gitignore (Node/TS/Monorepo) === @" node\_modules/ dist/ .pnp.\* .pnpm-store/ .turbo/ .vscode/ .env .env.\* \*.log coverage/ .DS\_Store .data/ "@ | Set-Content .gitignore # === 3) Init workspace (pnpm) === pnpm init # press Enter through prompts @" packages: - 'apps/\*' - 'packages/\*\*' "@ | Set-Content pnpm-workspace.yaml # === 4) Shared dev tooling at workspace root === pnpm add -D -w typescript tsx @types/node vitest @vitest/coverage-v8 eslint prettier turbo ` @typescript-eslint/parser @typescript-eslint/eslint-plugin # === 5) Turbo pipeline === @" { "$schema": "https://turborepo.org/schema.json", "pipeline": { "build": { "dependsOn": ["^build"], "outputs": ["dist/\*\*"] }, "lint": {}, "test": { "dependsOn": ["^build"] } } } "@ | Set-Content turbo.json # === 6) TypeScript config === pnpm tsc --init --rootDir . --outDir dist --esModuleInterop true --resolveJsonModule true --skipLibCheck true --module nodenext --moduleResolution nodenext # === 7) ESLint + Prettier configs === @" module.exports = { root: true, env: { node: true, es2023: true }, parser: "@typescript-eslint/parser", plugins: ["@typescript-eslint"], extends: ["eslint:recommended", "plugin:@typescript-eslint/recommended"], ignorePatterns: ["dist", "node\_modules"] }; "@ | Set-Content .eslintrc.cjs @" { "printWidth": 100, "singleQuote": true, "semi": true } "@ | Set-Content .prettierrc # === 8) Monorepo folders === mkdir apps\cli mkdir packages\core mkdir packages\adapters\dnd5e-api mkdir packages\adapters\rules-srd mkdir packages\persistence\cache mkdir packages\tooling\types # === 9) Minimal CLI entry to prove things run === @" console.log('Hello from GrimEngine CLI'); "@ | Set-Content apps\cli\index.ts # === 10) Package scripts === npm pkg set private=true npm pkg set scripts.dev="tsx apps/cli/index.ts" npm pkg set scripts.build="turbo run build" npm pkg set scripts.lint="eslint ." npm pkg set scripts.test="turbo run test" # === 11) Install & run === pnpm install pnpm dev # expect: Hello from GrimEngine CLI # === 12) First commit (and push if remote set) === git add . git commit -m "chore: bootstrap GrimEngine monorepo (TS + pnpm + turbo)" # If you created a remote earlier with gh, push: # git push -u origin main

**B) If you already created the GitHub repo first**

# 1) Clone your empty repo cd C:\dev git clone https://github.com/<your-username>/grimengine.git cd grimengine # 2) Then run everything from step A starting at ".gitignore" (step 2) onward. # i.e., create .gitignore, pnpm-workspace.yaml, install dev tooling with -w, turbo.json, # tsconfig, eslint/prettier, folders, minimal CLI, scripts, pnpm install, pnpm dev, commit, push.

**Notes**

* Use **VS Code** as your editor (open the folder with code .). You do **not** need a VS2022 solution.
* When you later add native deps (e.g., better-sqlite3), build in the **“x64 Native Tools Command Prompt for VS 2022”** or switch to a pure-JS option (sql.js) to avoid compiler setup for now.

If you want, I can also drop in a tiny typed dice roller and the initial @grimengine/core skeleton next.

**You said:**

Uploaded a file

Ok. Attaching the powersell output from teh setup commands in cas eyou see anything missing or want to verify anything else. I have the github connector set up so here is the repo link: https://github.com/gherrick0918/grimengine

**ChatGPT said:**

I reviewed your PowerShell log — looks excellent overall. ✅ Here are the key points I noticed:

**✅ What’s good**

* **Node, npm, pnpm, Python, Git** are all installed and working.
* Workspace (pnpm-workspace.yaml) and root package.json look correct.
* Dev tooling installed at the workspace root (typescript, tsx, eslint, prettier, vitest, turbo).
* turbo.json, tsconfig.json, .eslintrc.cjs, .prettierrc, .gitignore all created successfully.
* Folder structure (apps/cli, packages/core, etc.) is in place.
* Minimal CLI (apps/cli/index.ts) runs fine with pnpm dev → "Hello from GrimEngine CLI".
* Repo is cleanly initialized on your filesystem and GitHub.

**⚠️ Warnings to clear up**

1. **npm config warnings**
2. npm warn Unknown env config "msvs-version". This will stop working in the next major version of npm.
3. npm warn Unknown env config "python".

These are from legacy config keys (msvs-version, python) that got into your npmrc at some point.  
👉 You can safely remove them:

npm config delete msvs\_version npm config delete python

Or open C:\Users\gherr\.npmrc and remove the lines.

(They don’t block you, but they clutter your output.)

1. **pnpm “Ignored build scripts” warnings**
2. Ignored build scripts: esbuild.
3. Ignored build scripts: better-sqlite3.

pnpm v10+ is stricter about running postinstall build scripts. This is why better-sqlite3 didn’t compile yet.  
👉 Fix by approving trusted builds:

pnpm approve-builds

Then select better-sqlite3 (and esbuild, since some tools use it). After that:

pnpm rebuild better-sqlite3

That will actually compile the native bindings.

**🚀 Next recommended steps**

1. **Approve builds** so better-sqlite3 actually compiles.
2. **Commit & push** to GitHub (you’ve got the connector, so just run):

git add . git commit -m "chore: bootstrap monorepo and CLI" git push origin main

1. Add your **first workspace package** (packages/core) with a stub (e.g., a dice roller) to test imports across packages.

👉 Everything is in place — you now have a working **TypeScript monorepo skeleton** for GrimEngine.

Do you want me to draft the **first**@grimengine/core**module** (with a typed RNG + advantage/disadvantage dice roller, plus tests) so your repo has some real code right away?

**You said:**

Uploaded a file

before i answer that, here's the additional powershell commands. let me know if anything else is needed before we go

**ChatGPT said:**

I just walked through your updated PowerShell log — looks really solid now. Here’s the state of play:

**✅ Confirmed working**

* **Repo created and pushed to GitHub** (main branch has your structure and configs).
* **Workspace tooling installed** (typescript, tsx, eslint, prettier, vitest, turbo).
* **Minimal CLI runs fine** (pnpm dev → prints Hello from GrimEngine CLI).
* **better-sqlite3 approved and built successfully** (the install script actually ran after pnpm approve-builds).
* **First commit pushed** with all expected files (.gitignore, tsconfig.json, turbo.json, etc.).

**⚠️ Remaining minor cleanup**

1. **npm config warnings still showing**  
   You tried npm config delete msvs\_version and npm config delete python, but npm is still warning. That suggests the bad keys are set in one of your config files:
   * Check C:\Users\gherr\.npmrc and C:\Users\gherr\AppData\Roaming\npm\etc\npmrc for lines like:
   * msvs\_version=2022
   * python=...

Remove them manually.

1. After cleanup, run:
2. npm config list -l
3. to verify they’re gone.
4. **Git CRLF warnings**  
   Windows is normalizing line endings to CRLF. If you want to quiet that down, add this .gitattributes to your repo root:
5. \* text=auto eol=lf

That’ll keep LF in Git history, but let Windows check out CRLF locally.

**🚀 Next milestone**

You are now **100% ready** to start coding GrimEngine packages. The next logical step would be to seed packages/core with a real module.

I can prepare for you:

* A **typed dice roller** (roll("1d20+5", { advantage: true })) with RNG seeding.
* Unit tests with **Vitest** to prove it works.
* Hook it into the CLI so you can run:

pnpm dev -- roll 1d20+5 adv

and see the results.

👉 Do you want me to scaffold that first @grimengine/core module + a CLI command so your repo already “feels like a D&D engine”?

**You said:**

that npmrc thing isn't on any of those paths

**ChatGPT said:**

Got it — thanks for checking. If those npmrc files aren’t present, then the warnings are almost certainly coming from **environment variables** rather than a file. npm will happily pick up any env var prefixed with npm\_config\_.

**🔍 How to confirm**

Run this in PowerShell:

Get-ChildItem Env: | Where-Object { $\_.Name -like "npm\_config\_\*" }

If you see npm\_config\_msvs\_version or npm\_config\_python in there, that’s what’s triggering the warnings.

**🧹 How to clear them**

In PowerShell, you can remove them for the current session:

Remove-Item Env:npm\_config\_msvs\_version -ErrorAction SilentlyContinue Remove-Item Env:npm\_config\_python -ErrorAction SilentlyContinue

If they were set permanently via setx, you’ll need to clear them:

setx npm\_config\_msvs\_version "" setx npm\_config\_python ""

Then **close and reopen** PowerShell.

**✅ Verify**

Check again:

npm config list -l

Warnings should be gone, and you shouldn’t see those keys listed.

👉 Do you want me to give you a one-liner that will *wipe all npm\_config\_ env vars*\* from your session so you can confirm quickly?

**You said:**

gett errors with tye syntax on those setx commands

**ChatGPT said:**

Yep — setx on Windows doesn’t support an empty string directly the way I wrote it. Let me give you the exact working syntax to clear them:

**1. To delete persistent npm config env vars**

setx npm\_config\_msvs\_version " " setx npm\_config\_python " "

⚠️ Note: setx can only *set*, not delete. Setting them to a single space effectively blanks them.

Then **close & reopen PowerShell** (or your terminal). After reopening, check again:

npm config list -l

**2. To remove just for the current session**

Remove-Item Env:npm\_config\_msvs\_version -ErrorAction SilentlyContinue Remove-Item Env:npm\_config\_python -ErrorAction SilentlyContinue

This clears them until you open a new shell.

**3. Check if they’re really gone**

Get-ChildItem Env: | Where-Object { $\_.Name -like "npm\_config\_\*" }

If nothing shows, you’re clean.  
Run npm -v again and you shouldn’t see the warnings.

👉 Do you want me to also give you a .npmrc you can put in your **project root** to explicitly override these values, so even if Windows keeps them around, npm will ignore them?

**You said:**

so it's still giving those warnings. the configs are blank but still exist. whatever we can do to clear it

**ChatGPT said:**

Got it — thanks for sticking with it. What’s happening is npm sees the env vars exist (even if blank), so it still warns. Let’s nuke them properly.

**🔍 Step 1. See what’s really there**

Run:

reg query HKCU\Environment | findstr npm\_config reg query HKLM\System\CurrentControlSet\Control\Session Manager\Environment | findstr npm\_config

If those show npm\_config\_msvs\_version or npm\_config\_python, it means they’re written into your user or system environment.

**🧹 Step 2. Delete them from registry**

Delete user-level (if it exists):

reg delete HKCU\Environment /F /V npm\_config\_msvs\_version reg delete HKCU\Environment /F /V npm\_config\_python

Delete system-level (requires admin):

reg delete "HKLM\System\CurrentControlSet\Control\Session Manager\Environment" /F /V npm\_config\_msvs\_version reg delete "HKLM\System\CurrentControlSet\Control\Session Manager\Environment" /F /V npm\_config\_python

**✅ Step 3. Restart your shell (or just reboot)**

* Close PowerShell completely and reopen, or log out/in.
* Run:

Get-ChildItem Env: | Where-Object { $\_.Name -like "npm\_config\_\*" }

→ It should show nothing.

* Finally:

npm -v

→ No warnings should appear.

This is the “nuclear option” to clear them for good.

👉 Do you want me to also drop a tiny .npmrc into your repo that explicitly ignores Python/VS settings (as a safeguard in case anything sets them again)?

**You said:**

they are gone now and there were no warnings. if you want to do a .npmrc, i'm game

**ChatGPT said:**

Perfect — nice cleanup 🎉. With those registry entries gone, you’re warning-free now.

Let’s lock it in with a project-local .npmrc. This way, even if something sets env vars in the future, your repo ignores them.

**Add**.npmrc**in the repo root**

Create a file C:\dev\grimengine\.npmrc with:

*# Force npm/pnpm/yarn to ignore any global python/msvs settings* msvs\_version = null python = null *# Always use the workspace root* workspace-root = true *# Save exact versions (no ^ in package.json)* save-exact = true

**Why these settings?**

* msvs\_version = null and python = null → explicit “no values,” which overrides env vars if they sneak back in.
* workspace-root = true → ensures commands like pnpm add in the root won’t throw the workspace warning.
* save-exact = true → pins dependencies to exact versions (great for reproducibility, especially with native modules like better-sqlite3).

**Verify**

After saving .npmrc, run:

npm config list

You should see your project settings applied. No warnings.

👉 You want me to also prep a .gitattributes (to normalize line endings and silence the CRLF warnings Git showed earlier)? That way your commits stay clean on Windows/Linux.

**You said:**

even with those null in the file the warnings still came up so i just took them out and left it at that. a .gitattributes would be great

**ChatGPT said:**

Got it—here’s a clean .gitattributes that keeps code/text as LF in the repo, uses CRLF for Windows scripts, and treats binaries correctly so diffs don’t get noisy.

**Add this file at repo root:**.gitattributes

# Default: normalize to LF in repo; convert on checkout per rules below

\* text=auto eol=lf

# Windows scripts should stay CRLF on checkout

\*.bat eol=crlf

\*.cmd eol=crlf

\*.ps1 eol=crlf

# Unix-y scripts explicitly LF

\*.sh eol=lf

# Source & configs (force LF for consistency)

\*.ts text eol=lf

\*.tsx text eol=lf

\*.js text eol=lf

\*.cjs text eol=lf

\*.mjs text eol=lf

\*.json text eol=lf

\*.yml text eol=lf

\*.yaml text eol=lf

\*.md text eol=lf

\*.css text eol=lf

\*.html text eol=lf

pnpm-workspace.yaml text eol=lf

pnpm-lock.yaml text eol=lf

turbo.json text eol=lf

tsconfig.json text eol=lf

.eslintrc.\* text eol=lf

.prettierrc\* text eol=lf

.gitignore text eol=lf

# Binaries (never normalize / diff)

\*.png binary

\*.jpg binary

\*.jpeg binary

\*.gif binary

\*.webp binary

\*.pdf binary

\*.zip binary

\*.7z binary

\*.gz binary

\*.tar binary

\*.db binary

\*.sqlite binary

\*.wasm binary

**One-liner to create + commit (PowerShell)**

@" # Default: normalize to LF in repo; convert on checkout per rules below \* text=auto eol=lf \*.bat eol=crlf \*.cmd eol=crlf \*.ps1 eol=crlf \*.sh eol=lf \*.ts text eol=lf \*.tsx text eol=lf \*.js text eol=lf \*.cjs text eol=lf \*.mjs text eol=lf \*.json text eol=lf \*.yml text eol=lf \*.yaml text eol=lf \*.md text eol=lf \*.css text eol=lf \*.html text eol=lf pnpm-workspace.yaml text eol=lf pnpm-lock.yaml text eol=lf turbo.json text eol=lf tsconfig.json text eol=lf .eslintrc.\* text eol=lf .prettierrc\* text eol=lf .gitignore text eol=lf \*.png binary \*.jpg binary \*.jpeg binary \*.gif binary \*.webp binary \*.pdf binary \*.zip binary \*.7z binary \*.gz binary \*.tar binary \*.db binary \*.sqlite binary \*.wasm binary "@ | Set-Content .gitattributes git add .gitattributes git commit -m "chore: add .gitattributes (LF by default, CRLF for Windows scripts, binaries as binary)" git push

If you’re good with this, I can immediately scaffold packages/core with a typed, seedable dice roller (incl. advantage/disadvantage) and wire a roll command into the CLI so it’s doing something D&D-ish out of the gate.

**You said:**

yeah we are set and the repo at the link i sent you earlier is updated. the way i'd like to do this is for you to generate prompts for what we want to add which I'll feed into chat gpt codex and get hte prs that way

**ChatGPT said:**

Perfect 👌 — I like that workflow:

1. You and I design the feature,
2. I generate a **clear PR prompt** (with context, acceptance criteria, file paths),
3. You drop it into Copilot/Codex to generate the PR,
4. We iterate.

That keeps the repo clean and leverages the PR-driven workflow you used on Grimbrain.

**🔑 Next Steps**

The first milestone is to give packages/core some “meat.” The logical starter is:

**Feature 1 — Dice Roller (seedable, advantage/disadvantage, modifiers)**

**📋 Prompt Draft (for Codex → PR)**

**Title:** feat(core): add seedable dice roller with advantage/disadvantage

**Description / PR Body:**

Implement a dice roller in `packages/core` that supports standard D&D dice notation

and seedable RNG. Include advantage/disadvantage mechanics and arithmetic modifiers.

## Requirements

- Add a module at `packages/core/src/dice.ts`.

- Expose a function:

```ts

interface RollOptions {

seed?: string;

advantage?: boolean;

disadvantage?: boolean;

}

interface RollResult {

total: number;

rolls: number[];

expression: string; // e.g. "1d20+5 adv"

}

export function roll(expr: string, opts?: RollOptions): RollResult;

* Supported features:
  + Basic dice notation: 1d20, 2d6+3, 4d8-2.
  + Advantage/disadvantage (roll 2d20, take highest or lowest).
  + Deterministic if seed provided (use seedrandom).
* Add unit tests under packages/core/tests/dice.test.ts using Vitest:
  + Can roll a single die.
  + Handles modifiers.
  + Advantage/disadvantage returns the correct die.
  + Seeded rolls are deterministic across runs.

**CLI integration**

* Add a roll command to apps/cli/index.ts.
* Usage: pnpm dev -- roll "1d20+5" adv
* Output example:
* Rolling 1d20+5 with advantage...
* Rolls: [12, 18] → total 23

---

👉 Question for you:

Do you want me to go ahead and generate the \*\*next prompt for Codex (Feature 1)\*\* so you can paste it in and get PR *#1 opened in GrimEngine?*

**You said:**

yes please

**ChatGPT said:**

Alright, here’s a clean **prompt you can drop into Codex** to generate **PR #1 for GrimEngine**. It’s written in the same structured way you’d normally write your PR bodies.

**🔖 Prompt for Codex**

**Title:** feat(core): add seedable dice roller with advantage/disadvantage

**PR Body:**

Implement a dice roller module in `packages/core` that supports D&D dice notation,

advantage/disadvantage, and deterministic seeding. Wire it into the CLI.

*## Requirements*

*### Core*

- Create a new file: `packages/core/src/dice.ts`

- Implement and export a function:

```ts

interface RollOptions {

seed?: string;

advantage?: boolean;

disadvantage?: boolean;

}

interface RollResult {

total: number;

rolls: number[];

expression: string; *// e.g. "1d20+5 adv"*

}

export function roll(expr: string, opts?: RollOptions): RollResult;

* Features:
  + Support expressions like 1d20, 2d6+3, 4d8-2
  + Apply arithmetic modifiers
  + Advantage/disadvantage → roll 2 d20s, pick highest or lowest
  + If opts.seed is provided, use seedrandom for deterministic rolls

**Tests**

* Add packages/core/tests/dice.test.ts with Vitest
* Cover:
  + Simple rolls (e.g., 1d6)
  + Modifiers (e.g., 2d6+3)
  + Advantage and disadvantage
  + Seeded repeatability

**CLI**

* Update apps/cli/index.ts to add a roll command
* Usage:

pnpm dev -- roll "1d20+5" adv

* Output example:
* Rolling 1d20+5 with advantage...
* Rolls: [12, 18] → total 23

**Acceptance Criteria**

* pnpm dev -- roll "1d20+5" outputs a valid result
* pnpm dev -- roll "1d20+5" adv shows advantage and correct total
* Tests under packages/core/tests/dice.test.ts all pass with pnpm test

---

✅ This will give you a `@grimengine/core` starting point + CLI integration.

Next feature after this one could be \*\*character ability score rolling\*\* (4d6 drop lowest, assign to stats).

Do you want me to prep the \*\*Feature 2 prompt\*\* right after this so you can queue it up once PR *#1 merges?*

**You said:**

Uploaded a file

these reults look good to me. shoudl be ready to do feature 2

**ChatGPT said:**

Thought for 23s

Heck yes—Feature 2 time. Here’s a **drop-in PR prompt** for Codex that adds **ability score generation** (4d6 drop lowest, Standard Array, and Point Buy with validation), plus CLI commands and tests.

**🔖 Prompt for Codex → PR #2**

**Title:** feat(core): ability scores (4d6 drop lowest, standard array, point buy) + CLI

**PR Body:**

Add core utilities for generating and validating D&D 5e ability scores:

- 4d6 drop lowest (seedable)

- Standard Array

- Point Buy (27 points, 8–15 range, PHB/SRD costs)

Also expose CLI commands to use these features.

## Files & Structure

### Core

Create `packages/core/src/abilityScores.ts` with:

```ts

export type AbilityName = 'STR' | 'DEX' | 'CON' | 'INT' | 'WIS' | 'CHA';

export type AbilityScores = Record<AbilityName, number>;

export interface AbilityRollOptions {

seed?: string; *// use same seed strategy as dice.ts*

count?: number; *// default 6*

drop?: number; *// default 1 (drop lowest per-stat)*

sort?: 'none' | 'asc' | 'desc'; *// default 'none'*

}

export interface PointBuyOptions {

budget?: number; *// default 27*

min?: number; *// default 8*

max?: number; *// default 15*

}

*// Roll a single stat using 4d6 drop lowest (respecting `drop`)*

export function roll4d6DropLowest(rng: () => number, drop?: number): { rolls: number[]; total: number };

*// Roll a full set of ability scores (default 6 stats), optionally sorted*

export function rollAbilityScores(opts?: AbilityRollOptions): { sets: number[]; details: number[][] };

*// Return standard array*

export function standardArray(): number[]; *// [15, 14, 13, 12, 10, 8]*

*// Compute total point-buy cost for a 6-number array (PHB costs)*

export function calculatePointBuyCost(arr: number[]): number;

*// Validate a 6-number array against point-buy rules (budget 27, 8..15)*

export function validatePointBuy(arr: number[], opts?: PointBuyOptions): {

ok: boolean;

cost: number;

budget: number;

errors: string[];

};

Implementation details:

* **RNG**: reuse the same seeding approach as dice.ts (seedrandom under the hood). If opts.seed is provided in rollAbilityScores, all generated stats should be deterministic.
* **4d6 drop lowest**: roll 4d6 per ability, drop the lowest drop dice (default 1), sum the rest.
* **Standard Array**: [15, 14, 13, 12, 10, 8].
* **Point Buy**: PHB/SRD cost table (8→0, 9→1, 10→2, 11→3, 12→4, 13→5, 14→7, 15→9). Budget 27, min 8, max 15, exactly 6 scores required.

**Tests**

Create packages/core/tests/abilityScores.test.ts with Vitest covering:

* roll4d6DropLowest returns 4 raw rolls and a total of the top 3 (by default).
* rollAbilityScores({ seed: 'x' }) is deterministic and returns 6 values.
* Sorting works (asc, desc, none).
* standardArray() returns [15,14,13,12,10,8].
* calculatePointBuyCost([15,14,13,12,10,8]) yields 27.
* validatePointBuy:
  + ok=true for [15,14,13,12,10,8].
  + ok=false (and error messages) if: value < 8, value > 15, not 6 scores, or cost exceeds budget.

**CLI**

Update apps/cli/index.ts to add an abilities command group with three subcommands:

1. **Roll**
   * Usage:
   * pnpm dev -- abilities roll --seed "grim" --count 6 --drop 1 --sort desc
   * Output:
   * Ability Scores (4d6 drop lowest, seed="grim", sort=desc)
   * Sets: [17, 15, 14, 12, 11, 9]
   * Details per stat: [[6,6,3,2]->15, [5,5,4,1]->14, ...]
2. **Standard**
   * Usage:
   * pnpm dev *-- abilities standard*
   * Output:
   * Standard Array: [15, 14, 13, 12, 10, 8]
3. **Point Buy**
   * Usage (comma-separated 6 numbers):
   * pnpm dev -- abilities pointbuy "15,14,13,12,10,8"
   * Output (valid):
   * Point Buy: OK (cost 27 / budget 27)
   * Output (invalid example):
   * Point Buy: INVALID
   * - Value 7 is below minimum 8
   * - Total cost 31 exceeds budget 27

Notes:

* Keep CLI parsing simple; you can use process.argv or a tiny arg parser.
* Reuse the roll RNG seeding strategy from packages/core/src/dice.ts (export/utilize the RNG factory if helpful).

**Acceptance Criteria**

* pnpm test passes (new tests included).
* pnpm dev -- abilities roll prints a 6-number set and shows per-stat dice details.
* pnpm dev -- abilities standard prints [15, 14, 13, 12, 10, 8].
* pnpm dev -- abilities pointbuy "15,14,13,12,10,8" prints OK (cost 27 / budget 27).
* Invalid arrays produce clear error messages (min/max, count, budget).

---

Want me to queue \*\*Feature 3\*\* right after (e.g., \*\*Saving throw / ability check resolution\*\* using proficiency + ability mods, wired to the dice roller)?

**You said:**

Uploaded a file

Feature 2 results look good to me

**ChatGPT said:**

Beautiful ✅ — I see from your test + CLI output:

* **All tests passed** (13 total across dice + ability scores).
* CLI works for:
  + abilities roll (4d6 drop lowest w/ details).
  + abilities standard (prints the standard array).
  + abilities pointbuy (valid vs invalid arrays, correct budget validation).
* Invalid point buy exits with non-zero code (which is correct behavior for CLI tools).

That means **Feature 2 is locked in** and solid.

**🚀 Suggested Feature 3**

Now we’re ready to move up from dice + abilities into **checks and saves**. This ties into actual gameplay:

**Feature 3 — Checks & Saving Throws**

* Use the dice roller to resolve ability checks and saving throws.
* Support proficiency bonus, advantage/disadvantage, DC target.
* Wire into CLI for quick usage.

**🔖 Prompt for Codex → PR #3**

**Title:** feat(core): add ability checks and saving throws

**PR Body:**

Add core utilities to resolve ability checks and saving throws using the dice roller.

Support proficiency, modifiers, advantage/disadvantage, and DC targets.

*## Core*

Create `packages/core/src/checks.ts` with:

```ts

import { AbilityName } from './abilityScores';

export interface CheckOptions {

ability: AbilityName; *// e.g. 'STR', 'DEX'*

modifier?: number; *// ability modifier, e.g. +3*

proficient?: boolean; *// if true, add proficiency bonus*

proficiencyBonus?: number; *// default 2 if proficient*

advantage?: boolean;

disadvantage?: boolean;

dc?: number; *// difficulty class*

seed?: string;

}

export interface CheckResult {

rolls: number[];

total: number;

success?: boolean; *// only set if dc is provided*

expression: string; *// e.g. "1d20+3 adv vs DC 15"*

}

export function abilityCheck(opts: CheckOptions): CheckResult;

export function savingThrow(opts: CheckOptions): CheckResult;

Implementation:

* Roll 1d20 (adv/disadv if requested).
* Add modifier.
* If proficient, add proficiencyBonus (default 2).
* If dc provided, success = total >= dc.

**Tests**

Add packages/core/tests/checks.test.ts:

* Simple ability check (e.g. STR +3).
* With proficiency (adds +2).
* Advantage/disadvantage works.
* Saving throw identical behavior (for now).
* Passing and failing against a DC.

**CLI**

Update apps/cli/index.ts to add a check command:

Usage examples:

pnpm dev -- check STR +3 --dc 15

pnpm dev -- check DEX +2 --proficient --pb 2 --adv --dc 14

pnpm dev -- save CON +5 --dc 12

Output example:

Ability Check: STR +3 vs DC 15

Rolls: [12] → total 15

Result: SUCCESS

Another example with adv:

Ability Check: DEX +2 (proficient +2) adv vs DC 14

Rolls: [7, 18] → total 22

Result: SUCCESS

**Acceptance Criteria**

* pnpm test passes (new tests included).
* CLI check and save commands produce readable results with rolls, modifiers, and success/failure.
* Advantage/disadvantage correctly applies.
* Seeded runs are deterministic.

---

👉 Want me to go ahead and draft \*\*Feature 4\*\* after this (probably basic combat attack roll + damage resolution), so you’ll have it ready once PR *#3 merges?*

**You said:**

Uploaded a file

There's the command output from feature 3. Seems good. If you agree we can move to feature 4

**ChatGPT said:**

Thought for 11s

Absolutely—Feature 3 output looks perfect. Let’s roll into **Feature 4**: basic **attack roll + damage resolution**. Here’s a clean PR prompt you can paste into Codex.

**🔖 Prompt for Codex → PR #4**

**Title:** feat(core): attack roll + damage resolution (hit/miss/crit, adv/dis, resist/vuln)

**PR Body:**

Add attack roll and damage resolution utilities to @grimengine/core, using the

existing dice roller. Support advantage/disadvantage, ability/proficiency modifiers,

critical hits (natural 20), fumbles (natural 1), and damage resistance/vulnerability.

Wire minimal CLI commands to exercise it.

*## Core*

Create `packages/core/src/combat.ts` with:

```ts

import type { AbilityName } from './abilityScores';

export interface AttackRollOptions {

abilityMod?: number; *// e.g. STR or DEX mod applied to attack roll*

proficient?: boolean; *// adds proficiency bonus if true*

proficiencyBonus?: number; *// default 2 if proficient*

advantage?: boolean;

disadvantage?: boolean;

seed?: string;

targetAC?: number; *// if provided, compute hit/miss*

}

export interface AttackRollResult {

d20s: number[]; *// the raw d20(s) rolled (1 or 2 values)*

natural: number; *// the d20 result used (after adv/dis)*

total: number; *// attack total after mods*

isCrit: boolean; *// true if natural 20*

isFumble: boolean; *// true if natural 1*

hit?: boolean; *// set if targetAC provided*

expression: string; *// e.g. "1d20+7 adv vs AC 15"*

}

export interface DamageRollOptions {

*// e.g. "1d8+3" for a longsword with STR mod, or "2d6+4" for greatsword, etc.*

expression: string;

crit?: boolean; *// if true, double the dice (not modifiers)*

resistance?: boolean; *// halves final damage (round down)*

vulnerability?: boolean; *// doubles final damage*

seed?: string;

}

export interface DamageRollResult {

rolls: number[]; *// the individual damage dice (pre-crit-doubling)*

critRolls?: number[]; *// the extra crit dice (if crit=true)*

baseTotal: number; *// sum of dice + flat modifiers (pre resist/vuln)*

finalTotal: number; *// after resist/vuln*

expression: string; *// e.g. "1d8+3 (crit)" or "1d8+3 (resist)"*

}

export interface ResolveAttackOptions extends AttackRollOptions {

damage: DamageRollOptions; *// damage to roll if hit or on-crit*

}

export interface ResolveAttackResult {

attack: AttackRollResult;

damage?: DamageRollResult; *// present if attack.hit === true (or you decide to roll on crit-only)*

}

export function attackRoll(opts: AttackRollOptions): AttackRollResult;

*/\*\**

*\* Roll damage. If `crit === true`, double dice only (not flat modifiers).*

*\* For resistance, halve finalTotal (round down). For vulnerability, double finalTotal.*

*\*/*

export function damageRoll(opts: DamageRollOptions): DamageRollResult;

*/\*\**

*\* Convenience function:*

*\* - roll attack with `attackRoll`*

*\* - if hit (or natural 20), roll damage with `damageRoll`, passing crit=true on natural 20*

*\*/*

export function resolveAttack(opts: ResolveAttackOptions): ResolveAttackResult;

Implementation notes:

* Use the existing dice roller in packages/core/src/dice.ts.
* Advantage/disadvantage only affects the d20 for the attack roll.
* isCrit = (natural === 20), isFumble = (natural === 1).
* When crit === true, **double only the number of dice** in the damage expression; do not double flat modifiers.
* Resistance/vulnerability apply after modifiers; order: roll → add modifiers → resist/vuln transform → clamp to ≥ 0.
* targetAC is optional; if provided, compute hit = !isFumble && (isCrit || total >= targetAC).

**Tests**

Add packages/core/tests/combat.test.ts (Vitest) covering:

* Attack roll without adv/dis, with ability + prof bonus applied.
* Advantage and disadvantage behavior (use seeds for determinism).
* Natural 20 sets isCrit=true and hits regardless of AC; natural 1 sets isFumble=true and misses.
* damageRoll doubles dice on crit (modifiers not doubled).
* Resistance halves, vulnerability doubles (verify rounding down on halves).
* resolveAttack:
  + Miss: returns attack and no damage.
  + Hit non-crit: returns correct damage.
  + Crit: returns damage with doubled dice.

**CLI**

Update apps/cli/index.ts with two commands:

1. **attack** (roll only the attack)
   * Usage:
   * pnpm dev -- attack --mod +7 --pb 2 --proficient --adv --ac 15
   * Output example:
   * Attack: 1d20+7 adv vs AC 15
   * Rolls: [6, 17] → natural 17 → total 24
   * Result: HIT
2. **damage** (roll damage with flags)
   * Usage:
   * pnpm dev -- damage "1d8+3" --crit
   * pnpm dev -- damage "2d6+4" --resist
   * pnpm dev -- damage "1d10+2" --vuln
   * Output example:
   * Damage: 1d8+3 (crit)
   * Rolls: [7] + crit [5] → base 15 → final 15
3. **resolve** (one-shot: attack then conditional damage)
   * Usage:
   * pnpm dev -- resolve --mod +5 --ac 14 --dmg "1d8+3" --adv
   * Output example (miss):
   * Attack: 1d20+5 adv vs AC 14
   * Rolls: [2, 7] → natural 7 → total 12 → MISS
   * Output example (crit):
   * Attack: 1d20+5 vs AC 14
   * Rolls: [20] → natural 20 → total 25 → CRIT!
   * Damage: 1d8+3 (crit) → final 17

Parsing tips:

* --mod +5 ⇒ parse +5 or -1 into number.
* --pb 2 only matters if --proficient is present (default PB=2).
* Flags: --adv, --dis, --crit, --resist, --vuln.
* --dmg "<expr>" for resolve.

**Acceptance Criteria**

* pnpm test passes including new combat tests.
* pnpm dev -- attack ... prints rolls, natural, total, and HIT/MISS/CRIT clearly.
* pnpm dev -- damage ... prints rolls, crit extra dice (if any), base and final totals.
* pnpm dev -- resolve ... performs attack then conditional damage (crits double dice).
* Advantage/disadvantage and crit/fumble logic match 5e expectations.

---

If you want, I can prep \*\*Feature 5\*\* right after (weapon data model w/ properties like `finesse`, `versatile`, and auto-selecting STR/DEX mod & damage die).

**You said:**

Uploaded a file

4 looks good to me. ready for 5 if you agree

**ChatGPT said:**

Thought for 55s

Amazing momentum. Let’s lock in **Feature 5**: a small, SRD-flavored **weapon data model** + rules that auto-pick STR/DEX (finesse/ranged/thrown), handle **versatile** damage, and flow through your existing attack/damage pipeline. Includes tests and CLI.

**🔖 Prompt for Codex → PR #5**

**Title:** feat(core): weapon model + STR/DEX selection, thrown/finesse rules, versatile damage, CLI

**PR Body:**

Introduce a minimal SRD-style weapon data model and integrate it with combat:

- Auto-pick STR/DEX based on weapon & actor stats (finesse, ranged, thrown)

- Versatile damage when two-handed

- Proficiency bonus based on weapon category (simple/martial)

- Convenience "weapon attack" CLI command using the new rules

- A tiny rules-srd adapter that ships a handful of common weapons

This PR does NOT handle ammo tracking, loading rates, size-based disadvantages, or inventory.

(Those can come later.)

*## Files & Structure*

*### Core types & logic*

Create `packages/core/src/weapons.ts`:

```ts

export type WeaponCategory = 'simple' | 'martial';

export type WeaponType = 'melee' | 'ranged';

export type DamageType = 'slashing' | 'piercing' | 'bludgeoning';

export type AbilityName = 'STR' | 'DEX' | 'CON' | 'INT' | 'WIS' | 'CHA';

export interface Weapon {

name: string;

category: WeaponCategory; *// simple | martial*

type: WeaponType; *// melee | ranged*

damage: { expression: string; type: DamageType }; *// e.g. 1d8 slashing*

versatile?: { expression: string }; *// e.g. 1d10 if two-handed*

properties?: {

finesse?: boolean;

light?: boolean;

heavy?: boolean;

thrown?: { normal: number; long: number } | true; *// true = no range data*

reach?: boolean;

twoHanded?: boolean; *// informational only here*

ammunition?: boolean; *// informational only here*

loading?: boolean; *// informational only here*

};

range?: { normal: number; long?: number }; *// for ranged weapons*

}

*/\*\* Actor ability mods needed for weapon math \*/*

export interface AbilityMods {

STR?: number; DEX?: number; CON?: number; INT?: number; WIS?: number; CHA?: number;

}

export interface Proficiencies {

simple?: boolean;

martial?: boolean;

}

export interface WeaponAttackInput {

weapon: Weapon;

abilities: AbilityMods; *// e.g. { STR: 3, DEX: 1 }*

proficiencies?: Proficiencies; *// e.g. { martial: true }*

proficiencyBonus?: number; *// default 2 when proficient*

twoHanded?: boolean; *// choose versatile damage if available*

advantage?: boolean;

disadvantage?: boolean;

targetAC?: number;

seed?: string;

}

*/\*\* Decide which ability to use for this weapon attack: \*/*

export function chooseAttackAbility(weapon: Weapon, abilities: AbilityMods, opts?: { thrown?: boolean }): AbilityName;

*/\*\* Build attack/damage inputs from weapon + actor, then call existing combat helpers. \*/*

export function resolveWeaponAttack(input: WeaponAttackInput): import('./combat').ResolveAttackResult;

Implementation rules:

* **Melee**: use STR, unless finesse → pick higher of STR/DEX.
* **Ranged**: use DEX.
* **Thrown** melee weapon: “use the same ability modifier as a melee attack with that weapon”  
  (so STR by default; if the weapon is finesse, allow DEX).
* **Versatile**: if twoHanded and weapon.versatile present → use its expression for damage; otherwise base damage.
* **Proficiency**: if proficiencies[weapon.category] → add proficiencyBonus (default 2) to attack roll.
* Pass through advantage/disadvantage, AC, and seed to existing attackRoll/damageRoll via resolveAttack.

**SRD adapter (tiny, embedded list)**

Create packages/adapters/rules-srd/src/weapons.ts exporting a curated set:

import type { Weapon } from '@grimengine/core/src/weapons'; export const WEAPONS: Weapon[] = [ { name: 'Dagger', category: 'simple', type: 'melee', damage: { expression: '1d4+0', type: 'piercing' }, properties: { finesse: true, light: true, thrown: { normal: 20, long: 60 } } }, { name: 'Mace', category: 'simple', type: 'melee', damage: { expression: '1d6+0', type: 'bludgeoning' } }, { name: 'Handaxe', category: 'simple', type: 'melee', damage: { expression: '1d6+0', type: 'slashing' }, properties: { thrown: { normal: 20, long: 60 }, light: true } }, { name: 'Rapier', category: 'martial', type: 'melee', damage: { expression: '1d8+0', type: 'piercing' }, properties: { finesse: true } }, { name: 'Longsword', category: 'martial', type: 'melee', damage: { expression: '1d8+0', type: 'slashing' }, versatile: { expression: '1d10+0' } }, { name: 'Greatsword', category: 'martial', type: 'melee', damage: { expression: '2d6+0', type: 'slashing' }, properties: { heavy: true, twoHanded: true } }, { name: 'Longbow', category: 'martial', type: 'ranged', damage: { expression: '1d8+0', type: 'piercing' }, properties: { ammunition: true, heavy: true, twoHanded: true }, range: { normal: 150, long: 600 } }, ];

Notes:

* Set +0 in damage expressions; we’ll add ability mods in resolveWeaponAttack so data stays clean.
* This adapter is just a seed; later we can hydrate from the public 5e API with caching.

**Integrate with combat**

Use existing helpers from packages/core/src/combat.ts:

* Build AttackRollOptions:
  + ability mod from chooseAttackAbility
    - proficiency if actor is proficient with the weapon’s category
* Build DamageRollOptions:
  + expression = weapon.damage.expression or weapon.versatile.expression (if twoHanded)
  + add the same ability mod to damage (STR for melee/thrown; DEX for ranged/finesse)
  + forward seed

**Tests**

Add packages/core/tests/weapons.test.ts (Vitest). Cover:

* **Ability choice:**
  + Dagger (finesse): with STR=1, DEX=4, choose DEX.
  + Rapier (finesse): same as dagger.
  + Handaxe (thrown melee, not finesse): choose STR.
  + Longbow (ranged): choose DEX.
* **Proficiency:**
  + Longsword (martial): with { martial: true, pb=2 }, total includes +2 to attack; without proficiency, it doesn’t.
* **Versatile damage:**
  + Longsword one-handed uses 1d8+mod; two-handed uses 1d10+mod.
* **Resolve end-to-end:**
  + resolveWeaponAttack with seed → deterministic attack and damage.
  + Crits double **dice only** (already in damageRoll), ability mod applied once.
  + Resistance/vulnerability still applied from damageRoll if passed later (not required in this PR).

Use fixed seeds so tests are stable.

**CLI**

Update apps/cli/index.ts to add a weapon command group with:

1. weapon list
   * Prints available weapon names from the rules-srd adapter.
2. weapon info "<name>"
   * Prints category, type, damage, versatile (if any), and key properties.
3. weapon attack "<name>" --str <n> --dex <n> [--pb <n>] [--profs "simple|martial"] [--twohanded] [--adv|--dis] [--ac <n>] [--seed "<s>"]
   * Example:
   * pnpm dev -- weapon attack "Longsword" --str 3 --dex 1 --pb 2 --profs martial --twohanded --ac 15 --adv
   * Output should show:
     + chosen ability (STR/DEX)
     + attack expression (e.g., 1d20+5 adv vs AC 15)
     + d20 rolls, natural, total, hit/miss/crit
     + damage expression used (1d10+3 if two-handed), rolls, base and final totals

Parsing notes:

* --profs may be simple, martial, or a comma list (treat any match as true).
* Ability mods default to 0 if not provided.
* --twohanded toggles versatile, if present on the weapon.

**Acceptance Criteria**

* pnpm test passes with new weapons.test.ts.
* pnpm dev -- weapon list shows at least: Dagger, Mace, Handaxe, Rapier, Longsword, Greatsword, Longbow.
* pnpm dev -- weapon info "Longsword" displays category/type/damage/versatile/properties.
* pnpm dev -- weapon attack correctly picks ability mod (finesse/ranged/thrown), applies proficiency, and uses versatile damage when --twohanded.
* End-to-end attack+damage results are deterministic with --seed.

*---*

If you’re good with this, the natural \*\*Feature 6\*\* after weapons is a \*\*character sheet stub\*\* (race/class level, proficiency bonus derivation, ability modifiers, proficient weapons/skills, saves), so the CLI can compute mods automatically from a JSON character. Want me to prep that next?

**You said:**

Uploaded a file

5 looks good to me. if you agree, should be ready for 6

**ChatGPT said:**

Thought for 8s

Awesome—Feature 5 looks tight. Let’s move on to a lightweight but powerful **character sheet stub** that plugs into everything you’ve built (checks, saves, weapons).

Below is a **ready-to-paste PR prompt** for Codex.

**🔖 Prompt for Codex → PR #6**

**Title:** feat(core): character sheet stub (scores→mods, prof bonus by level, profs, skills/saves) + CLI

**PR Body:**

Add a minimal character model in @grimengine/core with:

- Ability scores → ability modifiers

- Proficiency bonus derived from level (5e table)

- Proficiencies: weapon categories (simple/martial), saving throws, skills

- Utilities to resolve checks/saves/weapon attacks using character data

- CLI: load character from JSON, show summary, run check/save/weapon attack

This is intentionally small (no classes/feats/spells yet). It should \*compose\* with existing

checks/combat/weapons modules.

*## Files & Structure*

*### Core types & helpers*

Create `packages/core/src/character.ts`:

```ts

import type { AbilityName } from './abilityScores';

import type { Proficiencies as WeaponProfs } from './weapons';

import { abilityCheck, savingThrow } from './checks';

import { resolveWeaponAttack } from './weapons';

import type { ResolveAttackResult } from './combat';

export type SkillName =

| 'Acrobatics' | 'Animal Handling' | 'Arcana' | 'Athletics' | 'Deception'

| 'History' | 'Insight' | 'Intimidation' | 'Investigation'

| 'Medicine' | 'Nature' | 'Perception' | 'Performance' | 'Persuasion'

| 'Religion' | 'Sleight of Hand' | 'Stealth' | 'Survival';

export interface CharacterAbilityScores { STR: number; DEX: number; CON: number; INT: number; WIS: number; CHA: number; }

export type AbilityMods = Record<AbilityName, number>;

export interface CharacterProficiencies {

weapons?: WeaponProfs; *// { simple?: boolean; martial?: boolean }*

saves?: AbilityName[]; *// e.g., ['WIS', 'CHA']*

skills?: SkillName[]; *// proficient skills*

}

export interface Character {

name: string;

level: number; *// 1..20*

abilities: CharacterAbilityScores; *// 3..18+ (raw score)*

proficiencies?: CharacterProficiencies;

}

export function abilityMod(score: number): number; *// floor((score - 10)/2)*

export function proficiencyBonusForLevel(level: number): number;

*// 1–4: +2, 5–8: +3, 9–12: +4, 13–16: +5, 17–20: +6*

export function abilityMods(abilities: CharacterAbilityScores): AbilityMods;

export function isProficientSave(c: Character, ability: AbilityName): boolean;

export function isProficientSkill(c: Character, skill: SkillName): boolean;

*/\*\* Resolve an ability check for the character (optionally vs DC). \*/*

export function characterAbilityCheck(c: Character, ability: AbilityName, opts?: {

dc?: number; advantage?: boolean; disadvantage?: boolean; seed?: string; extraMod?: number; // e.g. situational bonus

}): ReturnType<typeof abilityCheck>;

*/\*\* Resolve a saving throw for the character (optionally vs DC). \*/*

export function characterSavingThrow(c: Character, ability: AbilityName, opts?: {

dc?: number; advantage?: boolean; disadvantage?: boolean; seed?: string; extraMod?: number;

}): ReturnType<typeof savingThrow>;

*/\*\* Resolve a weapon attack for the character (uses STR/DEX rules + prof bonus). \*/*

export function characterWeaponAttack(c: Character, weaponName: string, opts?: {

twoHanded?: boolean; advantage?: boolean; disadvantage?: boolean; targetAC?: number; seed?: string;

}): ResolveAttackResult;

Implementation details:

* abilityMod(score) = Math.floor((score - 10) / 2).
* proficiencyBonusForLevel via the standard 5e thresholds (1–4:+2, 5–8:+3, 9–12:+4, 13–16:+5, 17–20:+6).
* characterAbilityCheck:
  + base modifier = abilityMod(abilities[ability])
  + if proficient in that *skill* (later) or specific ability? For Feature 6 keep it to raw ability  
    checks (no skills’ separate mapping yet) but accept extraMod so CLI can simulate a skill bonus.
  + if proficiency is intended: for this feature, do **not** add proficiency by default on ability  
    checks (skills come next), but do add it for saving throws *if* the character is proficient.
* characterSavingThrow:
  + base modifier = ability mod
  + add PB if ability is in proficiencies.saves
* characterWeaponAttack:
  + choose STR/DEX from existing weapon logic
  + add PB if proficient with weapon category
  + pass ability mod to damage
  + wire to resolveWeaponAttack
* In all resolvers, read PB via proficiencyBonusForLevel(c.level).

**Rules-SRD adapter – weapon lookup**

Update packages/adapters/rules-srd/src/weapons.ts to export a helper:

export function getWeaponByName(name: string): Weapon | undefined;

**Tests**

Add packages/core/tests/character.test.ts (Vitest) that covers:

* abilityMod: known values (8→-1, 10→0, 14→+2, 18→+4).
* proficiencyBonusForLevel: table checks (1→+2, 5→+3, 9→+4, 13→+5, 17→+6, 20→+6).
* characterSavingThrow:
  + proficient vs not proficient adds PB.
  + success/failure vs DC with deterministic seed.
* characterWeaponAttack:
  + with { level: 1, proficiencies: { martial: true } } attacking with Longsword adds PB.
  + finesse/ranged choice selects DEX when higher.
  + two-handed uses versatile damage.
  + deterministic with seed.

**CLI**

Update apps/cli/index.ts with a character command group:

1. character load "<path.json>"
   * Reads a JSON file and stores the character **in-memory** for the session (simple module-level var).
   * JSON shape:

{ "name": "Aerin", "level": 3, "abilities": { "STR": 16, "DEX": 14, "CON": 12, "INT": 10, "WIS": 10, "CHA": 8 }, "proficiencies": { "weapons": { "martial": true }, "saves": ["STR", "CON"], "skills": ["Athletics", "Perception"] } }

* + Output: "Loaded character Aerin (lvl 3). PB +2"

1. character show
   * Prints: name, level, PB, ability scores+mods, weapon profs, save profs, skills list (if present).
2. character check <ABILITY> [--dc N] [--adv|--dis] [--seed S] [--extraMod N]
   * Uses characterAbilityCheck.
   * Output: ability, mods, rolls, total, and success/failure if DC given.
3. character save <ABILITY> [--dc N] [--adv|--dis] [--seed S]
   * Uses characterSavingThrow.
4. character attack "<WEAPON\_NAME>" [--twohanded] [--ac N] [--adv|--dis] [--seed S]
   * Uses characterWeaponAttack and the rules-srd weapon list.

Parsing notes:

* Keep parsing minimal; default PB comes from level, ability mods from scores.
* If character load hasn’t run yet, error out with a helpful message.

**Acceptance Criteria**

* pnpm test passes including new character.test.ts.
* character load followed by character show prints PB and ability mods correctly.
* character save STR --dc 12 applies PB if STR is a proficient save.
* character attack "Longsword" --ac 15 uses STR/DEX and PB as expected; deterministic with --seed.
* The feature does not introduce circular imports or break existing tests.

---

If you want, I can queue \*\*Feature 7\*\* next: \*\*skills layer\*\* (mapping skills → abilities, adding proficiency/expertise to checks, passive Perception, and CLI `character skill <name>`).

**You said:**

tests passed. i could be doing this wrong, but it seems like the character command there doesn't exist after it

**ChatGPT said:**

Nice—if tests passed but character isn’t a recognized CLI command, that almost always means the PR added the **core** code but didn’t wire the **CLI** entry (apps/cli/index.ts). Two quick sanity checks, then I’ll give you a drop-in “hotfix” PR prompt to add the CLI group.

**Quick checks**

Run these from repo root:

# 1) See what the CLI currently exposes pnpm dev -- help # 2) Try a known-existing command to confirm CLI is running pnpm dev -- weapon list

If help doesn’t show a character group, let’s wire it up.

**🔖 Hotfix Prompt for Codex → PR**fix(cli): add 'character' command group

**Title:** fix(cli): wire up 'character' command group (load/show/check/save/attack)

**PR Body:**

The core for Feature 6 landed, but the CLI entry points were not added.

This PR wires a `character` command group into `apps/cli/index.ts` and

implements in-memory storage for the loaded character.

## Files to change

- `apps/cli/index.ts` (add command parsing + character subcommands)

- `packages/adapters/rules-srd/src/weapons.ts` (ensure getWeaponByName is exported)

## Requirements

### 1) CLI plumbing

In `apps/cli/index.ts`, add a tiny argument router:

- Top-level commands that already exist (keep): `roll`, `abilities ...`, `weapon ...`.

- New: `character <subcmd>`

Maintain a module-scope variable to hold the loaded character for the current process:

```ts

*// apps/cli/index.ts (top)*

import { readFileSync } from 'node:fs';

import {

abilityMod,

abilityMods as computeMods,

proficiencyBonusForLevel,

characterAbilityCheck,

characterSavingThrow,

characterWeaponAttack,

type Character,

} from '@grimengine/core/src/character';

import { getWeaponByName } from '@grimengine/rules-srd/src/weapons'; *// adjust path if package name differs*

let CURRENT\_CHARACTER: Character | null = null;

**2) Implement subcommands**

* character load "<path.json>"
  + Read JSON, parse into Character, store in CURRENT\_CHARACTER.
  + Print: Loaded character <name> (lvl <n>). PB +<pb>.
* character show
  + If not loaded: print helpful error and exit(1).
  + Print: name, level, PB, each ability score with mod (STR 16 (+3)), weapon profs, save profs, skills.
* character check <ABILITY> [--dc N] [--adv|--dis] [--seed S] [--extraMod N]
  + Use characterAbilityCheck(CURRENT\_CHARACTER, ability, {...}).
  + Print rolls, total, and SUCCESS/FAIL if DC provided.
* character save <ABILITY> [--dc N] [--adv|--dis] [--seed S]
  + Use characterSavingThrow.
* character attack "<WEAPON\_NAME>" [--twohanded] [--ac N] [--adv|--dis] [--seed S]
  + Look up weapon via getWeaponByName.
  + Use characterWeaponAttack.

Keep argument parsing lightweight (simple scan of process.argv.slice(2)).

**3) Examples (should work after this PR)**

pnpm dev -- character load ".\example\fighter.json"

pnpm dev -- character show

pnpm dev -- character check STR --dc 13 --seed grim

pnpm dev -- character save CON --dc 12 --adv

pnpm dev -- character attack "Longsword" --twohanded --ac 15 --seed grim

**4) Optional: sample character file**

Add example/fighter.json (not required for tests, helpful for manual runs):

{ "name": "Aerin", "level": 3, "abilities": { "STR": 16, "DEX": 14, "CON": 12, "INT": 10, "WIS": 10, "CHA": 8 }, "proficiencies": { "weapons": { "martial": true }, "saves": ["STR", "CON"], "skills": ["Athletics", "Perception"] } }

**Acceptance Criteria**

* pnpm dev -- character load "<path>" loads and confirms PB.
* pnpm dev -- character show prints scores + mods + profs.
* pnpm dev -- character check/save ... resolve and print results.
* pnpm dev -- character attack "Longsword" ... performs attack + damage using PB and STR/DEX selection.
* Existing commands still work (roll, abilities, weapon).

---

If you want, after this lands we can do \*\*Feature 7: Skills layer\*\* (map each skill → ability, apply proficiency/expertise to checks, add `character skill <name>` and `passive Perception` calc).

**You said:**

PS C:\dev\grimengine> pnpm dev -- character load "Characters\Aerin.json" > grimengine@1.0.0 dev C:\dev\grimengine > tsx apps/cli/index.ts "character" "load" "Characters\\Aerin.json" Loaded character Aerin (lvl 3). PB +2 PS C:\dev\grimengine> pnpm dev -- character show > grimengine@1.0.0 dev C:\dev\grimengine > tsx apps/cli/index.ts "character" "show" No character loaded. Use pnpm dev -- character load "<path.json>" first.  ELIFECYCLE  Command failed with exit code 1. PS C:\dev\grimengine>

**ChatGPT said:**

Thought for 6s

Good catch — that’s because each pnpm dev -- … runs a **new Node process**, so the in-memory CURRENT\_CHARACTER is lost between commands. Let’s persist the loaded character to a tiny session file.

Here’s a clean PR prompt to paste into Codex.

**🔖 Prompt for Codex → PR**fix(cli): persist loaded character across commands

**Title:** fix(cli): persist loaded character across commands (session store)

**PR Body:**

CLI currently keeps the loaded character only in-memory; each `pnpm dev *-- …` starts*

a new process, so `character show` fails after `character load`. This PR adds a tiny

session store so the character persists between commands.

## Changes

### 1) Add a CLI session store

Create `apps/cli/session.ts`:

```ts

import { mkdirSync, readFileSync, writeFileSync, existsSync, rmSync } from 'node:fs';

import { join } from 'node:path';

import type { Character } from '@grimengine/core/src/character';

const DIR = join(process.cwd(), '.data', 'session');

const FILE = join(DIR, 'character.json');

export function saveCharacter(c: Character) {

mkdirSync(DIR, { recursive: true });

writeFileSync(FILE, JSON.stringify(c, null, 2), 'utf-8');

}

export function loadCharacter(): Character | null {

if (!existsSync(FILE)) return null;

try {

const raw = readFileSync(FILE, 'utf-8');

return JSON.parse(raw) as Character;

} catch {

return null;

}

}

export function clearCharacter() {

try { if (existsSync(FILE)) rmSync(FILE); } catch {}

}

**2) Wire session into the CLI**

Update apps/cli/index.ts:

* On character load "<path.json>":
  + Read, parse, **saveCharacter(json)**, and print confirmation (PB, etc.).
* On all other character subcommands (show, check, save, attack):
  + Replace the in-memory variable with const CURRENT = loadCharacter();
  + If null, print: No character loaded. Use \pnpm dev -- character load "<path.json>"` first.` and exit(1).
* Add a convenience character unload (optional):
  + Calls clearCharacter() and prints Character session cleared.

**3) Small DX touches**

* Ensure .data/ is in .gitignore (already present).
* Keep everything else unchanged.

**Examples (now work across processes)**

pnpm dev *-- character load ".\Characters\Aerin.json"*

pnpm dev *-- character show*

pnpm dev *-- character check STR --dc 13 --seed grim*

pnpm dev *-- character save CON --dc 12 --adv*

pnpm dev *-- character attack "Longsword" --twohanded --ac 15 --seed grim*

pnpm dev *-- character unload*

**Acceptance Criteria**

* After character load, subsequent character commands work without reloading.
* character show prints the same data across separate invocations.
* character unload clears the session; show then errors as expected.
* No impact to other commands (roll, abilities, weapon).

---

If you want, I can queue \*\*Feature 7 (skills layer)\*\* next: skill→ability mapping, proficiency/expertise, `character skill <name>` check, and passive Perception calculation.

**You said:**

Uploaded a file

there we go

**ChatGPT said:**

Perfect ✅

From your logs I can see:

* **All tests passed** (38/38, turbo cached).
* character load → persisted properly.
* character show → prints abilities, mods, PB, proficiencies, skills.
* character check STR → applied base mod, failed DC check correctly.
* character save CON --adv → added PB, rolled adv, succeeded.
* character attack "Longsword" → picked STR, added PB, handled miss/hit correctly depending on seed.
* character unload → cleared session store as intended.

That means **Feature 6 is locked in and stable**. 🎉

**🚀 Next Feature Proposal (Feature 7)**

Now that the sheet is in place, the natural extension is **Skills**:

* Map each skill to its governing ability (e.g., Stealth → DEX, Athletics → STR).
* Add skill checks to character commands.
* Handle proficiency (and later expertise).
* Add passive Perception to character show.

Would you like me to draft the **PR prompt for Feature 7 (skills layer + passive perception)** so you can queue it up next?

**You said:**

yes

**ChatGPT said:**

Awesome — here’s a clean, drop-in PR prompt for **Feature 7: Skills layer + passive perception**. Paste this into Codex to generate the PR.

**🔖 Prompt for Codex → PR #7**

**Title:** feat(core): skills layer (mapping, proficiency/expertise) + passive perception + CLI

**PR Body:**

Add a skills system to @grimengine/core:

- Map each 5e skill to its governing ability

- Skill checks with proficiency and expertise (2× PB)

- Expose passive Perception on character summary

- CLI: run skill checks and list skills

This builds on existing checks/character modules; keep it small and composable.

## Files & Structure

### 1) Core: skills module

Create `packages/core/src/skills.ts`:

```ts

import type { AbilityName } from './abilityScores';

import type { SkillName } from './character';

export const SKILL\_ABILITY: Record<SkillName, AbilityName> = {

Acrobatics: 'DEX',

'Animal Handling': 'WIS',

Arcana: 'INT',

Athletics: 'STR',

Deception: 'CHA',

History: 'INT',

Insight: 'WIS',

Intimidation: 'CHA',

Investigation: 'INT',

Medicine: 'WIS',

Nature: 'INT',

Perception: 'WIS',

Performance: 'CHA',

Persuasion: 'CHA',

Religion: 'INT',

'Sleight of Hand': 'DEX',

Stealth: 'DEX',

Survival: 'WIS',

};

**2) Core: character skill helpers**

Update packages/core/src/character.ts:

* Extend CharacterProficiencies to allow **expertise**:

export interface CharacterProficiencies { weapons?: { simple?: boolean; martial?: boolean }; saves?: AbilityName[]; skills?: SkillName[]; *// proficiency* expertise?: SkillName[]; *// double PB on these skills* }

* Add helpers:

import { SKILL\_ABILITY } from './skills'; import { abilityCheck } from './checks'; export function skillAbility(skill: SkillName): AbilityName { return SKILL\_ABILITY[skill]; } export function isProficientSkill(c: Character, skill: SkillName): boolean { return !!c.proficiencies?.skills?.includes(skill); } export function hasExpertise(c: Character, skill: SkillName): boolean { return !!c.proficiencies?.expertise?.includes(skill); } export function characterSkillCheck( c: Character, skill: SkillName, opts?: { dc?: number; advantage?: boolean; disadvantage?: boolean; seed?: string; extraMod?: number } ) { const ability = skillAbility(skill); const mods = abilityMods(c.abilities); const base = mods[ability] ?? 0; const pb = proficiencyBonusForLevel(c.level); const prof = isProficientSkill(c, skill); const expert = hasExpertise(c, skill); const profBonus = prof ? (expert ? pb \* 2 : pb) : 0; const modifier = base + profBonus + (opts?.extraMod ?? 0); return abilityCheck({ ability, modifier, proficient: false, *// already applied above* proficiencyBonus: 0, advantage: opts?.advantage, disadvantage: opts?.disadvantage, dc: opts?.dc, seed: opts?.seed, }); } *// Passive Perception = 10 + Perception modifier (ability + proficiency/expertise + extras)* export function passivePerception(c: Character, extraMod = 0): number { const ability = 'Perception' as const; const mods = abilityMods(c.abilities); const base = mods['WIS'] ?? 0; const pb = proficiencyBonusForLevel(c.level); const prof = isProficientSkill(c, ability); const expert = hasExpertise(c, ability); const profBonus = prof ? (expert ? pb \* 2 : pb) : 0; return 10 + base + profBonus + extraMod; }

* Update character show usage in CLI (below) to display Passive Perception.

**3) Tests**

Add packages/core/tests/skills.test.ts with Vitest:

* Mapping sanity:
  + skillAbility('Athletics') === 'STR'
  + skillAbility('Stealth') === 'DEX'
  + skillAbility('Perception') === 'WIS'
* Proficiency / expertise math:
  + Character level 5 (PB +3), WIS 14 (+2):
    - Perception (no prof): mod +2
    - Perception (prof): mod +2 +3 = +5
    - Perception (expertise): mod +2 +6 = +8
* characterSkillCheck determinism with seed.
* passivePerception:
  + With examples above: 12 / 15 / 18 respectively (without extraMod).

**4) CLI**

Update apps/cli/index.ts:

* In character show, append:
* Passive Perception: <value>

(use passivePerception(loadedCharacter))

* Add a new subcommand:
* character skill "<SkillName>" [--dc N] [--adv|--dis] [--seed S] [--extraMod N]

Implementation:

* + Validate skill name against the known list (case-insensitive match to canonical).
  + Call characterSkillCheck.
  + Output example:
  + Skill Check: Stealth (DEX) vs DC 14
  + Mods: base +2, PB +2 // or PB +4 if expertise
  + Rolls: [7, 18] → total 22
  + Result: SUCCESS
* Add character skills to **list all skills** and their governing abilities (just a pretty print from SKILL\_ABILITY).

Parsing notes:

* For proficiency/expertise in this feature, rely on what’s in the loaded character JSON:

{ "proficiencies": { "skills": ["Athletics", "Perception"], "expertise": ["Stealth"] } }

* Keep argument parsing lightweight, like other commands.

**Acceptance Criteria**

* pnpm test passes including skills.test.ts.
* character show prints Passive Perception: N.
* character skills lists all skills with their abilities.
* character skill "<SkillName>" --dc N resolves check using the correct ability and PB/expertise.
* Seeded runs are deterministic; advantage/disadvantage applies.

---

When this lands, a nice \*\*Feature 8\*\* could be \*\*simple proficiency sources\*\* (e.g., infer weapon/skill save profs from a tiny “class at level” table in rules-srd, still SRD-compliant) or \*\*saving/loading characters\*\* to a `.data/characters/` folder with names. Your call!

**You said:**

Uploaded a file

looks good to me

**ChatGPT said:**

Perfect ✅

From your latest run:

* **All tests passed** (42/42, now including skills).
* character show → displays **Passive Perception: 12**.
* character skills → lists all 18 skills mapped to their abilities.
* character skill "Perception" --dc 15 → works, applies WIS + PB correctly → rolled 14 → total 16 → SUCCESS.
* The earlier error (--dc N) was just because N wasn’t an integer — parsing is strict by design.

So **Feature 7 is in and stable** 🎉

**🚀 Next logical milestone (Feature 8)**

We can go two directions:

1. **Character Data Expansion**
   * Load more from the SRD: race/class → derive proficiency bonus, saves, skills, and weapon profs automatically instead of JSON manual listing.
   * Store characters in .data/characters/ by name for quick load/save.
2. **Combat Encounters**
   * Add an “encounter” engine: multiple creatures, initiative order, turn loop, actions (attack, save, skill check).
   * CLI: encounter start, encounter add goblin, encounter next, etc.

👉 Do you want Feature 8 to be **character expansion (more automation from SRD)** or **first step toward encounters**?

**You said:**

take your pick

**ChatGPT said:**

Thought for 16s

Love it—let’s kick off the gameplay loop. I’m picking **Encounters** for Feature 8.

Here’s a clean PR prompt you can paste into Codex to generate the next PR.

**🔖 Prompt for Codex → PR #8**

**Title:** feat(core): minimal encounter engine (initiative, turns, actors, attacks) + CLI

**PR Body:**

Add a minimal encounter engine to @grimengine/core with:

- Actors (PCs from Character, simple Monsters from a tiny SRD adapter)

- Initiative roll & turn order

- Turn progression (next/round tracking)

- Basic actions that reuse existing checks/combat (attack, save/check hooks)

- CLI to run a simple combat loop

Keep this lean and deterministic-friendly; we’ll expand later.

*## Files & Structure*

*### 1) Core encounter model*

Create `packages/core/src/encounter.ts`:

```ts

import type { AbilityName } from './abilityScores';

import type { ResolveAttackResult } from './combat';

import { attackRoll, damageRoll, resolveAttack } from './combat';

export type Side = 'party' | 'foe';

export interface ActorBase {

id: string; *// unique within encounter*

name: string;

side: Side;

ac: number;

hp: number;

maxHp: number;

abilityMods: Partial<Record<AbilityName, number>>; *// STR/DEX etc.*

proficiencyBonus?: number; *// default 2*

}

export interface WeaponProfile {

name: string; *// e.g. "Longsword"*

attackMod: number; *// total to-hit modifier (ability + prof etc.)*

damageExpr: string; *// e.g. "1d8+3"*

versatileExpr?: string; *// e.g. "1d10+3" when two-handed (optional)*

}

export interface MonsterActor extends ActorBase {

type: 'monster';

attacks: WeaponProfile[]; *// choose index 0 by default*

}

export interface PlayerActor extends ActorBase {

type: 'pc';

*// optional default weapon (if set, use it when attacking without explicit weapon)*

defaultWeapon?: WeaponProfile;

}

export type Actor = MonsterActor | PlayerActor;

export interface InitiativeEntry {

actorId: string;

rolled: number; *// the d20 result used*

total: number; *// rolled + Dex mod (if provided)*

}

export interface EncounterState {

id: string;

seed?: string;

round: number; *// starts at 1 when initiative set*

turnIndex: number; *// 0-based index into initiative order*

order: InitiativeEntry[]; *// descending by total*

actors: Record<string, Actor>;

defeated: Set<string>;

}

export function createEncounter(seed?: string): EncounterState;

export function addActor(state: EncounterState, actor: Actor): EncounterState;

export function removeActor(state: EncounterState, actorId: string): EncounterState;

*/\*\* Roll initiative for each actor: d20 + DEX mod (if present). Deterministic with seed if set. \*/*

export function rollInitiative(state: EncounterState): EncounterState;

*/\*\* Advance to next turn; increment round when wrapping. Skips defeated actors. \*/*

export function nextTurn(state: EncounterState): EncounterState;

*/\*\* Get the current actor on turn (or null if none). \*/*

export function currentActor(state: EncounterState): Actor | null;

*/\*\**

*\* Perform a basic attack from attacker -> defender.*

*\* If attacker has a WeaponProfile (pc.defaultWeapon or monster.attacks[0]):*

*\* - build attack using attackMod and damageExpr (resolve versatileExpr if `twoHanded`).*

*\* - on hit (or crit), apply damage to defender; clamp hp >= 0; mark defeated when hp=0.*

*\* Return both the attack result and the damage (if hit).*

*\*/*

export function actorAttack(

state: EncounterState,

attackerId: string,

defenderId: string,

opts?: { twoHanded?: boolean; advantage?: boolean; disadvantage?: boolean; seed?: string }

): { attack: ResolveAttackResult['attack']; damage?: ResolveAttackResult['damage']; defenderHp: number };

Implementation notes:

* Use existing attackRoll/damageRoll/resolveAttack. For this feature, use the weapon profile’s  
  numeric attackMod directly (we’ll auto-derive from real weapons later).
* Initiative tiebreaker: higher DEX mod wins; if still tied, compare names; if still tied, stable.
* nextTurn skips defeated actors and wraps to round+1.
* All functions should be pure and return new state objects (no mutation in place).

**2) Tiny SRD monster adapter**

Create packages/adapters/rules-srd/src/monsters.ts:

import type { MonsterActor } from '@grimengine/core/src/encounter'; export const MONSTERS: Omit<MonsterActor, 'id' | 'side'>[] = [ { type: 'monster', name: 'Goblin', ac: 15, hp: 7, maxHp: 7, abilityMods: { DEX: 2, STR: -1 }, proficiencyBonus: 2, attacks: [ { name: 'Scimitar', attackMod: 4, damageExpr: '1d6+2' }, { name: 'Shortbow', attackMod: 4, damageExpr: '1d6+2' } ] }, { type: 'monster', name: 'Bandit', ac: 12, hp: 11, maxHp: 11, abilityMods: { DEX: 1, STR: 1 }, proficiencyBonus: 2, attacks: [ { name: 'Scimitar', attackMod: 3, damageExpr: '1d6+1' }, { name: 'Light Crossbow', attackMod: 3, damageExpr: '1d8+1' } ] } ]; export function getMonsterByName(name: string) { return MONSTERS.find(m => m.name.toLowerCase() === name.toLowerCase()); }

*(Values are SRD-ish approximations sufficient for testing; we’ll refine later.)*

**3) Tests**

Add packages/core/tests/encounter.test.ts (Vitest):

* Create encounter with seed; add two goblins (foe) and one pc (party).
* Roll initiative: verify ordering (deterministic by seed).
* currentActor returns the first in order; nextTurn advances; wraps to round 2 when cycling.
* Attack resolution:
  + From goblin → pc: on seeded hit, defender hp decreases correctly.
  + On crit, damage dice double (assert via high result with seed).
  + When hp reaches 0, actor is marked defeated and is skipped on turns.
* Removing an actor updates the order appropriately.

Use fixed seeds for stable tests.

**4) CLI**

Update apps/cli/index.ts with an encounter group:

Commands:

1. encounter start [--seed S]
   * Creates a session encounter at .data/encounters/current.json
   * Output: Encounter started (seed="S").
2. encounter add pc "<name>"
   * Requires a loaded character session (from Feature 6).
   * Converts to an Actor using: AC = 10 + DEX mod (simple placeholder), HP = 12 by default (placeholder),  
     abilityMods and PB from character, defaultWeapon inferred from characterWeaponAttack inputs is *not* required here;  
     for now, set defaultWeapon to { name: 'Unarmed', attackMod: abilityMods.STR ?? 0 + PB, damageExpr: '1d4+<STRmod>' }.

encounter add monster "<name>" [--count N]

* + Adds N copies of the SRD monster (names as Goblin #1, Goblin #2), side=foe.

1. encounter list
   * Prints round, turn pointer, and actors with hp/ac and side, marking defeated.
2. encounter roll-init
   * Rolls initiative: d20 + DexMod.
   * Prints the order.
3. encounter next
   * Advances the turn; prints the new current actor.
4. encounter attack "<attackerIdOrName>" "<defenderIdOrName>" [--adv|--dis] [--twohanded] [--seed S]
   * Uses attacker’s weapon profile (first attack for monsters; defaultWeapon for PCs).
   * Prints attack expression, d20 rolls, natural, total, hit/miss/crit, and damage if any, plus defender’s new HP.
5. encounter end
   * Clears the encounter session file.

Parsing notes:

* Store encounter state in .data/encounters/current.json (ensure .data is in .gitignore—already is).
* Resolve attackerIdOrName by exact id first, then by case-insensitive name match if unique.
* For now, simple placeholders for PC AC/HP are acceptable (we’ll wire real data later).

**Acceptance Criteria**

* pnpm test passes including encounter.test.ts.
* CLI happy path:
* pnpm dev *-- encounter start --seed grim*
* pnpm dev *-- character load ".\Characters\Aerin.json"*
* pnpm dev *-- encounter add pc "Aerin"*
* pnpm dev *-- encounter add monster "Goblin" --count 2*
* pnpm dev *-- encounter roll-init*
* pnpm dev *-- encounter list*
* pnpm dev *-- encounter next*
* pnpm dev *-- encounter attack "Goblin #1" "Aerin" --seed grim*
* pnpm dev *-- encounter list*
* pnpm dev *-- encounter end*
* Turn order advances correctly; defeated actors are skipped.
* Attack/damage behavior matches existing combat helpers (crits double dice only).

---

If you’re happy with this direction, I can also prep \*\*Feature 9\*\* to make PCs’ AC/HP/attacks less “placeholder” by deriving \*\*AC, HP, and default weapon profile from the loaded character\*\* (and from the weapon rules you already added).

**You said:**

Uploaded a file

looks good to me. we can mov eon if you agree

**ChatGPT said:**

Thought for 12s

Killer—Feature 8 looks solid in your logs. Let’s keep the momentum with **Feature 9: derive PCs’ AC/HP/attack from character + equipment** so encounters use real stats (no placeholders).

Below is a clean PR prompt you can drop into Codex.

**🔖 Prompt for Codex → PR #9**

**Title:** feat(core): derive PC AC/HP/attack from character + equipment; armor SRD adapter; wire into encounters

**PR Body:**

Replace placeholder PC AC/HP/attack in encounters with values derived from the loaded character

(abilities, level, prof bonus) and equipped gear. Add a tiny SRD armor/shield list and equip fields

to character JSON. Make CLI show derived stats and use them when adding PCs to an encounter.

## Scope

- No inventory system yet; just a simple "equipped" section in the character JSON.

- Armor rules: Unarmored, Light, Medium, Heavy, plus Shield.

- HP: max HP from hit die at level 1 (full) + average per level thereafter, plus Con mod per level.

- Default weapon for attack: chosen by name from rules-srd weapons (already present).

## Files & Changes

### 1) SRD Armor adapter

Create `packages/adapters/rules-srd/src/armor.ts`:

```ts

export type ArmorCategory = 'unarmored' | 'light' | 'medium' | 'heavy' | 'shield';

export interface Armor {

name: string;

category: Exclude<ArmorCategory, 'shield'>;

baseAC: number;

dexCap?: number; // e.g. medium: max +2; light: unlimited; heavy: 0

}

export interface Shield {

name: string; // e.g. "Shield"

bonusAC: number; // +2

}

export const ARMORS: Armor[] = [

{ name: 'Padded', category: 'light', baseAC: 11 }, // +DEX (no cap)

{ name: 'Leather', category: 'light', baseAC: 11 },

{ name: 'Studded Leather', category: 'light', baseAC: 12 },

{ name: 'Hide', category: 'medium', baseAC: 12, dexCap: 2 }, // +DEX up to +2

{ name: 'Chain Shirt', category: 'medium', baseAC: 13, dexCap: 2 },

{ name: 'Scale Mail', category: 'medium', baseAC: 14, dexCap: 2 },

{ name: 'Breastplate', category: 'medium', baseAC: 14, dexCap: 2 },

{ name: 'Half Plate', category: 'medium', baseAC: 15, dexCap: 2 },

{ name: 'Ring Mail', category: 'heavy', baseAC: 14, dexCap: 0 }, // no DEX

{ name: 'Chain Mail', category: 'heavy', baseAC: 16, dexCap: 0 },

{ name: 'Splint', category: 'heavy', baseAC: 17, dexCap: 0 },

{ name: 'Plate', category: 'heavy', baseAC: 18, dexCap: 0 },

];

export const SHIELD: Shield = { name: 'Shield', bonusAC: 2 };

export function getArmorByName(name: string) {

return ARMORS.find(a => a.name.toLowerCase() === name.toLowerCase());

}

**2) Character: equipped + derived stats**

Update packages/core/src/character.ts:

* Extend model:

export interface Equipped { armor?: string; *// e.g. "Chain Mail" or undefined for unarmored* shield?: boolean; *// true/false* weapon?: string; *// name, must exist in rules-srd weapon list* hitDie?: 'd6' | 'd8' | 'd10' | 'd12'; *// class hit die (for HP calc)* } export interface Character { *// existing fields...* equipped?: Equipped; }

* Add helpers:

import { getArmorByName, SHIELD } from '@grimengine/rules-srd/src/armor'; import { getWeaponByName } from '@grimengine/rules-srd/src/weapons'; import { chooseAttackAbility } from './weapons'; export function derivedAC(c: Character): number { const dexMod = abilityMod(c.abilities.DEX); const armorName = c.equipped?.armor; if (!armorName) return 10 + dexMod + (c.equipped?.shield ? SHIELD.bonusAC : 0); const armor = getArmorByName(armorName); if (!armor) return 10 + dexMod + (c.equipped?.shield ? SHIELD.bonusAC : 0); const dexPart = armor.category === 'light' ? dexMod : armor.category === 'medium' ? Math.min(dexMod, armor.dexCap ?? 2) : 0; *// heavy* const shield = c.equipped?.shield ? SHIELD.bonusAC : 0; return armor.baseAC + dexPart + shield; } export function derivedMaxHP(c: Character): number { *// Hit Die required. If missing, default d8.* const die = c.equipped?.hitDie ?? 'd8'; const dieMax = { d6: 6, d8: 8, d10: 10, d12: 12 }[die]; const con = abilityMod(c.abilities.CON); *// Level 1: full hit die + CON* let hp = dieMax + con; *// Levels 2..N: average per level (rounded up) + CON each level* const avg = Math.ceil((dieMax + 1) / 2); for (let lvl = 2; lvl <= c.level; lvl++) { hp += avg + con; } return Math.max(hp, 1); } export function derivedDefaultWeaponProfile(c: Character): { name: string; attackMod: number; damageExpr: string; versatileExpr?: string; } | null { const name = c.equipped?.weapon; if (!name) return null; const w = getWeaponByName(name); if (!w) return null; const mods = abilityMods(c.abilities); const ability = chooseAttackAbility(w, mods); const abilMod = mods[ability] ?? 0; const pb = proficiencyBonusForLevel(c.level); const proficient = c.proficiencies?.weapons?.[w.category] === true; const attackMod = abilMod + (proficient ? pb : 0); *// Base or versatile damage expression gets ability mod added.* const withMod = (expr: string) => { *// append +X or -X (assume expressions from adapter have +0)* const sign = abilMod >= 0 ? `+${abilMod}` : `${abilMod}`; return expr.replace(/\+0\b/, sign); }; const damageExpr = withMod(w.damage.expression); const versatileExpr = w.versatile ? withMod(w.versatile.expression) : undefined; return { name: w.name, attackMod, damageExpr, versatileExpr }; }

**3) Encounter: use derived stats for PCs**

Update packages/core/src/encounter.ts and/or the CLI layer:

* When adding a **PC** to an encounter (via CLI), compute:
  + ac = derivedAC(character)
  + hp = derivedMaxHP(character) and maxHp same at add-time
  + defaultWeapon = derivedDefaultWeaponProfile(character) ?? { name: 'Unarmed', attackMod: abilityMods.STR+PB, damageExpr: '1d4+<STRmod>' }

No change needed for monsters.

**4) CLI changes**

* In apps/cli/index.ts:
  + character show: append a “Derived” section:
  + Derived:
  + AC: <n> Max HP: <n>
  + Default Weapon: <name or none> (to-hit +X; damage <expr>[, versatile <expr>])
  + encounter add pc "<name>" now uses the derived AC/HP/defaultWeapon when creating the PC actor.
  + Add a helper command:
  + character derive

which prints AC, Max HP, and Default Weapon without needing to start an encounter.

* Optional: character load error if equipped.hitDie missing; instead just default to d8 and mention it in output.

**5) Tests**

Add packages/core/tests/derived.test.ts (Vitest):

* AC cases:
  + Unarmored DEX +2 => AC 12; with shield => 14
  + Medium armor (14 base, max +2 DEX) with DEX +3 => 16
  + Heavy armor ignores DEX
* HP cases:
  + Level 1 d10, CON +2 => 12
  + Level 3 d10, CON +2 => 12 + 2×(6 + 2) = 28 (avg for d10 is 6)
* Default weapon profile:
  + Character level 3, martial proficient, STR +3 → Longsword attackMod +5, damage 1d8+3, versatile 1d10+3
  + Non-proficient keeps only ability mod
* Use deterministic seeds only where needed.

**Acceptance Criteria**

* pnpm test passes including derived.test.ts.
* character show prints Derived block with correct AC/MaxHP/default weapon from equipped.
* character derive prints the same values.
* encounter add pc "<name>" uses derived AC/HP/weapon; visible in encounter list.
* End-to-end: with a character equipped with, e.g., Chain Mail + shield and a Longsword,  
  encounter attack shows correct to-hit and damage expressions from the derived weapon profile.

---

If that sounds good, the next step after *#9 could be \*\*Feature 10: local save/load of named characters\*\* (`.data/characters/<Name>.json`), plus `character save` and `character list` for quick swapping.*

**You said:**

codex is still churning, but could you maybe (for this and all future prs) just give me an exact list of suggested commands to run for testing

**ChatGPT said:**

Got it 👍 — from here on, I’ll always give you a **ready-to-run command checklist** for each PR so you can confirm quickly that everything works.

Here’s what to run once PR **#9** (derived AC/HP/weapon) lands:

**🧪 Test Commands for PR #9**

**1. Run all tests**

pnpm test

✅ Expect: new derived.test.ts passes along with existing ones.

**2. Load and show a character**

(assuming your Characters\Aerin.json is updated with "equipped" fields like armor, shield, weapon, hitDie)

pnpm dev -- character load ".\Characters\Aerin.json" pnpm dev -- character show

✅ Expect:

* New **Derived** section with AC, Max HP, Default Weapon (to-hit +X; damage exprs).

**3. Derive-only command**

pnpm dev -- character derive

✅ Expect: same derived AC / HP / weapon as in show.

**4. Encounter add PC (check derived stats flow through)**

pnpm dev -- encounter start --seed grim pnpm dev -- character load ".\Characters\Aerin.json" pnpm dev -- encounter add pc "Aerin" pnpm dev -- encounter list

✅ Expect: PC shows with derived AC, HP (Max HP), and weapon (to-hit +X; damage expr).

**5. Attack test with derived weapon**

pnpm dev -- encounter add monster "Goblin" pnpm dev -- encounter roll-init pnpm dev -- encounter attack "Aerin" "Goblin #1" --seed grim

✅ Expect:

* Attack expression uses correct ability mod + proficiency bonus.
* Damage expression matches derived weapon (e.g. longsword 1d8+3 or 1d10+3 if two-handed).
* Goblin HP decreases appropriately.

**6. Clean up**

pnpm dev -- encounter end pnpm dev -- character unload

👉 Once you’ve confirmed these, we’ll be set to move on to Feature 10 (character save/list/load from .data/characters).

Want me to go ahead and prep the PR + test command list for Feature 10 now so you can keep the chain rolling?

**You said:**

Uploaded a file

Looks good to me. we can move on if you agree

**ChatGPT said:**

Thought for 15s

Yesss—everything in your latest run looks spot-on. Let’s keep shipping. I propose **Feature 10: Character vault + quick equip** so you can save/list/load named characters locally and tweak equipment without editing JSON.

Below is the PR prompt for Codex **and** the exact test commands.

**🔖 Prompt for Codex → PR #10**

**Title:** feat(cli): character vault (save/list/load by name) + quick equip/set helpers

**PR Body:**

Add a simple local "character vault" so users can save, list, and load characters by name,

and add quick helpers to adjust equipment & common fields from the CLI (no manual JSON edit).

## Scope

- Local-only, stored under `.data/characters/<Name>.json`.

- Non-destructive: saving overwrites only that file; session still required for encounter use.

- Minimal "equip" & "set" helpers: armor, shield, weapon, hitDie, level.

## Files & Changes

### 1) CLI vault utilities

Create `apps/cli/char-vault.ts`:

```ts

import { mkdirSync, readFileSync, writeFileSync, existsSync, readdirSync } from 'node:fs';

import { join } from 'node:path';

import type { Character } from '@grimengine/core/src/character';

const ROOT = join(process.cwd(), '.data', 'characters');

export function ensureVault() { mkdirSync(ROOT, { recursive: true }); }

export function characterPath(name: string) { return join(ROOT, `${name}.json`); }

export function saveToVault(name: string, c: Character) {

ensureVault();

writeFileSync(characterPath(name), JSON.stringify(c, null, 2), 'utf-8');

}

export function loadFromVault(name: string): Character | null {

try {

const p = characterPath(name);

if (!existsSync(p)) return null;

return JSON.parse(readFileSync(p, 'utf-8')) as Character;

} catch { return null; }

}

export function listVaultNames(): string[] {

ensureVault();

return readdirSync(ROOT)

.filter(f => f.toLowerCase().endsWith('.json'))

.map(f => f.replace(/\.json$/i, ''));

}

**2) CLI: new subcommands in**apps/cli/index.ts

Add under the character command group:

* character save --name "<Name>"  
  Saves the **currently loaded** character to .data/characters/<Name>.json.  
  Output: Saved character "<Name>" to vault.  
  If no session loaded → error.
* character list  
  Lists all names in the vault.  
  Output: Characters: Aerin, Borin, ... (or No saved characters.)
* character load-name "<Name>"  
  Loads from the vault into the **session store** (same effect as character load <path>).  
  Output: Loaded character <Name> (lvl N). PB +X
* character equip [--armor "<ArmorName>"] [--shield on|off] [--weapon "<WeaponName>"] [--hitdie d6|d8|d10|d12]  
  Mutates the **session character** in memory and rewrites the session file.  
  Output lines for each changed field; then print the new Derived block (AC / Max HP / Default weapon).  
  If no session loaded → error.  
  Armor must match the SRD adapter names; weapon must exist in rules-srd weapons list.
* character set [--level N]  
  Sets common scalar fields. For this PR, only level is required.  
  After setting, reprint PB and Derived block.

Implementation notes:

* Reuse existing session.ts helpers (saveCharacter, loadCharacter) to update the session after equip/set.
* For shield: accept --shield, --shield on, --shield off (treat absent as no change).
* Leave skills/proficiencies editing for a later feature.

**3) Acceptance Criteria**

* character save --name "Aerin" writes .data/characters/Aerin.json.
* character list shows the saved names.
* character load-name "Aerin" restores into session; character show reflects it.
* character equip --armor "Chain Mail" --shield on --weapon "Longsword" --hitdie d10 updates session and prints new Derived (e.g., AC 18 or similar depending on DEX).
* character set --level 5 updates PB accordingly and prints Derived.
* All existing commands continue to work.

*---*

## 🧪 Test Commands for PR #10

Run these in order (copy/paste friendly):

```powershell

# 1) Tests

pnpm test

# 2) Load your existing JSON (session)

pnpm dev *-- character load ".\Characters\Aerin.json"*

pnpm dev *-- character show*

# 3) Save to vault, list, and load by name

pnpm dev *-- character save --name "Aerin"*

pnpm dev *-- character list*

pnpm dev *-- character load-name "Aerin"*

pnpm dev *-- character show*

# 4) Quick equip (update in session)

pnpm dev *-- character equip --armor "Chain Mail" --shield on --weapon "Longsword" --hitdie d10*

pnpm dev *-- character show*

pnpm dev *-- character derive*

# 5) Level bump and verify PB/derived change

pnpm dev *-- character set --level 5*

pnpm dev *-- character show*

# 6) Encounter sanity (should use derived defaults)

pnpm dev *-- encounter start --seed grim*

pnpm dev *-- encounter add pc "Aerin"*

pnpm dev *-- encounter list*

pnpm dev *-- encounter add monster "Goblin"*

pnpm dev *-- encounter roll-init*

pnpm dev *-- encounter attack "Aerin" "Goblin #1" --seed grim*

pnpm dev *-- encounter end*

# 7) Cleanup

pnpm dev *-- character unload*

If you want the next one queued up after #10, I’d suggest **Feature 11: monster lookup from the public 5e API with on-disk caching**, but we can also keep strengthening the CLI (e.g., character prof add/remove, character skill add/remove, or encounter save/load). Your call!

**You said:**

Uploaded a file

seems good to me

**ChatGPT said:**

Beautiful — your logs confirm:

* ✅ **All 52 tests passed**.
* ✅ character equip → correctly derived **AC 18, Max HP 25 → 39 after level bump, Longsword to-hit +6 / damage 1d8+3**.
* ✅ character save/list/load-name works as intended.
* ✅ encounter add pc → pulled derived AC/HP/weapon into the encounter actor.
* ✅ encounter attack → applied the +6 attack mod, hit AC 15, rolled damage 1d8+3, and defeated the goblin in one swing.
* ✅ Session cleared with encounter end + character unload.

That means **Feature 10 is locked in** and the vault + quick equip pipeline works exactly as planned. 🎉

**🚀 Suggested Feature 11**

Now that PCs are fully fleshed out, the next logical step is to enrich monsters beyond the static list. My suggestion:

**Feature 11 — Monster Lookup + Caching**

* Integrate the free 5e API (https://www.dnd5eapi.co) to fetch monster data on demand.
* Cache JSON responses to .data/cache/monsters/.
* Add CLI:
  + monster fetch "<name>" (pull into cache)
  + monster list (cached monsters)
  + encounter add monster "<name>" auto-fetches if missing.

Would you like me to prep the **PR prompt + test command list** for Feature 11 (monster API + caching), or would you rather focus next on **encounter save/load** (so you can resume fights later)?

**You said:**

yeah let's do it

**ChatGPT said:**

Awesome — here’s the **PR prompt for Feature 11 (Monster API + caching)** and the **exact commands to run** to verify it.

**🔖 Prompt for Codex → PR #11**

**Title:** feat(adapters): 5e monster fetch + on-disk cache; CLI fetch/list/show; encounter auto-fetch

**PR Body:**

Add a tiny adapter for fetching monsters from the public 5e API (https://www.dnd5eapi.co),

cache them on disk, normalize to our Encounter MonsterActor shape, and wire basic CLI:

- `monster fetch "<name>"`

- `monster list`

- `monster show "<name>"`

Also make `encounter add monster "<name>"` auto-fetch if not present in cache.

*## Scope / Constraints*

- \*\*No network in tests\*\*: tests use a local fixture JSON to exercise normalization.

- Cache path: `.data/cache/monsters/<slug>.json`

- "Slug" rule: lowercase, replace spaces with `-` (e.g., "Goblin" -> `goblin`).

- Normalization targets our `MonsterActor` (encounter.ts) with a best-effort first attack profile.

## Files & Changes

### 1) Adapter (fetch + cache + normalize)

Create `packages/adapters/dnd5e-api/src/monsters.ts`:

```ts

import { mkdirSync, readFileSync, writeFileSync, existsSync } from 'node:fs';

import { join } from 'node:path';

import { request } from 'undici';

import type { MonsterActor, WeaponProfile } from '@grimengine/core/src/encounter';

const ROOT = join(process.cwd(), '.data', 'cache', 'monsters');

export function slugify(name: string) {

return name.trim().toLowerCase().replace(/\s+/g, '-');

}

export function cachePath(name: string) {

return join(ROOT, `${slugify(name)}.json`);

}

export function readCachedMonster(name: string): any | null {

const p = cachePath(name);

if (!existsSync(p)) return null;

return JSON.parse(readFileSync(p, 'utf-8'));

}

export async function fetchMonsterFromAPI(name: string): Promise<any> {

const slug = slugify(name);

*// Try canonical endpoint first: /api/monsters/<slug>*

const url1 = `https:*//www.dnd5eapi.co/api/monsters/${slug}`;*

let res = await request(url1).catch(() => null);

if (res && res.statusCode === 200) return await res.body.json();

*// Fallback search: /api/monsters?name=<name>*

const url2 = `https:*//www.dnd5eapi.co/api/monsters?name=${encodeURIComponent(name)}`;*

res = await request(url2).catch(() => null);

if (res && res.statusCode === 200) {

const data = await res.body.json();

const first = data?.results?.[0];

if (first?.index) {

const res2 = await request(`https://www.dnd5eapi.co${first.url}`).catch(() => null);

if (res2 && res2.statusCode === 200) return await res2.body.json();

}

}

throw new Error(`Monster not found: ${name}`);

}

export function ensureCacheDir() {

mkdirSync(ROOT, { recursive: true });

}

export function writeCache(name: string, json: any) {

ensureCacheDir();

writeFileSync(cachePath(name), JSON.stringify(json, null, 2), 'utf-8');

}

*/\*\* Best-effort normalization from 5eAPI JSON → MonsterActor \*/*

export function normalizeMonster(name: string, api: any): Omit<MonsterActor, 'id' | 'side'> {

const ac = Array.isArray(api.armor\_class)

? (api.armor\_class[0]?.value ?? api.armor\_class[0] ?? api.armor\_class ?? 10)

: (api.armor\_class ?? 10);

const hp = api.hit\_points ?? 1;

*// DEX/STR ability mods (api uses ability scores 3..30)*

const toMod = (score?: number) => typeof score === 'number' ? Math.floor((score - 10) / 2) : 0;

const abilityMods = {

STR: toMod(api.strength),

DEX: toMod(api.dexterity),

CON: toMod(api.constitution),

INT: toMod(api.intelligence),

WIS: toMod(api.wisdom),

CHA: toMod(api.charisma),

};

const pb = api.prof\_bonus ?? 2; *// many entries lack this; default +2*

*// Try to build a primary WeaponProfile from the first attack-like action.*

*// 5eAPI has `actions`: [{ name, attack\_bonus, damage: [{damage\_dice, damage\_bonus}] }]*

let attacks: WeaponProfile[] = [];

const firstAtk = (api.actions || []).find((a: any) =>

typeof a.attack\_bonus === 'number' || (a.damage && a.damage.length));

if (firstAtk) {

const attackMod = typeof firstAtk.attack\_bonus === 'number'

? firstAtk.attack\_bonus

: (abilityMods.DEX || abilityMods.STR || 0) + pb;

*// Prefer first damage object*

const dmg0 = firstAtk.damage?.[0];

const dice = dmg0?.damage\_dice || '1d4';

const bonus = (typeof dmg0?.damage\_bonus === 'number') ? dmg0.damage\_bonus : (abilityMods.STR || 0);

const signed = bonus >= 0 ? `+${bonus}` : `${bonus}`;

const damageExpr = `${dice}${signed}`;

attacks = [{

name: firstAtk.name || api.name || 'Attack',

attackMod,

damageExpr,

}];

} else {

*// Fallback unarmed-ish profile*

const abil = (Math.max(abilityMods.STR, abilityMods.DEX) === abilityMods.DEX) ? abilityMods.DEX : abilityMods.STR;

attacks = [{ name: 'Attack', attackMod: abil + pb, damageExpr: `1d4+${abil}` }];

}

return {

type: 'monster',

name: api.name || name,

ac,

hp,

maxHp: hp,

abilityMods,

proficiencyBonus: pb,

attacks,

};

}

*/\*\* Main entry: get monster, preferring cache; when fetched, write cache. \*/*

export async function getMonster(name: string): Promise<Omit<MonsterActor, 'id' | 'side'>> {

const cached = readCachedMonster(name);

if (cached) return normalizeMonster(name, cached);

const api = await fetchMonsterFromAPI(name);

writeCache(name, api);

return normalizeMonster(name, api);

}

**2) Core/CLI integration**

* **Tests (no network)**  
  Add packages/core/tests/monster-normalize.test.ts with a local fixture file  
  packages/core/tests/fixtures/goblin.api.json containing a realistic subset from 5eAPI  
  (include name, armor\_class, hit\_points, strength, dexterity, actions[0] with attack\_bonus, damage[0]).  
  The test imports normalizeMonster (re-export it for tests) and asserts:
  + name copied, ac/hp mapped, STR/DEX mods correct,
  + attack profile attackMod matches attack\_bonus,
  + damageExpr matches damage\_dice + damage\_bonus.
* **CLI commands** (apps/cli/index.ts):
  + Add top-level monster group with:
    - monster fetch "<name>" → calls getMonster(name), prints summary, and confirms cached file path.
    - monster list → lists cached slugs (from .data/cache/monsters/\*.json).
    - monster show "<name>" → reads cached (or fetches) and pretty-prints AC/HP/attacks.
  + Update encounter add monster "<name>" [--count N]:
    - Try rules-srd static list first (existing behavior).
    - If not found, call getMonster(name) and add from normalized data.
    - Still support --count.

**3) Dev dependencies**

At repo root:

* Ensure undici is installed (HTTP client).
* Keep tests offline; the adapter’s fetch is only used in CLI code and manual runs.

**4) Acceptance Criteria**

* pnpm test passes (including new monster-normalize.test.ts).
* monster fetch "<name>" hits the API, caches JSON under .data/cache/monsters/, and prints a one-line summary.
* monster list shows cached names (slugs → display name).
* monster show "<name>" prints AC, HP, and first attack profile (to-hit & damage).
* encounter add monster "<name>" auto-fetches and adds if not found in static rules-srd list.
* All existing encounter commands keep working.

---

*## 🧪 Test Commands for PR #11*

Copy/paste in PowerShell once the PR lands:

```powershell

*# 1) Run tests (offline)*

pnpm test

*# 2) Try the new CLI (networked, caches to disk)*

pnpm dev -- monster fetch "Orc"

pnpm dev -- monster fetch "Skeleton"

pnpm dev -- monster list

pnpm dev -- monster show "Skeleton"

*# 3) Use fetched monsters in an encounter*

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add monster "Orc"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

pnpm dev -- encounter list

pnpm dev -- encounter attack "Orc" "Skeleton" --seed grim

pnpm dev -- encounter end

**Expected:**

* Tests all pass (including monster-normalize.test.ts using the local fixture).
* monster fetch prints that it cached each monster and where.
* monster list shows orc, skeleton (and any previously cached like goblin).
* monster show "Skeleton" prints something like:
* Skeleton — AC 13, HP 13
* Attack: Shortbow (+4) → 1d6+2
* Encounter can add those fetched monsters and run attacks as usual.

Want me to queue **Feature 12** next? Options:

* **Encounter save/load** (.data/encounters/<name>.json with encounter save/load/list)
* **Conditions engine** (prone, restrained, poisoned; simple flags that affect rolls)
* **OpenAI/Oracle plug** (optional, with strict cache/budget guard)

**You said:**

Uploaded a file

you'll see I did some different commands in combat but things look good

**ChatGPT said:**

Nice work — your logs confirm Feature 11 is **fully functional** 🎉

* ✅ Tests passed (53/53).
* ✅ monster fetch hits the 5e API, normalizes, and caches Orc/Skeleton correctly.
* ✅ monster list and monster show work as expected.
* ✅ encounter add monster auto-fetches from API and adds them.
* ✅ Initiative order rolled correctly with PC + monsters.
* ✅ Attacks resolved using derived PC stats and monster data — you spotted the detail that you need to include the full name (e.g. "Skeleton #1" instead of "Skeleton"), which matches our design.

So **Feature 11 is locked in** and stable.

**🚀 Suggested Feature 12**

At this point, the next highly valuable capability is **encounter persistence**:

**Feature 12 — Encounter Save/Load**

* Save current encounter state to .data/encounters/<name>.json.
* List saved encounters.
* Load a saved encounter back into session.
* CLI:
  + encounter save "<name>"
  + encounter list-saves
  + encounter load "<name>"
  + encounter delete "<name>"

This will let you pause/resume multi-round fights and keep multiple named encounters.

👉 Want me to prep the **PR prompt + test command list** for Feature 12 (encounter save/load)?

**You said:**

yes

**ChatGPT said:**

Awesome — here’s the **ready-to-paste PR prompt for Feature 12 (Encounter save/load)** plus a **copy-paste test command list**.

**🔖 Prompt for Codex → PR #12**

**Title:** feat(cli): encounter save/load/list/delete (named saves in .data/encounters)

**PR Body:**

Add persistence for encounters so users can pause/resume fights and keep multiple named encounters.

## Scope

- Save the current encounter session to a named JSON file.

- List, load, and delete named saves.

- Keep the existing session file (`.data/encounters/current.json`) as the "active" encounter.

- Saved files live under `.data/encounters/saves/<name>.json`.

## Files & Changes

### 1) Session helpers for encounters

Create `apps/cli/enc-session.ts`:

```ts

import { mkdirSync, readFileSync, writeFileSync, existsSync, rmSync, readdirSync } from 'node:fs';

import { join } from 'node:path';

import type { EncounterState } from '@grimengine/core/src/encounter';

const ROOT = join(process.cwd(), '.data', 'encounters');

const SAVES = join(ROOT, 'saves');

const CURRENT = join(ROOT, 'current.json');

function ensureDirs() {

mkdirSync(ROOT, { recursive: true });

mkdirSync(SAVES, { recursive: true });

}

export function readCurrentEncounter(): EncounterState | null {

if (!existsSync(CURRENT)) return null;

try { return JSON.parse(readFileSync(CURRENT, 'utf-8')) as EncounterState; } catch { return null; }

}

export function writeCurrentEncounter(state: EncounterState) {

ensureDirs();

writeFileSync(CURRENT, JSON.stringify(state, null, 2), 'utf-8');

}

export function clearCurrentEncounter() {

try { if (existsSync(CURRENT)) rmSync(CURRENT); } catch {}

}

export function saveEncounterAs(name: string, state: EncounterState) {

ensureDirs();

const p = join(SAVES, `${name}.json`);

writeFileSync(p, JSON.stringify(state, null, 2), 'utf-8');

return p;

}

export function loadEncounterByName(name: string): EncounterState | null {

const p = join(SAVES, `${name}.json`);

if (!existsSync(p)) return null;

try { return JSON.parse(readFileSync(p, 'utf-8')) as EncounterState; } catch { return null; }

}

export function deleteEncounterByName(name: string): boolean {

const p = join(SAVES, `${name}.json`);

if (!existsSync(p)) return false;

try { rmSync(p); return true; } catch { return false; }

}

export function listEncounterSaves(): string[] {

ensureDirs();

return readdirSync(SAVES)

.filter(f => f.toLowerCase().endsWith('.json'))

.map(f => f.replace(/\.json$/i, ''));

}

**2) Wire into CLI**

Update apps/cli/index.ts under the encounter group to add:

* encounter save "<name>"
  + Reads current encounter (readCurrentEncounter), errors if none.
  + Writes to .data/encounters/saves/<name>.json.
  + Output: Saved encounter as "<name>".
* encounter list-saves
  + Prints saved encounter names (or "No saved encounters.").
* encounter load "<name>"
  + Loads from saves; writes it to current.json as the active encounter.
  + Output: Loaded encounter "<name>" into session.
* encounter delete "<name>"
  + Deletes the named save.
  + Output: Deleted saved encounter "<name>".

Keep existing commands (start, add, list, roll-init, next, attack, end) unchanged.  
Optional nicety: after start, auto-write a blank EncounterState to current.json if not already done.

**3) Acceptance Criteria**

* Can save an active encounter with encounter save "<name>".
* encounter list-saves lists the saved names.
* Can encounter load "<name>" and resume turn order/HP exactly as saved.
* Can encounter delete "<name>" and it disappears from the list.
* Existing features and tests continue to pass.

---

*## 🧪 Test Command List for PR #12*

Copy/paste these after the PR lands:

```powershell

*# 1) Sanity: tests*

pnpm test

*# 2) Start a new encounter and add actors*

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter add monster "Orc"

pnpm dev -- encounter roll-init

pnpm dev -- encounter list

*# 3) Make a couple of attacks to change HP/turn state*

pnpm dev -- encounter attack "Orc #1" "Skeleton #1" --seed grim

pnpm dev -- encounter next

pnpm dev -- encounter list

*# 4) Save the encounter*

pnpm dev -- encounter save "graveyard-1"

pnpm dev -- encounter list-saves

*# 5) End session and verify it's gone*

pnpm dev -- encounter end

pnpm dev -- encounter list *# should say no active encounter or similar*

*# 6) Load the saved one and confirm state is intact*

pnpm dev -- encounter load "graveyard-1"

pnpm dev -- encounter list *# expect same actors/HP/round/turn as before saving*

*# 7) Resume and modify state, then save under a new name*

pnpm dev -- encounter next

pnpm dev -- encounter attack "Skeleton #1" "Orc #1" --seed grim

pnpm dev -- encounter save "graveyard-1b"

pnpm dev -- encounter list-saves

*# 8) Delete the first save and confirm*

pnpm dev -- encounter delete "graveyard-1"

pnpm dev -- encounter list-saves

*# 9) Clean up*

pnpm dev -- encounter end

Want me to prep **Feature 13** next? Two good options:

* **Conditions & simple effects** (prone, restrained, poisoned → advantage/disadvantage hooks, speed notes)
* **Loot/XP stubs** (award XP, generate simple coin/weapon drops, print a summary)

**You said:**

Uploaded a file

This looks good to me. if it's up to me, i'd do the loot and xp stubs net

**ChatGPT said:**

Thought for 14s

Heck yes—let’s add **Loot & XP stubs**. Below is a ready-to-paste PR prompt for Codex **(Feature 13)** plus a copy-paste **test command list**.

**🔖 Prompt for Codex → PR #13**

**Title:** feat(core+cli): loot & XP stubs (coin drops by CR, simple item rolls, XP calc & split)

**PR Body:**

Add a minimal loot + XP system that works with the current encounter engine.

Focus on SRD-friendly stubs: coin piles by CR, optional simple item rolls,

and XP totals/split. Persist results in the active encounter session.

*## Scope*

- Coin drops based on monster CR (lightweight table + dice expressions).

- Optional "simple item" roll that picks from our SRD weapons/armor lists.

- XP per defeated monster using a basic CR→XP map.

- CLI to roll loot for defeated monsters and compute/split XP.

- CLI helper to grant XP to the currently loaded character.

*## Files & Structure*

*### 1) Core: loot model & functions*

Create `packages/core/src/loot.ts`:

```ts

import { roll } from './dice';

/\*\* Minimal CR→XP (5e SRD/DMG values, subset sufficient for testing) \*/

export const CR\_XP: Record<string, number> = {

'0': 10,

'1/8': 25,

'1/4': 50,

'1/2': 100,

'1': 200,

'2': 450,

'3': 700,

'4': 1100,

'5': 1800,

'6': 2300,

'7': 2900,

'8': 3900,

'9': 5000,

'10': 5900

// extend later as needed

};

export type CoinBundle = { cp: number; sp: number; gp: number; pp: number };

export type LootRoll = { coins: CoinBundle; items: string[] };

/\*\* CR→coin dice expressions (very simple stub; expandable) \*/

const CR\_COIN\_EXPR: Record<string, Partial<Record<keyof CoinBundle, string>>> = {

'0': { cp: '1d6x5' },

'1/8': { cp: '1d6x10' },

'1/4': { sp: '1d6x5' },

'1/2': { sp: '1d6x10' },

'1': { gp: '2d6x10' },

'2': { gp: '4d6x10' },

'3': { gp: '5d6x10' },

'4': { gp: '6d6x10' },

'5': { gp: '8d6x10', pp: '1d6x1' },

'6': { gp: '10d6x10', pp: '1d6x2' },

'7': { gp: '12d6x10', pp: '2d6x2' },

'8': { gp: '15d6x10', pp: '2d6x5' },

'9': { gp: '18d6x10', pp: '2d6x10' },

'10': { gp: '20d6x10', pp: '3d6x10' }

};

/\*\* parse "NdM x K" mini-notation; supports "2d6x10" and "1d6" (K=1) \*/

function rollCoins(expr?: string, seed?: string): number {

if (!expr) return 0;

const m = expr.match(/^(\d+)d(\d+)(?:x(\d+))?$/i);

if (!m) return 0;

const n = Number(m[1]), faces = Number(m[2]), mult = Number(m[3] ?? 1);

let sum = 0;

for (let i = 0; i < n; i++) sum += roll(`${1}d${faces}`, { seed: seed ? `${seed}:${i}` : undefined }).total;

return sum \* mult;

}

/\*\* Roll a coin bundle for a given CR string; unknown CRs yield zeros. \*/

export function rollCoinsForCR(cr: string, seed?: string): CoinBundle {

const map = CR\_COIN\_EXPR[cr] || {};

return {

cp: rollCoins(map.cp, seed && `${seed}:cp`),

sp: rollCoins(map.sp, seed && `${seed}:sp`),

gp: rollCoins(map.gp, seed && `${seed}:gp`),

pp: rollCoins(map.pp, seed && `${seed}:pp`),

};

}

/\*\* Compute XP for a monster CR string. Unknown CRs return 0. \*/

export function xpForCR(cr: string): number {

return CR\_XP[cr] ?? 0;

}

/\*\* Sum XP for an array of CRs. \*/

export function totalXP(crs: string[]): number {

return crs.reduce((s, c) => s + xpForCR(c), 0);

}

**2) SRD adapter: simple item picker**

Create packages/adapters/rules-srd/src/loot.ts:

import { WEAPONS } from './weapons'; import { ARMORS } from './armor'; */\*\* Return a simple item name for loot. \*/* export function randomSimpleItem(seed?: string): string { *// naive: 70% weapons, 30% armor* const pick = (arr: string[], idx: number) => arr[idx % arr.length]; const weapons = WEAPONS.map(w => w.name); const armors = ARMORS.map(a => a.name); const bias = (seed && (seed.split('').reduce((a,c)=>a+c.charCodeAt(0),0) % 10)) ?? 0; if (bias < 7) { return pick(weapons, bias); } else { return pick(armors, bias - 7); } }

(We’ll replace this with real treasure tables later.)

**3) Encounter integration (persist loot/xp log)**

Update packages/core/src/encounter.ts:

* Extend EncounterState:

export interface EncounterState { *// existing...* lootLog?: { coins: { cp: number; sp: number; gp: number; pp: number }, items: string[], note?: string }[]; xpLog?: { crs: string[]; total: number }[]; }

* Add utilities:

export function recordLoot(state: EncounterState, entry: { coins: {cp:number;sp:number;gp:number;pp:number}, items: string[], note?: string }): EncounterState; export function recordXP(state: EncounterState, entry: { crs: string[], total: number }): EncounterState;

Both should return new state with appended logs (keep purity).

**4) CLI commands**

Update apps/cli/index.ts with **two encounter subcommands** and **one character helper**:

* encounter loot [--seed S] [--items N] [--note "<text>"]
  + Identify **defeated monsters** in the active encounter.
  + For each defeated monster, get a CR string:
    - First try API JSON cache at .data/cache/monsters/<slug>.json (challenge\_rating), else default '1/2'.
  + Roll coins once per monster (same seed base + :<id> suffix). Sum coin results across all defeated.
  + If --items N provided, roll N simple items via randomSimpleItem(seed + :i).
  + Persist to state via recordLoot.
  + Print a summary:
  + Loot (2 defeated):
  + Coins: 135 gp, 12 sp
  + Items: Longsword
  + Saved to encounter log.
* encounter xp [--party N]
  + Collect CRs for defeated monsters (as above).
  + Compute totalXP(crs) and print:
  + Total XP: 450 (party size 3 ⇒ 150 each)
  + Persist to state via recordXP.
* character add-xp <N>
  + Loads session Character, adds N to a new xp field (create if missing), rewrites session file, and prints new total.
  + (No auto-leveling; we’ll handle that later.)

**5) Tests**

Add packages/core/tests/loot.test.ts:

* xpForCR returns expected values for several CRs.
* totalXP(['1/2','1','2']) sums values correctly.
* rollCoinsForCR('1', seed) returns deterministic numbers > 0 for gp.
* recordLoot and recordXP append to EncounterState in a pure way.

(No network in tests.)

**Acceptance Criteria**

* pnpm test passes including loot.test.ts.
* encounter loot on an encounter with defeated monsters writes a loot entry and prints a summary.
* encounter xp --party 3 prints total and per-character share and writes an xp entry.
* character add-xp 150 updates the loaded character’s xp and prints the new total.
* Existing features/commands remain functional.

---

*## 🧪 Test Command List for PR #13*

```powershell

*# 1) Run tests*

pnpm test

*# 2) Start a quick encounter and make something defeated*

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter add monster "Orc"

pnpm dev -- encounter roll-init

*# Attack until one drops (use seed for determinism)*

pnpm dev -- encounter attack "Orc #1" "Skeleton #1" --seed grim

*# If not defeated yet, attack again or swap attacker/defender as needed*

pnpm dev -- encounter attack "Orc #1" "Skeleton #1" --seed grim

*# 3) Roll loot for defeated monsters (with 1 simple item)*

pnpm dev -- encounter loot --seed grim --items 1 --note "Graveyard stash"

pnpm dev -- encounter list *# (optional) to confirm state still ok*

*# 4) Compute XP and split among 2 (or 3) party members*

pnpm dev -- encounter xp --party 2

*# 5) Add XP to a character (load vault/name as needed)*

pnpm dev -- character load-name "Aerin"

pnpm dev -- character add-xp 150

pnpm dev -- character show *# should display XP total (if you surface it in show)*

*# 6) Clean up*

pnpm dev -- encounter end

pnpm dev -- character unload

If you want, next we can do **Feature 14: conditions & simple effects** (prone/grappled/poisoned → adv/disadv hooks + speed flags), or **Feature 15: SRD spell scaffolding** to prep for casters. Your call!

**You said:**

Uploaded a file

Look sgood to me. THe order you have for the next features is fine.

**ChatGPT said:**

Perfect — your logs confirm Feature 13 is **working end-to-end** ✅

* Tests all passed (58/58).
* Skeleton was dropped to **0 HP** with correct combat math.
* encounter loot rolled coins + simple item (15 sp, Longbow) and persisted to log with note.
* encounter xp --party 2 correctly computed **50 XP → 25 each**, logged it.
* character add-xp 150 updated vault character, and character show displayed XP.
* Fallback messaging worked (unarmed + default d8 hit die).
* Cleanup (encounter end, character unload) succeeded.

So **Loot + XP stubs are locked in** 🎉

**🚀 Next Up (Feature 14)**

As planned, we’ll move on to **Conditions & simple effects** — add support for tracking statuses like *prone*, *restrained*, *poisoned*, and let them affect rolls (adv/disadv) or notes in encounter list.

Do you want me to go ahead and draft the **PR prompt + test commands** for Feature 14 (conditions engine)?

**You said:**

yes

**ChatGPT said:**

Thought for 19s

Awesome — here’s the **ready-to-paste PR prompt for Feature 14 (Conditions & simple effects)** plus a **copy-paste test command list**.

**🔖 Prompt for Codex → PR #14**

**Title:** feat(core+cli): conditions engine (prone/restrained/poisoned/grappled), auto adv/dis on attacks, CLI manage

**PR Body:**

Add a small conditions system to encounters and make it affect attacks:

- Track per-actor conditions in EncounterState

- Auto-apply advantage/disadvantage for common 5e conditions

- CLI to add/remove/list conditions

- Show conditions in encounter list

- Allow attack mode hint (--melee/--ranged) so prone can be adjudicated

Scope is intentionally minimal (no distances, no saves). We only hook attacks.

Ability-check effects (e.g., poisoned) will apply later via encounter-aware checks.

*## Core*

*### 1) Conditions model & helpers*

Create `packages/core/src/conditions.ts`:

```ts

export type Condition = 'prone' | 'restrained' | 'poisoned' | 'grappled';

export interface ConditionSet {

[name: string]: true; *// simple set*

}

export function addCondition(set: ConditionSet | undefined, c: Condition): ConditionSet {

return { ...(set ?? {}), [c]: true };

}

export function removeCondition(set: ConditionSet | undefined, c: Condition): ConditionSet | undefined {

if (!set) return undefined;

const { [c]: \_, ...rest } = set;

return Object.keys(rest).length ? rest : undefined;

}

export function hasCondition(set: ConditionSet | undefined, c: Condition): boolean {

return !!set?.[c];

}

*/\*\* Combine advantage/disadvantage sources into final flags. \*/*

export function combineAdvantage(base: { advantage?: boolean; disadvantage?: boolean },

extra: { advantage?: boolean; disadvantage?: boolean }) {

const adv = !!base.advantage || !!extra.advantage;

const dis = !!base.disadvantage || !!extra.disadvantage;

*// cancel out if both present*

return adv && dis ? { advantage: false, disadvantage: false } : { advantage: adv, disadvantage: dis };

}

*/\*\**

*\* Compute extra adv/dis due to conditions for an attack.*

*\** @param attacker conditions of attacker

\* @param defender conditions of defender

\* @param mode 'melee' | 'ranged' (required to adjudicate prone)

\*/

export function attackAdvFromConditions(attacker: ConditionSet | undefined,

defender: ConditionSet | undefined,

mode: 'melee' | 'ranged'): { advantage?: boolean; disadvantage?: boolean } {

let advantage = false, disadvantage = false;

*// PRONE: attackers within 5 ft have advantage on melee, attackers at range have disadvantage.*

if (hasCondition(defender, 'prone')) {

if (mode === 'melee') advantage = true;

else if (mode === 'ranged') disadvantage = true;

}

*// RESTRAINED: attack rolls against the creature have advantage; restrained creature's attacks have disadvantage.*

if (hasCondition(defender, 'restrained')) advantage = true;

if (hasCondition(attacker, 'restrained')) disadvantage = true;

*// POISONED: creature has disadvantage on attack rolls and ability checks.*

if (hasCondition(attacker, 'poisoned')) disadvantage = true;

*// GRAPPLED: speed 0 only (no direct attack mod here, but kept for listing).*

*// (Future: if prone + grappled, cannot stand.)*

return { advantage, disadvantage };

}

**2) Wire conditions into encounter state & attack flow**

Update packages/core/src/encounter.ts:

* Extend ActorBase:

import type { ConditionSet } from './conditions'; *// ...* conditions?: ConditionSet;

* Add helpers:

import { addCondition, removeCondition } from './conditions'; export function setCondition(state: EncounterState, actorId: string, cond: Condition): EncounterState; export function clearCondition(state: EncounterState, actorId: string, cond: Condition): EncounterState;

Implement purely (return new state with updated actors[actorId].conditions).

* Update actorAttack signature to accept attack mode and auto adv/dis:

export function actorAttack( state: EncounterState, attackerId: string, defenderId: string, opts?: { mode?: 'melee' | 'ranged'; twoHanded?: boolean; advantage?: boolean; disadvantage?: boolean; seed?: string } ): { attack: ResolveAttackResult['attack']; damage?: ResolveAttackResult['damage']; defenderHp: number };

* + Default mode to 'melee' if not provided.
  + Compute extra = attackAdvFromConditions(attacker.conditions, defender.conditions, mode).
  + Combine with user-provided flags via combineAdvantage.
  + Pass the combined flags to resolveAttack.

**3) Show conditions in encounter list**

When printing actors in encounter list, append a bracketed conditions: prone, restrained if present.

**Tests**

Add packages/core/tests/conditions.test.ts:

* addCondition/removeCondition/hasCondition basic behavior.
* attackAdvFromConditions:
  + defender prone + melee ⇒ advantage.
  + defender prone + ranged ⇒ disadvantage.
  + defender restrained ⇒ advantage.
  + attacker restrained ⇒ disadvantage.
  + attacker poisoned ⇒ disadvantage.
  + combinations that produce both ⇒ cancel to normal.
* actorAttack integration:
  + Seeded attack with defender restrained gives advantage (two d20s) and uses the higher.
  + Seeded attack with attacker poisoned yields disadvantage (two d20s) and uses the lower.

(Use small seeded scenarios with a fixed WeaponProfile so results are predictable.)

**CLI**

Update apps/cli/index.ts:

* Add subcommands under encounter:
  1. encounter condition add "<actorIdOrName>" <prone|restrained|poisoned|grappled>
  2. encounter condition remove "<actorIdOrName>" <cond>
  3. encounter condition list
     + Prints each actor’s active conditions (or — if none).
* Extend encounter attack:
  1. Accept --melee or --ranged (default melee).
  2. Auto-apply condition-based adv/dis; still allow --adv/--dis which will combine/cancel as per rules.
  3. Echo the effective mode and whether advantage/disadvantage applied due to conditions:
  4. (conditions: advantage from restrained)

**Acceptance Criteria**

* pnpm test passes including conditions.test.ts.
* encounter condition add/remove/list works and state persists in the session file.
* encounter list shows conditions next to actors.
* encounter attack --melee/--ranged reflects condition-driven adv/dis in output and rolls two d20s when appropriate.
* Existing flows remain functional (weapons, resolveAttack, loot/xp, saves).

---

*## 🧪 Test Command List for PR #14*

Copy/paste after the PR lands:

```powershell

*# 1) Run tests*

pnpm test

*# 2) Start encounter and add combatants*

pnpm dev -- encounter start --seed grim

pnpm dev -- character load-name "Aerin"

pnpm dev -- encounter add pc "Aerin"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

pnpm dev -- encounter list

*# 3) Apply conditions and verify listing*

pnpm dev -- encounter condition add "Skeleton #1" prone

pnpm dev -- encounter condition add "Aerin" poisoned

pnpm dev -- encounter condition list

pnpm dev -- encounter list *# should show conditions next to names*

*# 4) Attack with melee vs prone target (expect advantage)*

pnpm dev -- encounter attack "Aerin" "Skeleton #1" --melee --seed grim

*# Output should mention advantage from conditions and show two d20 rolls (take higher).*

*# 5) Now try ranged vs prone target (expect disadvantage)*

pnpm dev -- encounter attack "Aerin" "Skeleton #1" --ranged --seed grim

*# Should show disadvantage from conditions and two d20 (take lower).*

*# 6) Remove conditions and confirm*

pnpm dev -- encounter condition remove "Aerin" poisoned

pnpm dev -- encounter condition remove "Skeleton #1" prone

pnpm dev -- encounter condition list

*# 7) Add restrained on defender (advantage to attacker)*

pnpm dev -- encounter condition add "Skeleton #1" restrained

pnpm dev -- encounter attack "Aerin" "Skeleton #1" --melee --seed grim

*# 8) Clean up*

pnpm dev -- encounter end

pnpm dev -- character unload

If this feels good, next up (**Feature 15**) I’d suggest **SRD spells scaffold** (spell data adapter + a simple “cast spell” action stub with save DC & damage dice), or we can deepen conditions (poisoned affecting **ability checks** via an encounter-aware check command). Your call!

**You said:**

Uploaded a file

had a test failure

**ChatGPT said:**

Thought for 10s

Thanks for the log — I see exactly what’s happening.

**What broke**

Your new conditions logic is correctly applying advantage/disadvantage, but the **test hard-codes exact d20 faces** ([20, 13]). After we introduced condition-based flags (and likely changed how the seed is threaded), the seeded sequence now returns [14, 2]. The semantics are still right (two rolls, take higher for adv / lower for dis), but the **specific numbers** shifted.

**Two clean ways to fix it**

**Option A — make tests robust (recommended)**

Assert the **behavior**, not the specific faces. This avoids future seed tweaks or minor refactors breaking tests.

**Option B — lock down deterministic seeding**

Standardize sub-seeds inside actorAttack so the same input always yields the same pair of d20s regardless of flag-combine order. (Useful if you truly need frame-perfect reproducibility.)

Below is a tiny PR prompt that does **both** (A + optional B). If you want minimal change, you can keep just the test edits.

**🔖 Hotfix Prompt for Codex →**fix(core): stabilize/relax conditions tests (+ optional stable sub-seed)

**Title:** test(core): relax conditions integration to assert semantics; optionally stabilize sub-seed

**PR Body:**

The conditions integration tests hard-coded exact d20 faces (e.g. [20, 13]),

which became brittle after condition-based adv/dis was introduced. This PR:

1) Updates `conditions.test.ts` to assert semantics:

- two d20s were rolled

- natural = max(d20s) for advantage

- natural = min(d20s) for disadvantage

- expression includes 'adv'/'dis'

2) (Optional but recommended) Stabilizes d20 sub-seeding inside `actorAttack`

so the same seed always rolls the same two d20s, regardless of how flags

were combined. Uses ':atk:0' and ':atk:1' suffixes.

No behavior change to game logic; only test resilience and deterministic seeding.

**Changes:**

1. **Update tests** packages/core/tests/conditions.test.ts — replace hard-coded faces:

*// Advantage case* const adv = actorAttack(encounter, attacker.id, defender.id, { mode: 'melee', seed: 'grim' }); expect(adv.attack.d20s.length).toBe(2); const [r1a, r2a] = adv.attack.d20s; expect(adv.attack.natural).toBe(Math.max(r1a, r2a)); expect(adv.attack.expression).toContain('adv'); *// Disadvantage case* const dis = actorAttack(encounter, attacker.id, defender.id, { mode: 'ranged', seed: 'grim' }); expect(dis.attack.d20s.length).toBe(2); const [r1d, r2d] = dis.attack.d20s; expect(dis.attack.natural).toBe(Math.min(r1d, r2d)); expect(dis.attack.expression).toContain('dis');

Remove expectations like toEqual([20, 13]).

1. **(Optional) Stabilize sub-seed inside**actorAttack packages/core/src/encounter.ts:

*// when building ResolveAttackOptions:* const baseSeed = opts?.seed; const atkSeed0 = baseSeed ? `${baseSeed}:atk:0` : undefined; const atkSeed1 = baseSeed ? `${baseSeed}:atk:1` : undefined; *// if your resolveAttack/attackRoll takes only one seed, ensure internally* *// it uses `:0` and `:1` for the two d20s under adv/dis; otherwise, if you* *// pass the seed directly here, standardize in the dice roller to derive* *// the pair with ':0' and ':1'.*

(If your attackRoll already derives both advantage dice internally, make sure it uses consistent :0/:1 suffixing based on the provided seed. The goal is “same inputs → same pair of faces,” independent of how flags were combined.)

**Acceptance:**

* All tests pass locally; conditions integration asserts semantics rather than exact faces.
* (If optional sub-seed applied) Re-running tests repeatedly yields stable pairs for a given seed.

**🧪 Run these commands now**

# 1) Re-run tests after Codex applies the patch pnpm test # 2) Quick manual spot-check still behaves: pnpm dev -- encounter start --seed grim pnpm dev -- character load-name "Aerin" pnpm dev -- encounter add pc "Aerin" pnpm dev -- encounter add monster "Skeleton" pnpm dev -- encounter roll-init pnpm dev -- encounter condition add "Skeleton #1" restrained pnpm dev -- encounter attack "Aerin" "Skeleton #1" --melee --seed grim # expect "adv" and two d20s pnpm dev -- encounter condition add "Aerin" poisoned pnpm dev -- encounter attack "Aerin" "Skeleton #1" --melee --seed grim # expect "dis" and two d20s pnpm dev -- encounter end pnpm dev -- character unload

If you prefer to lock in the **exact** [20, 13] pair from the original expectation instead, I can help you pin the RNG path precisely — but relaxing the tests to focus on **game semantics** is typically the better long-term move.

**You said:**

Uploaded a file

much better. ready to move forward if you agree

**ChatGPT said:**

Perfect — everything in your run looks **spot-on** ✅

* **All 69 tests passed** after the hotfix, confirming conditions logic and seeding are stable.
* restrained correctly granted **advantage** to Aerin’s melee attack (two d20 rolls → crit).
* poisoned properly applied **disadvantage**, canceled out advantage, and reverted to **normal** rolls.
* Advantage/disadvantage messages, damage, and HP tracking all match 5e expectations.
* Cleanup worked cleanly — encounter and session cleared.

So **Feature 14 (conditions)** is officially locked in. 🎉

**🚀 Next up — Feature 15: SRD Spell Scaffolding**

We’ll add:

* A lightweight SRD spell adapter (name, level, casting\_time, range, components, save, damage).
* A core castSpell utility for basic DC/save/damage resolution.
* CLI:
  + spell fetch "<name>" → from 5e API + cache
  + spell show "<name>"
  + character cast "<spell>" [--target <id|name>] [--seed S] → resolves save/damage

Do you want me to prep the **PR #15 prompt + test-command list** for that next?

**You said:**

yes

**ChatGPT said:**

Thought for 8s

Awesome — here’s the **ready-to-paste PR prompt for Feature 15 (SRD spells scaffold)** plus the **exact test command list** you can run after it lands.

**🔖 Prompt for Codex → PR #15**

**Title:** feat(adapters+core+cli): SRD spells (fetch/cache/normalize) + cast spell (save or attack)

**PR Body:**

Add a lightweight spells system:

- Fetch spells from the public 5e API, cache to disk, and normalize

- Core resolver to cast a spell: save-based (DC) and spell-attack types

- CLI to fetch/list/show spells and to cast from the loaded character against encounter targets

## Scope

- Minimal fields only (enough to resolve a basic cast)

- Save-based spells: target makes a save vs caster DC; "half" or "none" on success

- Spell-attack spells: caster rolls a spell attack vs target AC

- Single-target only for now; AoE/targeting will come later

- Concentration, durations, components are informational only in this PR

## Files & Changes

### 1) Adapter: 5e API spells (fetch/cache/normalize)

Create `packages/adapters/dnd5e-api/src/spells.ts`:

```ts

import { mkdirSync, readFileSync, writeFileSync, existsSync, readdirSync } from 'node:fs';

import { join } from 'node:path';

import { request } from 'undici';

const ROOT = join(process.cwd(), '.data', 'cache', 'spells');

export function slugify(name: string) { return name.trim().toLowerCase().replace(/\s+/g, '-'); }

function pathOf(name: string) { return join(ROOT, `${slugify(name)}.json`); }

export function ensureDir() { mkdirSync(ROOT, { recursive: true }); }

export function readCachedSpell(name: string): any | null {

const p = pathOf(name);

if (!existsSync(p)) return null;

return JSON.parse(readFileSync(p, 'utf-8'));

}

export function writeCachedSpell(name: string, json: any) {

ensureDir();

writeFileSync(pathOf(name), JSON.stringify(json, null, 2), 'utf-8');

}

export function listCachedSpells(): string[] {

ensureDir();

return readdirSync(ROOT).filter(f => f.endsWith('.json')).map(f => f.replace(/\.json$/,''));

}

export async function fetchSpellFromAPI(name: string): Promise<any> {

const idx = slugify(name);

*// try direct endpoint first*

let res = await request(`https:*//www.dnd5eapi.co/api/spells/${idx}`).catch(() => null);*

if (res && res.statusCode === 200) return await res.body.json();

*// fallback search*

res = await request(`https:*//www.dnd5eapi.co/api/spells?name=${encodeURIComponent(name)}`).catch(() => null);*

if (res && res.statusCode === 200) {

const s = await res.body.json();

const first = s?.results?.[0];

if (first?.url) {

const res2 = await request(`https:*//www.dnd5eapi.co${first.url}`).catch(() => null);*

if (res2 && res2.statusCode === 200) return await res2.body.json();

}

}

throw new Error(`Spell not found: ${name}`);

}

*/\*\* Normalized spell shape used by core casting. \*/*

export interface NormalizedSpell {

name: string;

level: number; *// 0 = cantrip*

attackType?: 'ranged' | 'melee'; *// for spell attacks only (if any)*

save?: { ability: 'STR'|'DEX'|'CON'|'INT'|'WIS'|'CHA'; onSuccess: 'half'|'none' };

damageDice?: string; *// e.g. "3d6" (slot/cantrip scaling not handled yet)*

damageType?: string; *// info only*

dcAbility?: 'INT'|'WIS'|'CHA'; *// preferred casting ability if API hints (optional)*

info?: { range?: string; concentration?: boolean; ritual?: boolean; casting\_time?: string };

}

export function normalizeSpell(api: any): NormalizedSpell {

const level = api.level ?? 0;

const attackType = api.attack\_type === 'ranged' || api.attack\_type === 'melee' ? api.attack\_type : undefined;

*// Save info*

let save: NormalizedSpell['save'] | undefined;

if (api.dc?.dc\_type?.name) {

const abName = api.dc.dc\_type.name.toUpperCase(); *// e.g. "DEX"*

const ability = (['STR','DEX','CON','INT','WIS','CHA'] as const).find(a => a === abName) as any;

const onSuccess = api.dc.dc\_success === 'half' ? 'half' : 'none';

if (ability) save = { ability, onSuccess };

}

*// Damage (take first block)*

let damageDice: string | undefined;

let damageType: string | undefined;

if (Array.isArray(api.damage?.damage\_at\_slot\_level)) {

*// rarely provided; prefer damage\_at\_character\_level or simple damage if present*

}

if (api.damage?.damage\_at\_character\_level) {

*// grab level 5 if present as a simple baseline for cantrips (we'll refine later)*

const dal = api.damage.damage\_at\_character\_level;

const firstKey = Object.keys(dal).sort((a,b)=>Number(a)-Number(b))[0];

damageDice = dal[firstKey];

} else if (Array.isArray(api.damage?.damage)) {

const d0 = api.damage.damage[0];

damageDice = d0?.damage\_dice;

damageType = d0?.damage\_type?.name;

} else if (api.damage?.damage\_dice) {

damageDice = api.damage.damage\_dice;

damageType = api.damage.damage\_type?.name;

}

let dcAbility: NormalizedSpell['dcAbility'] | undefined;

*// crude hint from classes (e.g., cleric→WIS, wizard→INT, bard/sorc/warlock→CHA)*

const classes: string[] = api.classes?.map((c: any) => c.name?.toLowerCase()) ?? [];

if (classes.includes('wizard')) dcAbility = 'INT';

else if (classes.includes('cleric') || classes.includes('druid') || classes.includes('ranger')) dcAbility = 'WIS';

else if (classes.some(c => ['bard','sorcerer','warlock','paladin'].includes(c!))) dcAbility = 'CHA';

return {

name: api.name,

level,

attackType,

save,

damageDice,

damageType,

dcAbility,

info: {

range: api.range,

concentration: !!api.concentration,

ritual: !!api.ritual,

casting\_time: api.casting\_time,

}

};

}

export async function getSpell(name: string): Promise<NormalizedSpell> {

const cached = readCachedSpell(name);

if (cached) return normalizeSpell(cached);

const api = await fetchSpellFromAPI(name);

writeCachedSpell(name, api);

return normalizeSpell(api);

}

**2) Core: spell casting**

Create packages/core/src/spells.ts:

import type { NormalizedSpell } from '@grimengine/dnd5e-api/src/spells'; *// adjust path if you namespace* import { abilityMod, proficiencyBonusForLevel, type Character, abilityMods as modsOf } from './character'; import { abilityCheck } from './checks'; import { attackRoll, damageRoll } from './combat'; export interface CastOptions { caster: Character; spell: NormalizedSpell; *// allow override of casting ability; else infer from spell.dcAbility or CHA/WIS/INT (pick highest)* castingAbility?: 'INT'|'WIS'|'CHA'; targetAC?: number; *// for spell-attacks only* seed?: string; } */\*\* Simple result object; extend later \*/* export interface CastResult { kind: 'save' | 'attack'; attack?: { rolls: number[]; natural: number; total: number; hit: boolean; expression: string }; save?: { ability: string; dc: number; success: boolean }; damage?: { base: number; final: number; expression: string }; notes?: string[]; } */\*\* Compute DC = 8 + PB + ability mod \*/* export function spellSaveDC(caster: Character, ability: 'INT'|'WIS'|'CHA'): number { const pb = proficiencyBonusForLevel(caster.level); return 8 + pb + abilityMod(caster.abilities[ability]); } */\*\* Choose casting ability if not provided: prefer spell hint, else highest of INT/WIS/CHA. \*/* export function chooseCastingAbility(caster: Character, spell: NormalizedSpell, override?: 'INT'|'WIS'|'CHA') { if (override) return override; if (spell.dcAbility) return spell.dcAbility; const picks: Array<['INT'|'WIS'|'CHA', number]> = ['INT','WIS','CHA'].map(a => [a as any, abilityMod(caster.abilities[a as any])]); picks.sort((a,b)=>b[1]-a[1]); return picks[0][0]; } export function castSpell(opts: CastOptions): CastResult { const { caster, spell } = opts; const cab = chooseCastingAbility(caster, spell, opts.castingAbility); const pb = proficiencyBonusForLevel(caster.level); const cabMod = abilityMod(caster.abilities[cab]); const notes: string[] = []; *// Damage expression (basic): if none, no damage* const dmgExpr = spell.damageDice ? `${spell.damageDice}+${Math.max(0, cabMod)}`.replace(/\+0$/,'') : undefined; if (spell.attackType) { *// Spell attack vs AC* const toHitMod = cabMod + pb; const atk = attackRoll({ abilityMod: toHitMod, advantage: false, disadvantage: false, targetAC: opts.targetAC, seed: opts.seed }); let damage; if (atk.hit && dmgExpr) { const d = damageRoll({ expression: dmgExpr, seed: opts.seed }); damage = { base: d.baseTotal, final: d.finalTotal, expression: d.expression }; } return { kind: 'attack', attack: { rolls: atk.d20s, natural: atk.natural, total: atk.total, hit: !!atk.hit, expression: atk.expression }, damage, notes }; } else if (spell.save) { const dc = spellSaveDC(caster, cab); const success = false; *// save result resolved externally in CLI (we don't know defender mods here)* let damage; if (dmgExpr) { *// We'll compute damage and let CLI cut it in half on a successful save.* const d = damageRoll({ expression: dmgExpr, seed: opts.seed }); damage = { base: d.baseTotal, final: d.finalTotal, expression: d.expression }; } return { kind: 'save', save: { ability: spell.save.ability, dc, success }, damage, notes }; } else { return { kind: 'save', notes: ['No attack or save mechanics found for this spell.'] }; } }

**3) CLI**

Update apps/cli/index.ts:

* Add spell group:
  + spell fetch "<name>" → fetch & cache; print basic summary (level, save or attack type, damage).
  + spell list → list cached spells.
  + spell show "<name>" → show normalized details.
* Add character cast "<spell>" [--ability INT|WIS|CHA] [--target "<id|name>"] [--seed S] [--ranged|--melee]
  + Load spell (fetch if not cached).
  + If **spell-attack**: require --target in an active encounter; use targetAC from that actor and cast with castSpell. For now, ignore conditions; (later we’ll integrate).
  + If **save-based**: require --target in an active encounter; compute **target save mod** from its abilityMods (fall back 0), compare vs caster DC; if success and spell.save.onSuccess === 'half', halve damage (round down).
  + Print a concise summary:
  + Casting Sacred Flame (WIS DC 13) on Skeleton *#1...*
  + Target DEX save: rolled 7 + 2 = 9 → FAIL
  + Damage: 1d8 → 6 radiant
  + If no encounter target provided for save-based, just print DC and damage preview without applying to a creature.

**4) Tests (offline; no network)**

* packages/core/tests/spells.test.ts
  + spellSaveDC for known ability scores/PB.
  + chooseCastingAbility picks hinted or highest mod.
  + A save-based spell (mock NormalizedSpell with save+damage) returns DC and damage preview.
  + An attack-based spell returns an attack result and, on hit, a damage result.
* packages/core/tests/fixtures/
  + Add one small spell.api.json fixture (e.g., Sacred Flame-like with dc and damage\_damage\_dice) and test normalizeSpell (re-export normalizeSpell for tests same as monsters).

**Acceptance Criteria**

* pnpm test passes (new spells tests included).
* spell fetch, spell list, spell show all work; cache is written under .data/cache/spells/.
* character cast "<spell>" --target "<name or id>":
  + If attack spell: rolls to hit vs AC, then damage on hit.
  + If save spell: prints DC, rolls target save, applies half/no damage per rule, and subtracts HP.
* Existing features remain functional.

---

*## 🧪 Test Command List for PR #15*

```powershell

*# 1) Run tests*

pnpm test

*# 2) Fetch a couple of spells and inspect*

pnpm dev -- spell fetch "Fire Bolt" *# spell attack cantrip, ranged*

pnpm dev -- spell fetch "Sacred Flame" *# save cantrip (DEX save, no half on success)*

pnpm dev -- spell list

pnpm dev -- spell show "Sacred Flame"

*# 3) Prepare an encounter*

pnpm dev -- encounter start --seed grim

pnpm dev -- character load-name "Aerin"

pnpm dev -- encounter add pc "Aerin"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

pnpm dev -- encounter list

*# 4) Cast a SAVE-based spell (Sacred Flame) on the Skeleton*

pnpm dev -- character cast "Sacred Flame" --target "Skeleton #1" --seed grim

*# Expect: prints WIS DC X, target DEX save roll + mod, fail/success, and damage (none on success)*

*# 5) Cast an ATTACK-based spell (Fire Bolt) on the Skeleton*

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

*# Expect: spell attack roll (uses caster PB + casting ability), vs AC; damage on hit*

*# 6) Show updated HP after casts*

pnpm dev -- encounter list

*# 7) Clean up*

pnpm dev -- encounter end

pnpm dev -- character unload

If you want, after spells we can either (a) add **slot/cantrip scaling** + **concentration tracking**, or (b) circle back to **encounter-aware checks** so conditions like *poisoned* auto-disadvantage ability checks through the CLI.

**You said:**

Uploaded a file

Uploaded a file

Uploaded a file

Uploaded a file

Uploaded a file

this all looks good to me. if you agree, pick whichever of those optiosn you think is best for the next PR

**ChatGPT said:**

Thought for 19s

Perfect—let’s build on spells next with **slot/cantrip scaling + concentration**. Here’s a clean PR prompt you can paste into Codex for **PR #16**, plus a copy-paste checklist of commands to verify it locally.

**🔖 Prompt for Codex → PR #16**

**Title:** feat(spells): slot/cantrip scaling + concentration tracking (start/end/check) + CLI hooks

**PR Body:**

Add two big spell features:

1) \*\*Damage scaling\*\*

- Support cantrip scaling by \*\*character level\*\* (e.g., Fire Bolt 1d10 → 2d10 at 5th).

- Support slot scaling for leveled spells with \*\*damage\_at\_slot\_level\*\*.

2) \*\*Concentration\*\*

- Track which actor is concentrating on which spell/target.

- Provide CLI to start/end concentration and to make concentration checks (CON save DC 10 or half damage).

Keep the implementation minimal and composable with existing modules.

---

*## Files & Changes*

*### A) Adapter: expose raw scaling fields (no network in tests)*

Update `packages/adapters/dnd5e-api/src/spells.ts`:

- In `NormalizedSpell`, add optional fields:

```ts

damageAtCharacterLevel?: Record<number, string>; // e.g., {1:"1d8",5:"2d8",11:"3d8",17:"4d8"}

damageAtSlotLevel?: Record<number, string>; // e.g., {1:"3d6",2:"4d6",...}

concentration?: boolean;

* In normalizeSpell(api), populate those from api.damage.damage\_at\_character\_level or  
  api.damage.damage\_at\_slot\_level (if present), and set concentration from api.concentration.

Note: Keep damageDice as a baseline (first available dice) for preview; scaling will pick the right dice in core.

**B) Core: scaling + concentration helpers**

Create/Update packages/core/src/spells.ts:

* New input for casting:

export interface CastOptions { caster: Character; spell: NormalizedSpell; castingAbility?: 'INT'|'WIS'|'CHA'; targetAC?: number; *// for attack spells* slotLevel?: number; *// for leveled spells (1..9)* seed?: string; }

* Add helpers:

*// Pick dice by caster level for cantrips (highest key <= level); fallback to base damageDice.* export function diceForCharacterLevel(spell: NormalizedSpell, characterLevel: number): string | undefined; *// Pick dice by slot for leveled spells (exact key match, else fallback to base).* export function diceForSlotLevel(spell: NormalizedSpell, slotLevel?: number): string | undefined;

* Update castSpell damage expression selection:
  + If spell.level === 0: use diceForCharacterLevel.
  + Else if slotLevel provided and damageAtSlotLevel exists: use diceForSlotLevel.
  + Else: use damageDice baseline.
  + Still append casting ability mod (e.g., +CHA mod), removing +0.
* **Concentration model** in encounter state:
  + In packages/core/src/encounter.ts, extend:

export interface ConcentrationEntry { casterId: string; spellName: string; targetId?: string; } export interface EncounterState { *// ...* concentration?: Record<string, ConcentrationEntry>; *// keyed by casterId* }

* + Add pure helpers:

export function startConcentration(state: EncounterState, entry: ConcentrationEntry): EncounterState; export function endConcentration(state: EncounterState, casterId: string): EncounterState; export function getConcentration(state: EncounterState, casterId: string): ConcentrationEntry | undefined;

* + Concentration check utility (DC 10 or half damage (round down), CON save):

export function concentrationDCFromDamage(dmg: number): number { return Math.max(10, Math.floor(dmg / 2)); }

Use existing saving throw logic to roll CON (for a PC: add PB if proficient in CON saves; for monsters: use abilityMods.CON, no PB unless you choose to extend).

**C) CLI wiring**

Update apps/cli/index.ts:

* character cast "<spell>" ...
  + After resolving the cast, if spell.concentration === true, and an active encounter exists:
    - Start concentration for the acting character (PC only for this PR) with:  
      startConcentration({ casterId: <pcId>, spellName: spell.name, targetId })
    - Print: Concentration started on <Spell> (caster: <name>, target: <id or none>).
* encounter concentration start "<casterIdOrName>" "<Spell Name>" [--target "<id|name>"]
  + Manually begin concentration (for monsters or if casting was done outside the helper).
* encounter concentration end "<casterIdOrName>"
  + Ends concentration for that actor.
* encounter concentration check "<casterIdOrName>" <damage>
  + Compute DC = concentrationDCFromDamage(damage).
  + Roll a CON save for that actor:
    - If PC in session: use characterSavingThrow with ability CON and DC.
    - Else for monsters: roll 1d20 + (abilityMods.CON || 0) against DC (no PB).
  + On failure: endConcentration and print Concentration broken. Otherwise print Maintained.
* **Encounter list**: if an actor is concentrating, show e.g. conc: <Spell> after the actor line.

**D) Tests (no network)**

Add/extend tests in packages/core/tests/spells.test.ts:

* diceForCharacterLevel picks 1/5/11/17 thresholds correctly.
* diceForSlotLevel picks exact slot match; falls back when absent.
* spellSaveDC and chooseCastingAbility keep working.
* A castSpell cantrip at L1 vs L5 uses different dice strings.

Add packages/core/tests/concentration.test.ts:

* concentrationDCFromDamage for values (e.g., 1→10, 15→10, 22→11).
* startConcentration / endConcentration / getConcentration pure-behavior.
* A simulated concentration check that fails ends concentration.

**E) Acceptance Criteria**

* pnpm test passes including the new tests.
* character cast "Concentration Spell" starts concentration automatically when an encounter is active.
* encounter concentration start/end/list/check interoperate as described.
* Cantrip/slot scaling visibly changes damage dice at L1 vs L5 (and slot level for leveled spells).
* Existing features untouched (attacks, saves, loot/xp, prior spell behavior).

---

*## 🧪 Test Command List for PR #16*

```powershell

*# 1) Run tests*

pnpm test

*# 2) Verify cantrip scaling (level 3 vs level 5)*

*# (A) Level 3 character*

pnpm dev -- character load-name "Aerin"

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add pc "Aerin"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

*# Cast Fire Bolt (ranged spell attack cantrip)*

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

*# Expect: damage dice show 1d10 at level 3 (your PB/mods per your sheet)*

*# (B) Bump to level 5 and re-cast (should use 2d10)*

pnpm dev -- character set --level 5

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

*# Expect: damage dice now 2d10 (visible in output)*

*# 3) Concentration flow*

*# Fetch a known concentration spell (e.g., "Bless" or "Guiding Bolt"—if you have one cached; if not, pick any concentration spell)*

pnpm dev -- spell fetch "Bless"

pnpm dev -- spell show "Bless" *# Should indicate concentration: true (if not, try a different concentration spell like "Hold Person")*

*# Start concentration manually (simulate already cast)*

pnpm dev -- encounter concentration start "Aerin" "Bless" --target "Skeleton #1"

pnpm dev -- encounter list *# Should show conc: Bless on Aerin*

*# Force a concentration check with damage = 7 (DC = 10)*

pnpm dev -- encounter concentration check "Aerin" 7

*# Expect: rolls CON save vs DC 10; on failure -> "Concentration broken", on success -> "Maintained"*

*# End concentration explicitly*

pnpm dev -- encounter concentration end "Aerin"

pnpm dev -- encounter list *# conc tag should be gone*

*# 4) Slot scaling (leveled spell)*

pnpm dev -- spell fetch "Guiding Bolt" *# has damage\_at\_slot\_level*

pnpm dev -- spell show "Guiding Bolt"

*# Cast at slot 1 (baseline)*

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 1

*# Cast at slot 3 (should have higher dice than slot 1)*

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 3

*# 5) Cleanup*

pnpm dev -- encounter end

pnpm dev -- character unload

If you want to queue what comes after #16, I’d go:

* **#17: Encounter-aware ability checks** (so conditions like *poisoned* auto-disadvantage checks via encounter check <actor> <ability> --dc N), or
* **#17: Spell slots + per-long-rest counters** (basic resource tracking in the character session).

Your call!

**You said:**

Uploaded a file

Uploaded a file

Uploaded a file

Looks good to me. I think encounter aware stuff is a good next step. You'll see in the log that I cast a number of things on the skeleton after it was dead :)

**ChatGPT said:**

Thought for 9s

Love the momentum — and yep, I saw you nuked that poor Skeleton a few extra times 😅. Let’s harden targeting and add **encounter-aware ability checks** next.

Here’s a **ready-to-paste PR prompt (Feature 17)** plus a **copy-paste test command list**.

**🔖 Prompt for Codex → PR #17**

**Title:** feat(core+cli): encounter-aware checks (DC + conditions), and targeting guards for defeated actors

**PR Body:**

Add encounter-aware ability/skill checks that apply conditions (e.g., poisoned ⇒ disadvantage),

and guard rails to prevent attacking/casting at defeated targets.

*## Scope*

1) Encounter checks:

- Run a DC check for any actor in an active encounter.

- Auto-apply disadvantage if the actor is \*poisoned\* (per 5e: ability checks have disadvantage).

- Allow optional skill use:

- If the actor is the session PC, apply PB/expertise from the loaded character.

- Otherwise (monsters/NPCs), just use the ability mod.

2) Targeting guards:

- `encounter attack` and `character cast` refuse actions against defeated targets (HP ≤ 0).

- Friendly message and non-zero exit.

3) Nice-to-haves:

- Block attacks \*from\* defeated actors.

- When listing, show `(DEFEATED)` as we do now; no change.

*## Files & Changes*

*### A) Core: encounter-aware checks*

Update `packages/core/src/encounter.ts`:

- Add a pure helper:

```ts

import { combineAdvantage } from './conditions';

import { abilityCheck } from './checks';

import type { AbilityName } from './abilityScores';

import { hasCondition, attackAdvFromConditions } from './conditions'; *// reuse combine logic style*

export interface EncounterCheckInput {

actorId: string;

ability: AbilityName;

dc?: number;

baseMod: number; *// caller computes (abilityMods[...] + PB if any)*

advantage?: boolean;

disadvantage?: boolean;

seed?: string;

}

*/\*\* For now: only poisoned affects ability checks (disadvantage). \*/*

export function encounterAbilityCheck(state: EncounterState, input: EncounterCheckInput) {

const a = state.actors[input.actorId];

if (!a) throw new Error('Unknown actor');

const condDis = hasCondition(a.conditions, 'poisoned') ? { disadvantage: true } : {};

const flags = combineAdvantage({ advantage: input.advantage, disadvantage: input.disadvantage }, condDis);

return abilityCheck({

ability: input.ability,

modifier: input.baseMod,

proficient: false,

proficiencyBonus: 0,

advantage: flags.advantage,

disadvantage: flags.disadvantage,

dc: input.dc,

seed: input.seed,

});

}

* Targeting guards in **actorAttack**:

*// before rolling:* if (state.defeated.has(attackerId)) throw new Error('Attacker is defeated.'); if (state.defeated.has(defenderId)) throw new Error('Target is defeated.');

**B) CLI**

Update apps/cli/index.ts:

* New subcommand under encounter:
* encounter check "<actorIdOrName>" <ABILITY> [--dc N] [--skill "<SkillName>"] [--adv|--dis] [--seed S]

Behavior:

* + Resolve actor; compute base modifier:
    - If --skill given AND actor is the loaded session PC:
      * Use characterSkillCheck logic to compute modifier parts (ability mod + PB/expertise + extraMod(0));
      * BUT invoke via encounterAbilityCheck so poisoned disadvantage applies from conditions.
      * Print which ability the skill uses.
    - Else: use actor’s abilityMods[ABILITY] || 0.
  + Print:
  + Check: <Actor> <ABILITY> [skill: <SkillName>] vs DC <N>
  + Rolls: [..] → total <T> → SUCCESS/FAIL
  + (conditions: disadvantage from poisoned) // only when it applied
* Targeting guards:
  + In encounter attack: if attacker or defender is defeated, print a clear error and exit 1.
  + In character cast:
    - If --target provided but target is defeated, refuse with a clear message and exit 1.

**C) Tests**

Add packages/core/tests/encounter-checks.test.ts:

* Build a tiny state with one PC-like actor and one monster.
* Give PC condition poisoned.
* encounterAbilityCheck with seed:
  + Without poisoned: rolls single d20; total = baseMod + d20; success vs DC as expected.
  + With poisoned: two d20s with disadvantage (min selected).
* actorAttack throws on defeated attacker/defender.

**D) Acceptance Criteria**

* pnpm test passes (new tests included).
* encounter check works:
  + With --skill Stealth on the session PC, uses DEX + PB/expertise (if present in char JSON).
  + Shows disadvantage line when actor is poisoned.
* encounter attack and character cast cannot target or act from defeated actors (friendly errors).
* Existing commands keep working.

---

*## 🧪 Test Command List for PR #17*

```powershell

*# 1) Run tests*

pnpm test

*# 2) Set up encounter*

pnpm dev -- character load-name "Aerin"

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add pc "Aerin"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

*# 3) Encounter-aware ability check (no conditions)*

pnpm dev -- encounter check "Aerin" STR --dc 12 --seed grim

*# 4) Apply poisoned (disadv on checks), then check again*

pnpm dev -- encounter condition add "Aerin" poisoned

pnpm dev -- encounter check "Aerin" STR --dc 12 --seed grim

*# Expect: output notes disadvantage from poisoned, shows two d20 and lower chosen*

*# 5) Skill check path (PC proficiency/expertise applied if present)*

pnpm dev -- encounter check "Aerin" DEX --skill "Stealth" --dc 14 --seed grim

*# 6) Targeting guards demo*

*# Drop Skeleton to 0 HP quickly*

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

*# (a) Attack guard*

pnpm dev -- encounter attack "Aerin" "Skeleton #1" --melee --seed grim

*# Expect: error "Target is defeated."*

*# (b) Cast guard*

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 1

*# Expect: error "Target is defeated."*

*# 7) Clean up*

pnpm dev -- encounter end

pnpm dev -- character unload

If you want, we can follow this with **PR #18: spell slots & per-rest counters** (track slots, simple long-rest refill) or **PR #18: short notes/logs on actors/encounters** (so Bless/Guiding Bolt advantage notes or “prone until end of next turn” can be recorded cleanly).

**You said:**

Uploaded a file

Looks good to me. if you agree, ok with whateve ryou want to do next

**ChatGPT said:**

Thought for 9s

Chef’s kiss — your run looks flawless. ✅

* 85/85 tests passing, encounter-aware checks + poisoned disadvantage working, and the defeated-target guards fired exactly as designed. Nice!

I’ll keep the pace and propose **Feature 18: Spell slots & rests** so casters actually spend resources.

**🔖 Prompt for Codex → PR #18**

**Title:** feat(core+cli): spell slots & rest counters (long/short), spend/restore, cast checks

**PR Body:**

Add basic spell-slot tracking and rest mechanics:

## Scope

- Track per-level spell slots (1–9) and remaining counts on the loaded character session.

- Enforce slot availability when casting leveled spells (level ≥ 1).

- Support short/long rest:

- Long rest: restore all slots to max; clear concentration.

- Short rest: no default slot restore (we’ll extend later for specific classes if needed).

- CLI to view/set slots, spend on cast, and rest.

## Files & Changes

### 1) Character model

Update `packages/core/src/character.ts`:

```ts

export interface SpellSlots {

max: Record<number, number>; *// e.g., {1:2, 2:0, 3:0, ...}*

remaining: Record<number, number>;

}

export interface Character {

*// existing...*

slots?: SpellSlots;

xp?: number; *// already present from Feature 13*

}

Add helpers:

export function ensureSlots(c: Character): SpellSlots { const zeros = ((): Record<number, number> => { const o: any = {}; for (let i=1;i<=9;i++) o[i]=0; return o; })(); const max = { ...zeros, ...(c.slots?.max ?? {}) }; const remaining = { ...zeros, ...(c.slots?.remaining ?? max) }; c.slots = { max, remaining }; return c.slots; } export function canSpendSlot(c: Character, level: number): boolean { const s = ensureSlots(c); return s.remaining[level] > 0; } export function spendSlot(c: Character, level: number): boolean { const s = ensureSlots(c); if (s.remaining[level] > 0) { s.remaining[level]--; return true; } return false; } export function restoreAllSlots(c: Character) { const s = ensureSlots(c); for (let i=1;i<=9;i++) s.remaining[i] = s.max[i]; } export function setSlots(c: Character, maxByLevel: Partial<Record<number, number>>) { const s = ensureSlots(c); for (const k of Object.keys(maxByLevel)) { const i = Number(k); s.max[i] = maxByLevel[i]!; *// If remaining previously exceeded new max, clamp.* if (s.remaining[i] > s.max[i]) s.remaining[i] = s.max[i]; } }

**2) Concentration clear on long rest**

In packages/core/src/encounter.ts, export a helper:

export function clearAllConcentration(state: EncounterState): EncounterState { if (!state.concentration) return state; return { ...state, concentration: {} }; }

**3) Casting enforcement**

In packages/core/src/spells.ts (or in CLI glue if you prefer minimal core changes):

* When spell.level >= 1 and a slotLevel is passed (or defaults to spell.level):
  + Verify canSpendSlot(caster, slotLevel); if not, return a result with notes: ["No slot available at level X"] and **do not** roll damage/attack.
  + If okay, proceed and **remember to call**spendSlot (mutating the loaded character session file).

(For cantrips level === 0, no slot usage.)

**4) CLI**

Update apps/cli/index.ts:

* character slots show
  + Prints a table: Level | Max | Remaining (only levels with any nonzero max).
* character slots set "<spec>"
  + <spec> format: comma-separated L=X pairs, e.g. 1=2,2=0,3=0
  + Updates slots.max, clamps remaining, saves session, and prints new table.
* character rest long
  + Restores all slots via restoreAllSlots.
  + If an encounter is active, also clear concentration via clearAllConcentration on the current encounter file.
  + Print: Long rest complete: slots restored; concentration cleared.
* character rest short
  + For now: print Short rest: no slot recovery by default. (placeholder for future class features).
* **Casting guard** (in existing character cast):
  + If the spell is leveled (level ≥ 1):
    - Determine slot level: use --slotLevel if provided; else default to spell’s level.
    - If no slot available → error message and exit 1.
    - On success → spend slot and show (slot L spent: remaining R/M) in the output.

**5) Tests**

Add packages/core/tests/slots.test.ts:

* ensureSlots populates zeros and mirrors max to remaining on first setup.
* setSlots updates max levels and clamps remaining.
* spendSlot returns true when available, false when empty.
* Simple integration: simulate a leveled spell cast that refuses without remaining slots.

**6) Acceptance Criteria**

* pnpm test passes including slots.test.ts.
* character slots set "1=2,2=1" then character slots show prints the expected table.
* Casting a leveled spell at --slotLevel 1 spends a slot and shows updated remaining; failure when empty.
* character rest long restores remaining to max and clears encounter concentration if present.
* Cantrips continue to cast without slots.

---

*## 🧪 Test Command List for PR #18*

```powershell

*# 1) Tests*

pnpm test

*# 2) Prep character + slots*

pnpm dev -- character load-name "Aerin"

pnpm dev -- character slots set "1=2,2=0,3=0"

pnpm dev -- character slots show

*# 3) Start an encounter (for concentration clearing demo later)*

pnpm dev -- encounter start --seed grim

pnpm dev -- encounter add pc "Aerin"

pnpm dev -- encounter add monster "Skeleton"

pnpm dev -- encounter roll-init

*# 4) Cast leveled spell with slot spending*

pnpm dev -- spell fetch "Guiding Bolt"

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 1

*# Expect: success + "slot 1 spent: remaining 1/2"*

*# 5) Cast again until slots empty, then one more to trigger guard*

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 1

pnpm dev -- character cast "Guiding Bolt" --target "Skeleton #1" --seed grim --ranged --ability WIS --slotLevel 1

*# Expect: error "No slot available at level 1"*

*# 6) Long rest (restore + clear concentration)*

pnpm dev -- character rest long

pnpm dev -- character slots show *# remaining back to max*

*# 7) Cantrip still free*

pnpm dev -- character cast "Fire Bolt" --target "Skeleton #1" --seed grim --ranged

*# 8) Cleanup*

pnpm dev -- encounter end

pnpm dev -- character unload

If that sounds good, after #18 I’d recommend **#19: actor/encounter notes & timed tags** (e.g., “Bless on Aerin until end of round 4” with decrement on nextTurn), which will make concentration/buffs feel way nicer in the loop.

**You said:**

Uploaded a file

looks good to me. ready to move on if you agree

**ChatGPT said:**

Thought for 19s

Heck yes — your run looks perfect. Let’s keep the rhythm with **Feature 19: Notes & Timed Tags** so you can mark buffs like “Bless (ends R4)” and have them auto-expire as the encounter advances.

Here’s the **PR prompt** for Codex and the **exact test commands**.

**🔖 Prompt for Codex → PR #19**

**Title:** feat(core+cli): notes & timed tags on actors (auto-expire on turns/rounds)

**PR Body:**

Add lightweight notes/timed tags to actors in an encounter. Use them for reminders like

“Bless (ends R4)” or “Marked by Hunter’s Mark,” and auto-expire them as time advances.

## Goals

- Attach any number of notes/tags to an actor.

- Optional duration in rounds (relative to the current round) with automatic removal.

- Show notes in `encounter list`.

- Manage via CLI (add/list/remove/clear).

- On `encounter next`, auto-expire any tag whose end round is now in the past.

## Core

### 1) Data model

Edit `packages/core/src/encounter.ts`:

```ts

export interface ActorTag {

id: string; *// unique per actor*

text: string; *// e.g., "Bless"*

addedAtRound: number; *// round when added*

expiresAtRound?: number; *// inclusive; when round > expiresAtRound, tag is removed*

note?: string; *// optional free text detail (e.g., "D4 to attacks/saves")*

source?: string; *// e.g., "Aerin", "Spell: Bless"*

}

export interface ActorBase {

*// existing...*

tags?: ActorTag[];

}

**2) Helpers (pure functions)**

Add to packages/core/src/encounter.ts:

*// Add a tag (auto-generate short id like "t1", "t2" per actor)* export function addActorTag(s: EncounterState, actorId: string, tag: Omit<ActorTag, 'id'|'addedAtRound'>): EncounterState; *// Remove a tag by id* export function removeActorTag(s: EncounterState, actorId: string, tagId: string): EncounterState; *// Clear all tags from one actor* export function clearActorTags(s: EncounterState, actorId: string): EncounterState; *// Expire tags whose expiresAtRound < current round (called during nextTurn)* export function expireActorTags(s: EncounterState): EncounterState;

Implementation notes:

* addActorTag should create a fresh copy of state and append { id: auto, addedAtRound: s.round }.
* expireActorTags returns a **new** state with all expired tags removed across actors.
* Call expireActorTags inside nextTurn *after* updating turnIndex/round so expiration happens at the start of the new actor’s turn.

**3) List formatting**

In the existing encounter list output, show tags after conditions, e.g.:

Aerin (id=aerin) AC 18 HP 25/25 — conditions: — — tags: Bless (R1→R4), Hunter’s Mark (R2→R6)

If a tag has no expiry, just print the text (and, if present, note as “— note: …”).

**CLI**

Update apps/cli/index.ts with a new encounter note group:

1. encounter note add "<actorIdOrName>" "<text>" [--rounds N] [--note "<detail>"] [--source "<who/what>"]
   * Adds a tag to the actor.
   * If --rounds N given, set expiresAtRound = state.round + (N - 1) so “1 round” ends this round, “2 rounds” ends next round, etc.
   * Print the assigned tag id and its expiry if any:
   * Added tag t3 to Aerin: "Bless" (expires at round 4)
2. encounter note list
   * Print all actors with their tags (id, text, Rstart→Rend, note).
3. encounter note remove "<actorIdOrName>" <tagId>
   * Remove a single tag by id.
4. encounter note clear "<actorIdOrName>"
   * Remove all tags for that actor.

Behavioral hook:

* encounter next already advances turns. After incrementing round/turn, call expireActorTags and print a small footer if something expired:
* Expired tags: Aerin[t3: Bless], Skeleton #2[t1: Hunter’s Mark]

**Tests**

Add packages/core/tests/tags.test.ts (Vitest):

* Add tag with and without expiresAtRound; verify addedAtRound matches current round.
* Advance rounds via nextTurn enough to trigger expiration; confirm removal.
* Multiple actors with interleaved tags; ensure only expired ones are removed.
* Remove-by-id and clear-all work and are pure (original state unchanged).

**Acceptance Criteria**

* pnpm test passes (new tags.test.ts included).
* encounter note add creates tags with proper expiry math and unique ids per actor.
* encounter next triggers expiration; console shows which tags expired.
* encounter list displays tags in a compact readable way alongside conditions.
* encounter note remove and clear work; errors are friendly for unknown ids/actors.
* No regressions to existing commands/features.

*---*

## 🧪 Test Command List for PR #19

```powershell

# 1) Tests

pnpm test

# 2) Set up a quick encounter

pnpm dev *-- character load-name "Aerin"*

pnpm dev *-- encounter start --seed grim*

pnpm dev *-- encounter add pc "Aerin"*

pnpm dev *-- encounter add monster "Skeleton"*

pnpm dev *-- encounter roll-init*

pnpm dev *-- encounter list*

# 3) Add notes/tags (one timed, one permanent)

pnpm dev *-- encounter note add "Aerin" "Bless" --rounds 4 --note "d4 to attacks/saves" --source "Spell"*

pnpm dev *-- encounter note add "Skeleton #1" "Marked" --note "Hunter's Mark"*

pnpm dev *-- encounter note list*

pnpm dev *-- encounter list # should show tags on both actors*

# 4) Advance turns until expiration triggers

pnpm dev *-- encounter next*

pnpm dev *-- encounter next*

pnpm dev *-- encounter next*

pnpm dev *-- encounter next # by now, Bless should be at or near expiry depending on initiative order*

pnpm dev *-- encounter list # Bless removed once round > expiresAtRound, log printed on expiry*

# 5) Remove and clear via CLI

pnpm dev *-- encounter note add "Skeleton #1" "Taunted" --rounds 2*

pnpm dev *-- encounter note list*

# Suppose the new id was t2; remove it:

pnpm dev *-- encounter note remove "Skeleton #1" t2*

pnpm dev *-- encounter note clear "Skeleton #1"*

pnpm dev *-- encounter note list*

# 6) Cleanup

pnpm dev *-- encounter end*

pnpm dev *-- character unload*

If that lands cleanly, two nice follow-ups:

* **#20: Bless/Hunter’s Mark helpers** (sugar commands that both add the tag *and* start concentration when appropriate), or
* **#20: Advantage cues** — e.g., optional note metadata so a tag can print a reminder line during attack (“Reminder: Bless active (add d4 manually)”).

**You said:**

Uploaded a file

This looks good to me. Ready to move on if you ware. Whichever feature you think is best next is fine. I just want to move the disucssion to a new thread since this one is starting to get big, so can you give me a message to carry into a new thread that will get that next PR / list of commands going?

**ChatGPT said:**

Perfect — all 93 tests passed again, and your encounter log confirms timed tag expiration and manual removal both work exactly as designed ✅

We’re now ready for the next feature set (Feature 20). I’d like to take us into **Bless/Hunter’s Mark helpers** — “smart concentration buffs” that both:

* apply advantage/bonus cues automatically, and
* create the proper tag + concentration link in one step.

Here’s a short message you can paste into a **new thread** to pick up exactly where we left off:

**Message to start next thread:**

Continuing GrimEngine development — ready for **Feature 20**.  
The last confirmed feature was Notes & Timed Tags (Feature 19).  
Please generate the **PR prompt + test command list** for the next milestone:  
“**Bless / Hunter’s Mark helpers** – smart concentration buffs that add tags and concentration in one command.”

Include all relevant CLI syntax, test coverage, and acceptance criteria as usual.

Once you post that in a new thread, I’ll deliver the full PR #20 prompt and test-run checklist to keep the build chain going cleanly.

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info.