Motor RPG data-driven (cálculo de stats) Especificación funcional

Documento de definición funcional para un motor de servidor que gestiona estado de juego y calcula stats resultantes de personajes y equipo. Este documento evita detalles de implementación y se centra en contratos, modelos de datos y reglas.

1. Resumen

El sistema ejecuta instancias de juegos RPG de fantasía definidas por una configuración (JSON). Toda la ejecución ocurre en servidor: el cliente envía transacciones atómicas y el servidor devuelve resultados y/o el nuevo estado. El alcance actual del motor es el cálculo de stats finales (personaje + gear + bonos de set).

Principios clave:

- Data-driven: nuevas stats, slots, clases, gear, sets y algoritmos se describen en JSON.
- Servidor autoritativo: el cliente no persiste estado; solo envía transacciones.
- Estado en memoria con snapshots periódicos a JSON (sin base de datos).
- Multi-config: el mismo motor aloja múltiples instancias de juego, cada una con su configuración.

2. Alcance y no-alcance

En alcance (fase actual):

- Gestión de jugadores, personajes y gear (creación y pertenencia).
- Gestión de niveles (personaje y gear) y cálculo de coste de subida de nivel (sin economía obligatoria, ver sección 7.3).
- Equipar/desequipar gear en slots, incluyendo gear que ocupa 2 slots.
- Cálculo de stats finales incluyendo bonos por set (2 piezas, 4 piezas).
- Persistencia del estado mediante snapshots JSON y carga al arrancar.

Fuera de alcance (por ahora):

- Combate, habilidades, buffs/debuffs temporales, IA, mundo, misiones, loot/crafting.
- Economía completa (monedas, inventario de materiales) salvo que se decida incorporarla explícitamente.
- Cambios de configuración en caliente (se reinicia el servidor o la instancia).

3. Glosario y entidades

Término	Definición
Stat	Atributo numérico (float) identificado por statId (p. ej. strength, crit_chance).
Slot	Hueco de equipamiento en un personaje (p. ej. head, chest, right_hand).
Clase	Tipo de personaje (human, elf, etc.) con stats base iniciales.

GearDef	Definición de un tipo de gear (stats base, slots compatibles, setId, restricciones, etc.).
Gear (instancia)	Objeto concreto que pertenece a un jugador (gearId) con nivel propio y opcionalmente equipado.
Personaje (instancia)	Entidad concreta (characterId) que pertenece a un jugador con clase, nivel y equipamiento.
Set	Conjunto de gear que activa bonos al equipar N piezas (p. ej. 2 y 4).
Transacción	Comando atómico enviado por cliente que modifica el estado del juego.
Instancia de juego	Ejecución aislada asociada a una configuración; contiene su propio estado y jugadores.

4. Separación Configuración vs Estado

El sistema separa estrictamente configuración (definiciones) de estado (instancias y progreso). Una instancia de juego se ejecuta con una configuración fija durante su vida útil.

GameConfig (JSON, inmutable durante la ejecución):

- Catálogo de stats (statId).
- Catálogo de slots (slotId).
- Clases (classId) con stats base.
- Definiciones de gear (gearDefId): stats base, compatibilidad de slots, setId, restricciones.
- Definiciones de sets (setId) con bonos por umbral (2 piezas, 4 piezas).
- Algoritmos: progresión de stats por nivel y coste de subida de nivel.
- Nivel máximo global para personajes y gear.

GameState (JSON, mutable):

- Jugadores: playerId.
- Personajes: characterId, ownerPlayerId, classId, level, equipamiento.
- Gear: gearId, ownerPlayerId, gearDefId, level, estado de equipamiento (en inventario o equipado).
- Versionado de estado (stateVersion) para auditar cambios.

5. Modelo de datos conceptual

Las siguientes estructuras son conceptuales (no dictan estructura exacta de JSON), pero fijan el contrato semántico.

5.1 Catálogo de stats y slots

- Todos los valores de stats son float.
- La lista de stats y slots es común para todos los personajes y para todo gear.

5.2 Clases de personaje

- Cada clase define stats base de nivel 1 para todas las stats (las no indicadas se consideran 0).
- Un personaje siempre pertenece a una única clase.

5.3 Definición e instancia de gear

- Un gearDef define stats base (nivel 1), compatibilidad de slots y (opcional) setId.
- Un gear (instancia) referencia un gearDef y mantiene su propio nivel.
- Un gear pertenece siempre a un jugador (ownerPlayerId).

5.4 Equipamiento

- Un personaje tiene un mapa slotId -> gearId (o vacío).
- Un gear puede requerir ocupar 1 slot o exactamente 2 slots (p. ej. right_hand + off_hand).
- Un slot no puede contener más de un gear.

6. Reglas de equipamiento

6.1 Compatibilidad de slots

- Un gearDef declara uno o más patrones de ocupación (listas exactas de slotId).
- Ejemplos: [right_hand] o [right_hand, off_hand].
- Si declara varias alternativas, el cliente debe indicar cuál usa (o el servidor elige por una regla definida en config).

6.2 Conflictos y política de reemplazo

- Modo estricto (por defecto): si algún slot requerido está ocupado, la transacción falla sin cambios.
- Modo swap (opcional): si se solicita explícitamente, el servidor desequipa lo existente y equipa el nuevo gear como una única transacción atómica.

6.3 Restricciones de equipamiento

- Restricciones por clase: allowlist o blocklist de classId.
- Restricciones por nivel: se define una o varias reglas, por ejemplo:
- - requiredCharacterLevel <= characterLevel
- - gearLevel <= characterLevel + X
- Si una restricción falla, la transacción se rechaza con código y motivo.

6.4 Pertenencia y unicidad

- Un gearId solo puede estar en inventario o equipado en un personaje del mismo jugador.
- No existen personajes ni gear sin ownerPlayerId.

7. Niveles, progresión y coste

7.1 Nivel máximo

- Nivel mínimo: 1.
- Nivel máximo global configurable para personajes y gear (mismo máximo para ambos).

7.2 Progresión de stats por nivel

La progresión define cómo se obtienen las stats efectivas a un nivel dado a partir de stats base (nivel 1).

- Función conceptual: scaledStats = GrowthAlgorithm(baseStats, level, context).
- El contexto puede incluir classId (para personajes) o gearDefId (para gear).
- El algoritmo concreto se selecciona por algorithmId en la configuración.

7.3 Coste de subir de nivel

- Existe un algoritmo para calcular el coste de subir nivel para personajes (experiencia) y para gear (materiales).
- En esta fase, el motor puede: (a) solo calcular y exponer el coste, o (b) además validar y descontar recursos si el estado los modela.
- La elección se fija en la configuración del juego.

8. Cálculo de stats finales

El motor calcula las stats finales de un personaje sumando las contribuciones escaladas por nivel del personaje, del gear equipado y de los bonos de set activos. Todas las combinaciones son por suma.

8.1 Definición formal

```
charStats = Grow(classBaseStats[classId], characterLevel)
gearStats = Σ Grow(gearBaseStats[gearDefId], gearLevel)   (para cada gear equipado)
setBonusStats = Σ ActiveSetBonuses(equipment)
finalStats = charStats + gearStats + setBonusStats
```

8.2 Consideraciones numéricas

- Las stats son float y se conservan con precisión completa internamente.
- Opcionalmente, cada stat puede definir un mínimo (p. ej. 0) y/o máximo para aplicar clamp tras el cálculo.

9. Sets y bonos

- Cada gearDef puede referenciar un setId (o ninguno).
- Un set define umbrales (p. ej. 2 piezas y 4 piezas) y para cada umbral un vector de bonusStats.
- La activación se evalúa por personaje: N piezas equipadas del mismo setId en ese personaje.
- Para gear que ocupa 2 slots, la configuración define si cuenta como 1 o 2 piezas (por defecto 1).

10. Servidor, instancias y API

10.1 Instancias de juego

- El motor aloja múltiples instancias, cada una con gameInstanceId y una configuración asociada (gameConfigId/version).
- No se requiere cambio de configuración en caliente; se reinicia servidor/instancia al cambiar.

10.2 Contrato de transacciones

- El cliente envía transacciones atómicas al servidor; el servidor valida y aplica o rechaza sin cambios.
- El motor es autoritativo: el cliente no persiste estado.
- Las transacciones pueden incluir aleatoriedad; los resultados que afectan al estado deben quedar registrados.

10.3 Tipos mínimos de transacciones

- CreatePlayer
- CreateCharacter (classId)
- CreateGear (gearDefId)
- LevelUpCharacter
- LevelUpGear

- EquipGear (gearId, characterId, opcional: patrón/slot principal, modo swap)
- UnequipGear

10.4 Consultas (sin cambio de estado)

- Obtener configuración de la instancia.
- Obtener resumen de estado del jugador (personajes, inventario).
- Obtener stats finales de un personaje.

10.5 Respuestas y errores

- Respuesta estándar: accepted (bool), errorCode (si falla), errorMessage, stateVersion.
- Códigos típicos: NOT_FOUND, OWNERSHIP_VIOLATION, SLOT_OCCUPIED, SLOT_INCOMPATIBLE, RESTRICTION_FAILED, MAX_LEVEL_REACHED, INVALID_CONFIG_REFERENCE.

11. Persistencia y compatibilidad Config-Estado

11.1 Snapshots

- El estado se mantiene en memoria y se escribe a JSON cada snapshotIntervalSeconds.
- Al reiniciar, el motor carga el último snapshot disponible para cada instancia.
- Se acepta (si así se define) pérdida de cambios entre el último snapshot y una caída del proceso.

11.2 Compatibilidad y migración best-effort

- Al cargar un snapshot con una configuración que no encaja: se hace match por IDs.
- Stats: las que coinciden por statId se mantienen; nuevas se inicializan a 0; desaparecidas se ignoran.
- Slots: si desaparece un slot que estaba ocupado, la política debe estar definida: desequipar a inventario o marcar como inválido y excluir del cálculo.
- Definiciones inexistentes (classId/gearDefId/setId): la política debe definirse (rechazar carga o invalidar entidades).
- El servidor siempre expone la configuración activa de la instancia.

12. Extensibilidad data-driven

El objetivo es poder añadir contenido y lógica sin modificar el núcleo del motor.

- Añadir stats/slots/clases/gear/sets: mediante edición de GameConfig (JSON).
- Añadir algoritmos de progresión y coste: mediante un sistema declarativo en JSON (p. ej. DSL/expresiones o tablas), referenciado por algorithmId.
- Añadir reglas de equipamiento: mediante reglas declarativas parametrizadas en config (allowlists, restricciones por nivel, etc.).

Si se desea incorporar reglas arbitrarias complejas no expresables en el formato declarativo elegido, debe ampliarse el lenguaje declarativo o admitir extensiones externas. Esta decisión se considera parte del producto.

Anexo A. Estructura JSON conceptual

Ejemplo orientativo de cómo podrían organizarse los documentos JSON. Es conceptual y no impone nombres definitivos.

```
{
  "gameConfigId": "fantasy_v1",
  "maxLevel": 60,
  "stats": ["strength", "agility", "intelligence", "armor", "hp"],
  "slots": ["head", "chest", "legs", "right_hand", "off_hand"],
  "classes": {
    "human": { "baseStats": { "strength": 5, "hp": 20 } },
    "elf":   { "baseStats": { "agility": 7, "hp": 16 } }
  },
  "algorithms": {
    "growth": { "algorithmId": "linear_table_v1", "params": { } },
    "levelCostCharacter": { "algorithmId": "xp_curve_v1", "params": { } },
    "levelCostGear": { "algorithmId": "mat_curve_v1", "params": { } }
  },
  "gearDefs": {
    "two_handed_sword": {
      "baseStats": { "strength": 3 },
      "equipPatterns": [["right_hand", "off_hand"]],
      "restrictions": { "allowedClasses": ["human", "elf"], "maxLevelDelta": 5 },
      "setId": "warrior_set"
    }
  },
  "sets": {
    "warrior_set": {
      "bonuses": [
        { "pieces": 2, "bonusStats": { "strength": 2 } },
        { "pieces": 4, "bonusStats": { "hp": 10 } }
      ]
    }
  }
}
```

Ejemplo orientativo de snapshot de estado:

```
{
  "gameInstanceId": "instance_001",
  "gameConfigId": "fantasy_v1",
  "stateVersion": 184,
  "players": {
    "p1": {
      "characters": {
        "c1": {
          "classId": "human",
          "level": 12,
          "equipped": {
            "right_hand": "g9",
            "off_hand": "g9"
          }
        }
      },
      "gear": {
        "g9": {
          "gearDefId": "two_handed_sword",
          "level": 10,
          "equippedBy": "c1"
        }
      }
    }
```

```
    }
  }
```