Planificación de desarrollo - Motor RPG server-side data-driven

Roadmap por fases para concretar requisitos, diseñar componentes, integrar y validar el motor de cálculo de stats

Fecha: 05/02/2026

1. Objetivo y alcance inicial

Construir un motor que se ejecute enteramente en servidor, multi-instancia y data-driven (config JSON), que procese transacciones atómicas sobre un estado en memoria (persistido por snapshots) y que, en esta fase, ofrezca: creación de jugadores/personajes/gear, progresión de niveles, equipamiento con restricciones, bonos por set y cálculo de stats finales. El cliente no guarda estado: solo envía transacciones y consulta resultados.

Fuera de alcance (por ahora): combate, loot aleatorio, economía completa, habilidades, buffs temporales.

2. Principios guía del desarrollo

- Diseño guiado por contratos: esquemas JSON y catálogo de errores definidos antes de implementar lógica.
- Core determinista por defecto; la aleatoriedad (si existe) debe quedar registrada en el estado como resultado.
- Invariantes fuertes: ownership obligatorio, gear único (una ubicación), equipamiento consistente con slots.
- Pruebas desde el día 1: golden tests para cálculo de stats + tests de atomicidad y migración.
- Incremento por 'vertical slices': cada hito atraviesa API -> validación -> transacción -> estado -> persistencia -> tests.

3. Fase 0 - Preparación (base técnica y criterios de éxito)

Entregables

- Definición de hitos y Definition of Done por hito (qué significa 'terminado').
- Estructura inicial del repositorio, CI para tests, linter/formatter.
- Primeros 'golden files' de ejemplo (config mínima + casos esperados de stats).

Decisiones a cerrar

- Nivel de durabilidad: ¿se acepta pérdida entre snapshots? (si no, se necesita un log de transacciones).
- Modelo de extensibilidad 'sin tocar código': DSL declarativa vs catálogo fijo parametrizable vs plugins.
- Política de concurrencia: serialización por instancia o por jugador (suficiente para empezar).

4. Fase 1 - Concreción final de requisitos (taller de especificación)

Objetivo: eliminar ambigüedades antes del diseño definitivo de interfaces.

4.1 Semántica del dominio (decisiones)

- Equipamiento: conflicto de slots (rechazar vs swap explícito), selección de slot cuando hay alternativas.
- Gear multi-slot: ocupa conjuntos exactos; política de conteo para sets (1 pieza vs 2).
- Restricciones: definición exacta de la regla de nivel ('x niveles por encima') y listas blancas/negras de clase.
- Cálculo: permitir negativos o clamp por stat; redondeo solo en presentación (si aplica).
- Migración config <-> estado: qué hacer si desaparecen slots/gearDefs/clases en una nueva config.

4.2 Esquemas y contratos

- JSON Schema de GameConfig, GameState, Transaction y TransactionResult.
- Catálogo de errorCode y normas de validación (rechazo sin efectos parciales).
- Ejemplos mínimos: config pequeña, estado pequeño, transacciones de ejemplo y respuestas.

Entregable principal

Anexo de Semántica + Esquemas (documento breve que deja el sistema 'cerrado' en significado).

5. Fase 2 - Diseño de arquitectura y contratos (sin entrar en implementación profunda)

Componentes lógicos (interfaces internas)

- ConfigLoader: carga y valida config, y genera un modelo interno eficiente.
- StateStore: mantiene estado en memoria por instancia; versionado de estado; snapshots.
- Migrator: carga snapshots antiguos y aplica best-effort con config nueva.
- TransactionProcessor: valida y aplica transacciones atómicas; genera TransactionResult.
- Rules/Algorithms Engine: restricciones, crecimiento de stats, coste de nivel, sets (pluggable/DSL).
- StatsCalculator: cálculo final puro y testeable (sin side effects).
- QueryService: endpoints de lectura (no mutan estado).

Contrato API (alto nivel)

- Comando: endpoint único de transacciones (POST tx) + consultas (GET config, GET stats, GET estado jugador).
- Idempotencia: decidir si txId evita duplicados ante reintentos.
- Respuestas estandarizadas: accepted, errorCode, errorMessage, stateVersion (y opcional stateHash).

6. Fase 3 - Implementación incremental por 'vertical slices'

La prioridad es tener un esqueleto funcionando end-to-end y aumentar complejidad gradualmente.

Slice 1 - Instancia + config + health + snapshots mínimos

- Arranque del servidor, carga de config válida, creación de instancia vacía, snapshots periódicos y recarga tras reinicio.
- Endpoints mínimos: health/config/stateVersion.

Slice 2 - Jugadores y ownership

- CreatePlayer y consultas de jugador.
- Invariantes: nada existe sin owner; pertenencia a instancia; no duplicados.

Slice 3 - Personajes (clase + nivel) + stats base

- CreateCharacter(classId), LevelUpCharacter, GetCharacterStats (solo stats del personaje).
- Golden tests de crecimiento.

Slice 4 - Gear instanciable + inventario

- CreateGear(gearDefId), LevelUpGear, inventario de jugador.
- Golden tests de crecimiento de gear.

Slice 5 - Equipamiento 1 slot + suma de stats

- Equip/Unequip (caso simple), stats finales = personaje + suma gear.
- Validación de ocupación y unicidad del gear.

Slice 6 - Gear multi-slot + conflictos + (opcional) swap

- Ocupación de varios slots (conjuntos exactos). Política de conflicto aplicada (reject o swap explícito).
- Consistencia al desequipar.

Slice 7 - Restricciones de equipamiento

- Restricciones por clase y por nivel; errores estandarizados.
- Matriz de tests por restricción.

Slice 8 - Sets (2/4 piezas)

- Activación por conteo, bonusStats sumados.
- Casos límite y política para gear multi-slot.

Slice 9 - Migración config <-> estado

- Cambios en stats/slots sin romper carga; políticas para slots/defs desaparecidos.
- Golden tests de migración con snapshots antiguos.

Slice 10 - Extensibilidad (DSL o mecanismo elegido)

- Incorporar el modelo elegido para añadir algoritmos y reglas sin tocar el core.
- Tests de seguridad/limitación (si hay DSL) y fixtures de expresiones.

7. Estrategia de pruebas y validación

- Tests de esquema: GameConfig/Transaction válidos e inválidos (falla rápido).
- Golden tests de StatsCalculator: configs pequeñas con resultados esperados y reproducibles.
- Tests de invariantes: gear no puede estar en dos ubicaciones; slots ocupados coherentes; ownership.
- Tests de atomicidad: cualquier fallo deja el estado idéntico (sin efectos parciales).
- Replay tests: aplicar secuencia de transacciones y comparar con snapshot esperado.
- Migration tests: cargar snapshot antiguo con config nueva y validar best-effort.

8. Observabilidad y operación

- Logging estructurado de transacciones (aceptada/rechazada, motivo, duración).
- Métricas básicas: latencia por tx, tamaño del estado, frecuencia y duración de snapshots.
- Herramientas operativas: exportar snapshot, forzar snapshot, listar instancias (si aplica).

9. Riesgos y mitigaciones

- Extensibilidad 'sin tocar código': si no se decide pronto (DSL vs catálogo vs plugins), se rediseña el núcleo. Mitigación: decisión en Fase 1.
- Snapshots sin log: si luego se exige durabilidad fuerte, añadir log puede ser intrusivo. Mitigación: declarar pérdida aceptable o introducir log mínimo desde el inicio.
- Multi-instancia: complica aislamiento y memoria. Mitigación: empezar con 1 instancia funcional y habilitar multi-instancia después (Slice 2-3).

10. Checklist de decisiones antes de codificar (cierre de especificación)

- Política de conflictos al equipar (reject vs swap explícito).
- Semántica exacta de restricciones de nivel.
- Conteo de sets con gear multi-slot.
- Negativos/clamp/redondeo por stat (defaults).
- Durabilidad: pérdida aceptable entre snapshots o log mínimo.
- Modelo de extensibilidad: DSL vs catálogo vs plugins.
- Idempotencia (txId) y serialización de transacciones (instancia vs jugador).