ChatGPT

Stake Engine SDK – Comprehensive Slot Game Development Guide

Introduction to the Stake Development Kit

The **Stake Development Kit (SDK)** is a comprehensive framework that simplifies the creation, simulation, and optimization of slot games on the Stake platform [1]. It consists of two main components:

1. **Math Framework:** A Python-based engine for defining game rules, simulating outcomes, and optimizing win distributions [2]. This math SDK generates all necessary backend files, configuration tables, and simulation results for your game.
2. **Frontend Framework:** A PixiJS/Svelte-based toolkit for creating the game's visual interface in the browser, integrating seamlessly with math outputs to ensure consistency between game logic and player experience [2].

By leveraging the **Carrot Remote Gaming Server (RGS)**, developers can integrate their games with Stake, uploading static game outcome files for verification and deployment [3] [4]. All game outcomes (spins) are precomputed and stored in compressed files, each mapped to a line in a lookup CSV with probability and payout columns [5]. When a player initiates a round, the RGS selects an outcome by weight and returns the corresponding sequence of events via the `/play` API [6]. This design ensures fairness, testability, and security, as every possible result is known and balanced in advance.

What Does the SDK Offer? In summary:

- **Full Math Engine:** Tools to define reels, symbols, paytables, and game logic, then run large-scale simulations to balance Return-to-Player (RTP) and hit rates [2].
- **Full Frontend Engine:** Tools to render the slot's UI (reels, symbols, animations) and handle game flow events in a browser, using modern web tech (PixiJS for graphics via WebGL and Svelte for UI components) [2].
- **Integration Support:** A defined format for output files and an RGS API interface to upload games and handle play sessions.

This guide explores **all publicly accessible components** of the Stake Engine SDK, including the Math SDK, Frontend SDK, technical details, file formats, architecture, example games, and more. By the end, you will have a clear understanding of how to develop a slot machine game efficiently with Stake's SDK, from configuring game math to implementing frontend visuals and integrating with the RGS.

Math SDK – Technical Overview

Engine Setup and Installation

Setting up the math SDK requires Python 3 (>= 3.12) and PIP, plus Rust/Cargo if you plan to use the optimization algorithm [7] [8]. The SDK repository can be cloned from the StakeEngine GitHub. You have two installation options:

- **Makefile (Recommended):** If you have GNU Make and Python3 installed, simply run `make setup` in the repository. This will create and activate a Python virtual environment, install all required packages (`requirements.txt`), and install the math SDK in editable mode [9] [10]. After configuring your game, you can execute simulations with `make run GAME=<game_id>` [11] (where `<game_id>` corresponds to a game directory).
- **Manual Setup:** Create and activate a Python virtual environment (`python3 -m venv env` then `source env/bin/activate` on Mac/Linux) [12]. Install dependencies with `pip install -r requirements.txt` and then install the SDK package in editable mode via `pip install -e .` [13] [14]. This editable install means any changes you make to SDK source code take effect without reinstalling. When done, deactivate the venv with `deactivate` [15].

Rust/Cargo: If you will use the optimization algorithm for balancing games, install Rust and Cargo before running simulations [16]. The Makefile's setup handles Python environment; for Rust, run the official installer (e.g. via `curl https://sh.rustup.rs | sh`) [8].

Once installed, verify by listing pip packages or importing the package in Python (e.g. `import math_sdk` – use your actual package name) [17]. After setup, you're ready to generate math results for a game.

Quickstart – Running a Sample Game

The SDK includes **several example games** under the `games/` directory to illustrate common slot mechanics [18]. For instance, `games/0_0_lines/` is a 5-reel, 3-row slot with 20 paylines (win-lines) [18]. To run its simulation and produce outputs, ensure you installed the SDK and then execute:

```
make run GAME=0_0_lines
# (Alternatively, activate the venv and run the script directly:)
python3 games/0_0_lines/run.py
```

This will generate all required math output files for that game [19]. By default, the run will use settings in `run.py` (e.g. number of simulations, whether to run optimization, etc.) – more on these shortly. All output files will be placed in the `library/publish_files/` directory of the game, including the compressed event data ("books"), lookup tables, and configuration JSONs [20]. *Note:* Even if you develop your own math code outside this SDK, you must still produce these output files in the correct format for Stake's platform [20].

Understanding `run.py` and Simulation Parameters

Each game directory has a `run.py` script which controls simulation parameters and orchestrates the simulation and optimization. Key settings in `run.py` include [21] [22]:

- `num_sim_args`: a dictionary defining how many simulations to run for each mode (e.g. `"base": 100000, "bonus": 100000` for large-scale runs). In testing, you can set smaller numbers (like 100 each) [21].
- `compression`: whether to output compressed `.jsonl.zst` books or human-readable `.jsonl` (for debugging). For debugging, set `compression = False` to get JSON output [21].
- `num_threads`: how many CPU threads to use for simulation. Large simulation batches can be split across threads for speed (e.g. 10 or 20 threads) [23].
- `run_conditions`: a dict toggling phases: `"run_sims"` (simulate outcomes), `"run_optimization"` (balance the weights/RTP), `"run_analysis"` (generate a PAR sheet analysis), and `"upload_data"` (if uploading to cloud) [24]. For a first run, you might disable optimization/analysis until basic logic is verified.

When you run simulations in **debug mode** (small `num_sim_args` and no compression), you can inspect the JSON output directly. For example, setting `"base": 100` sims and `compression=False` will output a file `books_base.jsonl` with 100 JSON entries (one per simulation) [21] [25]. Each entry has the following format:


```
{
    "id": 58,
    "payoutMultiplier": 10,
    "events": [
        { "index": 0, "type": "reveal", "board": [...], "paddingPositions":
[...], "gameType": "basegame", "anticipation": [...] },
        { "index": 1, "type": "winInfo", "totalWin": 10, "wins": [ {...} ] },
        { "index": 2, "type": "setWin", "amount": 10, "winLevel": 2 },
        { "index": 3, "type": "setTotalWin", "amount": 10 },
        { "index": 4, "type": "finalWin", "amount": 10 }
    ],
    "criteria": "basegame",
    "baseGameWins": 0.1,
    "freeGameWins": 0.0
}
```


This example (simulation ID 58) shows a base game spin with a 10x payout [26] [27]. The list of `events` is critical – it contains everything the frontend needs to display the result: a `reveal` event with the final board symbols, a `winInfo` event listing what was won (here, one line win of 10x on symbol "L5"), and then events setting the win amounts and marking the final win [28] [29]. The `criteria` field indicates which **distribution criteria** this spin fulfilled (here `"basegame"`, meaning a normal win in base play), and `baseGameWins` vs `freeGameWins` show how the payout splits between modes (useful if a base spin triggers free spins) [30]. In this case, all 10x came from the base spin (0.1 = 10/100 in decimal RTP contribution) [31].

Notice that in the lookup table CSV (which maps simulation IDs to outcomes), simulation 58's entry might look like `58,1,10` – meaning ID 58 has weight 1 and payout 10x [31]. Initially, all simulations have weight 1. The **optimization algorithm** later adjusts those weights to achieve the target RTP [31]. For now, in our raw simulation, every outcome is equally likely.

Iterating Simulations and Optimization

After verifying a few sample outputs, you'll typically run a much larger simulation batch (e.g. 100k+ spins per mode) to generate a rich outcome set and ensure statistical stability [32]. The SDK supports multi-threading these runs. For example, you might set `num_sim_args = {"base": int(1e5), "bonus": int(1e5)}` and `num_threads = 20` to run 100k base and 100k bonus spins across 20 threads [23]. You would also set `run_optimization: True` to invoke the RTP balancing step after simulation [33]. As threads finish, the console will print interim RTPs, e.g. `Thread 0 finished with 1.632 RTP. [baseGame: 0.043, freeGame: 1.588]` [34]. This indicates that in raw simulations, the free spins were paying far above the intended RTP (163.2% total here, mostly from freegame) [35]. That's expected when we *force* rare events more often for testing; the optimizer will later adjust weights to reach the desired overall RTP (~97% typically) [36].

When `run_analysis: True`, the SDK also produces a PAR sheet summarizing stats like hit rates for different win types [37]. This uses the segmented lookup tables (like `lookUpTableSegmented_base.csv` which breaks down what portion of RTP came from various features) in conjunction with paytable and force files to compute frequency of events, average wins, etc [37]. These analysis outputs help verify the game's behavior (e.g. frequency of free-spin triggers, distribution of wins).

Finally, after achieving satisfactory results, you might integrate the outputs with the frontend. The SDK's example frontend (Storybook) can directly consume the math outputs for testing [38]. You can also start creating your own game by copying one of the sample game directories as a template and modifying its logic and configuration [39]. Generally, game-specific values (symbol payouts, reel strips, feature trigger counts, etc.) live in `game_config.py`, while unique game logic resides in `game_calculations.py` and custom event handling in `game_events.py` [39]. Reusable functionality should stay in the core `src/` modules, whereas one-off game behavior goes in the game's own files [40]. We will cover the directory structure and file roles next.

Math SDK Directory Structure

The math SDK repository is organized into clearly defined directories, separating engine logic from game-specific content [41] [42]:

- `games/` – Contains sample game implementations demonstrating various mechanics [43]. For example:
- `0_0_cluster`: Cascading *cluster-pays* game (wins for adjacent symbols) [44].
- `0_0_lines`: Basic lines (payline) game [45].
- `0_0_ways`: Basic ways-pay game (243 ways) [46].
- `0_0_scatter`: Pay-anywhere scatter game (like cluster but anywhere on board) [47].
- `0_0_expwilds`: Game with expanding wild reels and a special prize-collection feature [48].
- `src/` – The core math engine code shared by all games (edit these with caution, as they affect all games) [42]. Key subdirectories:

- `calculations/` – Board setup logic and win evaluation algorithms for different win types (lines, ways, cluster, etc.) [49].
- `config/` – Code for generating configuration files (for RGS, frontend, optimization) [49].
- `events/` – Data structures and functions for events passed from math to frontend [49].
- `executables/` – Common game logic sequences grouping calculations and events (e.g. functions to execute a spin or trigger free spins) [50].
- `state/` – Manages game state during simulation (GameState classes, see below) [50].
- `wins/` – The "wallet" management (win accumulation, RTP tracking) and win criteria logic [51].
- `write_data/` – Handles writing simulation outputs to files, compression, and generating force files for event tracking [52].
- `utils/` – Utility modules for simulation analysis and other helpers [53]. For instance:
- `analysis/` – Functions to analyze win distributions and properties from simulation data (e.g. to produce hit rate tables) [54].
- `game_analytics/` – Uses recorded events, paytables, and lookup tables to compute statistics like feature frequency and average wins [55].
- `tests/` – Basic PyTest tests for core functionality [56] (e.g. verifying that line win calculations yield expected results in `tests/win_calculations/` [57]).
- `uploads/` – Tools for uploading game files to an AWS S3 bucket for testing (if not using direct RGS upload). It contains the method to push outputs to cloud storage [58] [59].

- `optimization_program/` – An **experimental genetic algorithm** (written in Rust) for balancing discrete outcome games [60] [61]. This compiles to a binary used by the math SDK to adjust the lookup table weights.

- `docs/` – Markdown documentation (for internal reference; not needed for game development).

Each game (in `games/<game_name>/`) follows a recommended structure, which you can use as a template for new games [62]:

```
game/
├── library/
│   ├── books/            # Uncompressed outcome logs (JSONL)
│   ├── books_compressed/ # Compressed outcome logs (.jsonl.zst)
│   ├── configs/          # Generated config JSONs (frontend, backend, math)
│   ├── forces/           # Force records of custom events
│   ├── lookup_tables/    # CSV summary tables of outcomes
├── reels/                # Reel strip CSV files (if any)
├── readme.txt            # Developer notes for the game
├── run.py                # Simulation runner script for this game
├── game_config.py        # Game-specific Config subclass (parameters)
├── game_executables.py   # Game-specific Executables subclass (custom logic)
├── game_calculations.py  # Game-specific Calculations subclass (custom logic)
├── game_events.py        # Game-specific Events (custom event definitions)
└── game_override.py      # Game-specific GameStateOverride subclass
```

When you run a simulation, any needed `library/` subfolders that don't exist (books, lookup_tables, etc.) will be auto-created [63]. The `readme.txt` can contain notes about the game's mechanics for future reference [63].

The game files (`game_config.py`, `game_executables.py`, etc.) allow you to **override** or extend the core engine for unique game behaviors [64]. For example, if your game has an extra bonus feature or a different free-spin trigger, you implement that by overriding methods in these game files, rather than altering the core `src/` code. This approach keeps the engine extensible: the core provides common functionality, and games can plug in their specifics as needed [64]. We will detail many of these classes (GameState, Config, Executables, etc.) in the following sections.

High-Level Architecture: State Machine and Game Structure

At the heart of the math engine is the **GameState** – a state machine that orchestrates simulations. The `GameState` class (and its subclasses) manage all aspects of a simulation batch: simulation parameters, game configurations, and accumulation of results [65] [66]. When you run `run.py`, it creates a Config object (with all your game settings) and then creates a GameState instance to coordinate the run [67].

The GameState State Machine

The GameState serves as the *central hub* for a game simulation, with key responsibilities in two categories [68] [69]:

- **Simulation-Level Settings:** Manages compression on/off, tracing flags, multi-thread coordination, output file writing, and the overall win manager for cumulative stats [70]. These ensure all simulations in a batch adhere to the same conditions (e.g. compressed output, number of threads, etc.).
- **Game Configuration:** Holds global game data like bet mode definitions, paytable, symbol list, reel strips, etc., which remain constant across all simulations [69]. GameState is initialized with this config and makes it easily accessible during each spin.

Importantly, when `GameState.run_spin()` is called for a particular simulation, the engine actually creates a **sub-instance** of a `GeneralGameState` (a subclass) for that spin [71]. This allows each spin to modify its own state (reels, symbol positions, etc.) via `self` without interfering with other spins. It also avoids having to pass large state objects between functions – everything for that spin is encapsulated in `self` within the spin's context [71].

A high-level flowchart of the math engine's operation is illustrated in the documentation [72]. *(Diagram omitted here)*. In simple terms:

1. **Initialization:** `GameState` is created with the game config. It sets up output containers (like an empty `book` to record events) and distribution quotas for the simulation.
2. **Simulation Loop:** For each simulation ID, `run_spin()` is executed. Inside it, the game draws a board, evaluates wins, triggers any features (like free spins), and repeats if certain criteria aren't met (explained in "Simulation Acceptance" below).
3. **Finalize:** After all spins, the GameState writes out the accumulated results (books, lookup tables, etc.), possibly runs the optimizer, and produces final files for upload.

Extending Core Functionality: The SDK is designed so that **GameState** can be extended via inheritance and Python's Method Resolution Order (MRO) [73]. The base `GameState` class (defined in `src/state/state.py`) provides core methods (resetting state, running spins, etc.). Each game can override these by providing a `GameStateOverride` class in `game_override.py` which inherits from `GameState` and changes certain methods. The SDK ensures that the game-specific override is first in MRO, so its versions of methods take precedence [74] [75]. We'll see examples of this pattern soon.

After simulations complete, the GameState sequentially generates the output files for each bet mode (books, CSVs, config JSONs) which can then be optimized and uploaded [74]. This step is done automatically by the `create_books()` and `generate_configs()` calls at the end of `run.py` [76] [77].

GameState Inheritance and Game Overrides

Each game typically defines three classes in its Python files to override parts of GameState logic [75] [78]:

- **GameStateOverride (in `game_override.py`):** This class inherits `GameState` and is placed first in the MRO. It is used to modify or extend base state behaviors without altering the engine for all games [75]. For example, many sample games override the `reset_book()` method to reset some game-specific counters or flags each spin [78]. A typical override might call the base implementation then set custom properties:

```
def reset_book(self):
    super().reset_book()
    self.reset_grid_mults()
    self.reset_grid_bool()
    self.tumble_win = 0
```

  Here, after resetting the book (which clears out last spin's data), the game's override resets some grid multipliers/flags and a `tumble_win` accumulator used for its cascading logic [79] [80].
- **GameExecutables (in `game_executables.py`):** Inherits from the core `Executables` class and groups common game actions into functions [81] [82]. Overriding these allows customizing sequences like how free spins are triggered or how special wins are handled. For example, the base engine might define a generic `update_freespin_amount()` that triggers 8, 10, or 12 free spins for 3, 4, 5 scatters [83]. A particular game (like the scatter-pay sample) might override this to a different rule (e.g. 2x number of scatters):

```
def update_freespin_amount(self, scatter_key="scatter"):
    self.tot_fs = self.count_special_symbols(scatter_key) * 2
    fs_trigger_event(self, basegame_trigger=basegame_trigger,
freegame_trigger=freegame_trigger)
```

  This ensures that, in that game, free spins awarded = 2 × number of scatter symbols, instead of the default mapping [84].
- **GameCalculations (in `game_calculations.py`):** Inherits from core `Calculations` and is meant for any complex, game-specific calculation logic [85]. Often, this class will itself inherit from

GameExecutables (meaning it has all those methods too), and you might not need to override much here unless your game introduces entirely new types of symbol interactions or bonus calculations beyond what's in Executables.

Using these classes, you can tweak the engine's behavior for your game's needs. The general guideline is: **if the engine already supports a feature via configuration, use config; if not, override in these classes**. We will touch on specific override points (like special symbol handling, repeat spin conditions, etc.) in context below.

Game "Books" and Libraries

Throughout simulation, GameState keeps a **"book"** – a record object for the current simulation result [86]. A "book" contains the simulation's payout and all events that occurred (as we saw in the JSON example) [87]. Each simulation yields one book, and collectively they form a **library** for the batch [88]. The library (attached to GameState) is essentially the list of all books, and it's what gets written to the output files.

The book is reset at the start of each spin. The default `reset_book` sets up a fresh dictionary:

```
self.book = {
    "id": self.sim + 1,
    "payoutMultiplier": 0.0,
    "events": [],
    "criteria": self.criteria
}
```

as shown in the engine code [89]. This ensures each simulation starts with zero payout and an empty event list, ready to be populated.

After all simulations, the library of books is used to generate: - The **books file** (JSONL or compressed) containing each book. - The **lookup table** CSV summarizing each book's payout (and initial weight). - The **force** files and analysis for events (if used).

In short, *book* = one round's result, *library* = all rounds.

Lookup Tables and RTP

The **lookup table** (`lookUpTable_<mode>.csv`) is a simple but crucial summary: each row has *Simulation Number, Simulation Weight, Payout Multiplier* [90] [91]. Initially, `Simulation Weight` = 1 for all. The **payoutMultiplier** in the book is the final outcome of that simulation (e.g. 1150 means 1150× bet) [92] [93]. The lookup table thus provides a quick view of all outcomes and their frequencies.

Why are lookup tables useful? They allow: - **Win Distribution Analysis:** You can quickly see the distribution of payouts (by grouping identical multipliers or ranges) [94]. - **RTP Calculation:** Summing (Weight × Payout) across the table and dividing by total sims yields the return-to-player. The engine and RGS use these tables to compute RTP and verify it matches expectations [94]. - **Optimization:** These tables serve as input to the optimizer, which will adjust the `Weight` column to achieve a target RTP or distribution shape [95].

The SDK differentiates between **initial** vs **optimized** lookup tables by naming convention: initial runs produce `lookUpTable_mode.csv` and the optimizer outputs an `lookUpTable_mode_0.csv` (and possibly further numbered versions for iterative passes) [96]. In summary, before optimization all weights are 1, and after optimization, weights are tuned such that the weighted average payout meets the desired RTP [97].

The lookup table's completeness ensures that *every possible outcome is accounted for*. This is required by Stake's upload criteria – all game outcomes must be in these static files for verification [6].

Gamestate Structure and Simulation Acceptance Criteria

We now delve into how a single spin is structured and when a simulation is accepted or repeated. Stake Engine uses a clever system of **distribution quotas** and a `repeat` loop to ensure your simulation batch meets certain quotas (like a certain percentage of spins being free-spin triggers, max wins, etc.) [98] [99].

Distribution Criteria and Quotas

In your `GameConfig`, each BetMode can be split into different **win criteria segments** for simulation. For example, you might want 0-win spins, regular wins, free-spin triggers, and max-win outcomes all well-represented in your simulation data [100]. The SDK allows defining multiple `Distribution` entries per BetMode, each with: - A **criteria name** (shorthand, e.g. `"basegame"`, `"freegame"`, `"wincap"` for max win) [101]. - A **quota** – the target ratio of simulations that should meet this criteria [101]. Quotas are normalized so they need not sum to 1; e.g. you could set quota 0.001 for max-win (0.1%) and 0.999 for everything else and the system will allocate appropriately [102]. At least 1 simulation will be allocated per criteria regardless [102]. - A set of **conditions** that force certain behavior to achieve that criteria [103]. The required keys are: - `reel_weights`: which reelstrip to use for each gametype (base or free) – often used to bias certain symbols [104]. - `force_wincap`: (bool) whether to force the max win in this spin [104]. - `force_freegame`: (bool) whether to force a free-game trigger in this spin [104]. - (Any number of other condition keys can be used as needed, e.g. specific `mult_values` to force certain multipliers, or `scatter_triggers` to specify how many scatters appear) [105] [106].

For example, consider a Bonus Buy mode with two distribution criteria: - `"wincap"` with quota 0.001, conditions forcing both a freegame trigger and the max win (uses special reel weights, forces freegame, etc.) [107] [108]. - `"freegame"` with quota 0.999, conditions forcing freegame but not necessarily max win (with different symbol multiplier weights, etc.) [109] [110].

In code, these might appear in `GameConfig.bet_modes` as shown in the sample (a bonus mode where 0.1% of spins are forced maximum win and 99.9% are normal freegame triggers) [111] [106]. The **purpose** of doing this is to ensure your simulation includes enough rare events (like max wins) to analyze them properly, rather than relying on pure probability which might yield zero occurrences in a finite run [112] [113].

Without quotas, if a max win naturally has a 1 in 500,000 chance, a 1 million spin simulation might only see ~2 of them [112]. By pre-assigning quotas, you could instead force, say, 200 max-win spins to study their behavior, then later weight them appropriately with a small probability [113]. This technique of **predetermining acceptance** greatly improves simulation efficiency for rare events.

Simulation Acceptance and Repeat Logic

Each simulation number is **pre-tagged** with a criteria according to the quotas before simulations begin [99]. For instance, simulation #10 might be marked as a "freegame" criteria spin, #500 as a "max-win" spin, etc., to meet the distribution quotas across the whole run [99]. The GameState's `check_repeat()` function at the end of each spin then decides whether the spin satisfied its criteria or needs to be repeated [114].

For example: - If simulation #10 was supposed to trigger a freegame (criteria `"freegame"`), but when we ran the spin it didn't drop enough Scatters, then `check_repeat` will detect that and set `self.repeat = True` to redo sim #10 [114]. - If a simulation's criteria was `"0"` (no-win) but it ended up paying something, it will repeat [114]. - If criteria was `"wincap"` (max win) but the max payout wasn't achieved, repeat until it is.

The `self.repeat` flag is initially set true at spin start (in `reset_book()` it's set False, but immediately many games set it True before spinning) [115]. The spin runs in a `while self.repeat:` loop [116] [117]. When the spin outcome matches the predetermined criteria, `check_repeat()` sets `self.repeat = False`, allowing the loop to exit and the simulation to be *accepted* [114]. If not, it resets and tries again (drawing a new board, etc.). This loop ensures each simulation number yields the type of outcome it was intended for.

A caveat: more stringent criteria can cause more repeats and slow down simulations if the condition is hard to meet by chance [118]. That's why, for extremely unlikely outcomes like max wins, developers often **force** them via special reelstrips or conditions. For example, you might create a special reel (see `reel_weights`) that has only high-paying symbols to help naturally achieve a max win [119]. The Distribution conditions allow biasing the random outcome generation to meet the criteria more efficiently [120] [121].

In summary, the **Simulation Acceptance Criteria** system provides fine control over your simulation content: - You decide categories of outcomes (0-win, feature trigger, jackpot, etc.) and what fraction of spins they should occupy [98]. - The engine tags each spin and repeats it until the outcome matches the tag [99] [114]. - This yields a rich simulation set that deliberately contains the rare scenarios needed for balancing and testing, rather than leaving it to pure luck.

Predetermining Acceptance vs Post-Filtering

It's worth noting that one might think to simulate everything and then categorize outcomes after the fact. However, because of multithreading, if one category is much harder to hit, threads would finish unevenly (others done while one thread keeps grinding for that last max win) [122]. Preassigning quotas avoids this by making each thread handle a proportional share of each criteria from the start [123]. This leads to better parallelism and ensures the last few simulations don't all belong to the hardest category [123]. The engine's approach, using the `quota` and repeat system, elegantly solves that problem.

Configuration Files and Sections (GameConfig & BetModes)

Now we shift focus to how you **define a game's rules and data** in the SDK. This is done in the `GameConfig` class (in your `game_config.py`), which inherits the core `Config` class [124]. In

`GameConfig.__init__()`, you specify all the game parameters – name, RTP, reel layout, symbol payout values, special symbol behavior, etc. [125] [126]. Let's break down the important sections of `GameConfig`:

- **Basic Game Info:** At minimum, set `self.game_id` (string identifier), `self.provider_number` (if provided by Stake), `self.working_name` (human-readable name, optional), `self.rtp` (target return to player in decimal, e.g. 0.96), and `self.wincap` (max win multiplier, e.g. 5000 for 5000×) [127]. Also specify `self.num_reels` and `self.num_rows` (list of row counts per reel, or a single number if uniform) to define board dimensions [128].
- **Win Type and Paytable:** Choose a `self.win_type` (e.g. `"lines"`, `"ways"`, `"cluster"`) to indicate how wins are evaluated [126]. Then define `self.paytable` as a dictionary mapping `(kind, symbol)` to payout. For example: `{(3,"A"): 5.0, (4,"A"): 10.0, (5,"A"): 50.0, ...}` to pay 50× for five A's on a line [129]. The `kind` is the count of symbols in the win. If your game uses ranges (like 12+ symbols same payout), you can define a `pay_group` with range tuples and call `self.convert_range_table(pay_group)` to auto-expand it into a full paytable [130] [131].
- **Symbols Setup:** The engine uses your paytable and a `special_symbols` dict to determine valid symbols. `self.special_symbols` groups special symbol names by attribute, e.g. `{"wild": ["W"], "scatter": ["S"], "bonus":["B"]}` [132] [133]. Any symbol not listed in paytable or special_symbols will cause a load error if found on a reel strip [134]. So, include all symbol codes in either paytable or special list. Special symbols can have multiple attributes and aren't necessarily paying symbols (e.g. a scatter might trigger free spins and have its own pay).
- **Reel Strips:** Provide reel strips in CSV files under the game's `reels/` directory, then load them in `GameConfig`. For example, you might list reel files `BR0.csv`, `BR1.csv` for base reels, and `FR0.csv` for free-spin reels. In config, you set:

```
reels = {"BR0": "BR0.csv", "BR1": "BR1.csv", "FR0": "FR0.csv"}
self.reels = {}
for r, file in reels.items():
    self.reels[r] = self.read_reels_csv(self.reels_path + "/" + file)
```

  which loads each CSV into `self.reels` dict [135] [136]. You will also use these keys in Distribution `reel_weights` to pick which reel set to use for a given spin (for example, one distribution might weight `{"BR0":2, "BR1":1}` meaning 2/3 chance to use BR0, 1/3 BR1 for base game) [137].
- **Scatter Triggers & Anticipation:** If your base game triggers free spins with scatters, define `self.freespin_triggers` as `{self.basegame_type: {<n_scatters>: <free_spins>}, self.freegame_type: {...}}` [138]. For example, in basegame 3 scatters → 10 free spins, 4→15, 5→20; and perhaps in freegame, 2 scatters retrigger 4 spins, 3→6, etc. This mapping is used by the engine to automatically award free spins and by the event system to display "Scatter anticipation" (i.e. slow down reels if the first reels have two scatters and one more would trigger) [138]. The engine generates an `anticipation` array indicating on which reels to add delay [139]. For example, if 3 scatters are needed and reels 0 and 1 already landed scatters, the anticipation array might be `[0,0,1,2,3]` for a 5-reel game, meaning reel2 has a slight delay, reel3 more, reel4 even more, building excitement [140].
- **Special Symbol Functions:** If special symbols have dynamic properties (like a Wild that can carry a random multiplier), you can assign a *function* to run when that symbol is created. In `GameState.assign_special_sym_function()` (which you override in game override class), you

populate `self.special_symbol_functions` with mapping from symbol name to a function [141] [142]. For example,

```
self.special_symbol_functions = { "W": [self.assign_mult_property] }
```

and define `assign_mult_property(self, symbol)` to randomly set `symbol.multiplier` value [143]. The SDK will call this for each 'W' symbol as it's created. In the sample, they get a random multiplier from current distribution conditions (`mult_values`) and assign it to the symbol object [144] [145]. This way, e.g., each Wild on the board might get a random 2×,3×,5× multiplier as defined by distribution. Such functions are game-specific and allow rich symbol behaviors beyond static attributes.

- **Bet Modes:** The `self.bet_modes` list defines all BetMode instances for the game. Each BetMode includes attributes like `name` (e.g. "base" or "bonus"), `cost` (e.g. 1.0 for base bet, 100.0 for a bonus buy that costs 100×), `rtp` (usually equal for all modes unless specified), `max_win` (use `self.wincap`), and the flags `auto_close_disabled`, `is_feature`, `is_buybonus` as needed [146] [147]:
- `auto_close_disabled`: If False (default), the RGS will auto-close the round after play for efficiency. If True, the front-end is expected to manually send an `/endround` (this is often set True for bonus features so a 0 win doesn't auto-close, allowing the player to resume if connection lost) [146] [148].
- `is_feature`: If True, this mode doesn't require player action to continue – if a game enters this mode, the front-end will persist in it without waiting for bets (good for free spin mode so that spins auto-play until done) [149].
- `is_buybonus`: True if this mode is a purchased feature entry (so the frontend can label it or handle assets differently) [150].

Each BetMode also has a `distributions` list of Distribution conditions as discussed earlier [151] [152]. For example, the base BetMode might have distributions for `"0"` (zero win), `"basegame"` (normal win), `"freegame"` (triggers free spins), etc., each with a quota and conditions (like forcing free spins on the `"freegame"` criteria) [152] [153]. These distributions tie back into the Simulation Acceptance logic above.

All these configuration elements come together such that when the **GameState** is initialized: - It copies these values in (e.g. into `self.paytable`, `self.reels`, etc.). - It uses them to configure the BetModes and criteria (like preparing the `sim_to_criteria` mapping for repeats). - It ensures all necessary inputs are present (the base `Config` class likely validates fields and sets up file paths for output, etc.) [154] [155].

Summary of Config capabilities: The `Config` class also allows defining **win levels** for categorizing wins (used by events like `set_win_event` to indicate a "Big Win" or other animation category) [154], and it sets up output file paths and reel file verification logic. Essentially, your GameConfig provides all the *static data* for your game, while the dynamic behaviors are handled in GameState and Executables.

The Simulation Loop: Spin, Features, and Repeat

To ground all this, let's walk through what happens during a **single simulation (spin)**, in terms of engine calls and game logic, using the structure in `GameState.run_spin()`:

Running a Base Spin (`run_spin` logic)

A generic `run_spin(self, sim)` function does roughly the following in pseudocode (the actual SDK provides a template in the documentation) [156] [157]:

```
self.reset_seed(sim)            # Seed RNG for reproducibility
self.repeat = True
while self.repeat:
    self.reset_book()          # Prepare a fresh record for this attempt
    self.draw_board()          # Randomly generate a board (spin the reels)
    # Evaluate wins on the board:
    ... # e.g., win_data = self.get_lines() or get_scatterpay_wins()
    ... # and update wallet:
self.win_manager.update_spinwin(win_data["totalWin"])
    ... # and emit win events: self.emit_linewin_events(), etc.
    self.win_manager.update_gametype_wins(self.gametype)  # track base vs
freegame portion
    if self.check_fs_condition():
        self.run_freespin_from_base()   # trigger free spins if scatter
condition met
    self.evaluate_finalwin()       # compute final total win for this spin
    self.check_repeat()           # see if criteria satisfied; sets self.repeat
False if yes
# end while
self.imprint_wins()               # finalize the win into cumulative stats
```

Let's break that down in terms of what core functions do and what you as a developer might customize:

- **Seeding**: The RNG is seeded with the simulation number (`sim`) to ensure determinism [158]. This means if you re-run the same simulation number with the same config, you'll get the same outcome, which aids debugging and verifying specific outcomes.
- **reset_book**: Clears last attempt's events and sets `self.book` for this simulation [159] [160]. Note the game's `GameStateOverride.reset_book` might set additional flags or counters to initial state (e.g. `self.tumble_win = 0` as shown earlier) [78].
- **draw_board**: This calls the `Board.create_board_reelstrips()` or similar, to select a reelset (accounting for `reel_weights` condition) and choose random stop positions for each reel [161] [162]. It then creates the `self.board` as a 2D array of Symbol objects for the visible grid [161]. It also fills `self.reel_positions` (the index stops chosen) and `self.padding_positions` (the symbols just above/below the visible board if `include_padding=True`) for use in reveal events [163]. If scatter anticipation applies, it computes the `self.anticipation` array (with incremental delays as described) [140]. By default, `draw_board` emits a `reveal_event` (unless `emit_event=False` when called) which will later be sent to the frontend to display the spun symbols [164].
- *Developer note:* If you need to force a specific board (for testing or as part of a feature), the engine provides `force_board_from_reelstrips(reelset_id, positions)` in Executables [165]. This can be used in special cases, e.g. if a feature guarantees a wild reel, you might force that reel's stop to a position with all Wilds.

- **Evaluate wins**: After the board is set, the engine calculates wins. Depending on `win_type`, it will call the appropriate function:
- For lines: `win_data = self.get_lines()` which uses `LinesWins` to find any payline wins [166] [167].
- For ways: `win_data = self.get_ways()` which finds all combinations of symbol on consecutive reels [168] [169].
- For cluster: `win_data = self.get_cluster_wins()` which finds groups of adjacent symbols.
- For scatter: `win_data = self.get_scatterpay_wins()` which counts symbols anywhere.

These functions return a structure `win_data = {'totalWin': X, 'wins': [ {symbol, kind, win, positions, meta{...}}, ...]}` [170] [171]. The `wins` list contains each distinct winning combination and its info. The `meta` might include things like line index or multiplier contributions [171] [172]. For example, a line win entry's meta includes which payline it was on and any line-specific multiplier [173]. A scatter win's meta might include an `'overlay'` position (for where to center the animation) [174]. Ways wins meta includes the count of ways [175].

After computing win_data, the engine triggers **wallet updates** and **win events**: - `self.win_manager.update_spinwin(win_data["totalWin"])` adds this spin's win to the running total for the bet [176]. - It then likely calls an event emitter like `self.emit_linewin_events()` (or emit_scatter_win, etc.) which generates one or more `winInfo` events describing each win and a `setWin` event for the spin win [177]. For example, after a lines win, an event with `"type":"winInfo", "wins": [...], "totalWin": X` is added to explain what was won [178] [179].

- **Update gametype wins**: After calculating base game wins, the call `update_gametype_wins(self.gametype)` will add the spin's amount to either the basegame or freegame running sum in the wallet manager [157]. If this spin *triggered* free spins, those free spin outcomes will accumulate separately in `freegame_wins` – the wallet tracks how much of the total came from base vs free for transparency [30].
- **Free spin trigger check**: `if self.check_fs_condition(): self.run_freespin_from_base()`. Here, `check_fs_condition` likely checks if the number of scatter symbols on this board ≥ the minimum defined in `freespin_triggers` [180]. If yes, `run_freespin_from_base` is called to transition into the free spin mode [181]. In `GameExecutables.run_freespin_from_base`, typically it will:
- Call `self.record({...})` to log this trigger in the force file (e.g. record how many scatters triggered it) [182].
- Call `self.update_freespin_amount()` to set `self.tot_fs` (the total number of free spins awarded) based on the scatter count [183].
- Finally, call `self.run_freespin()` to actually execute the free spins round.

The free spins (`run_freespin`) function runs similarly to base spins but in a loop for each free spin:

```
self.reset_fs_spin()
while self.fs < self.tot_fs:
    self.update_freespin()
# increment spin count and emit a "freespin start" event
```

```
    self.draw_board()         # spin reels with freegame reelset
    ... evaluate wins, update wallet ...
    if self.check_fs_condition():
        self.update_fs_retrigger_amt()  # if retrigger, increase tot_fs
    self.win_manager.update_gametype_wins(self.gametype)  # accumulative
freegame win
self.end_freespin()           # emit free spins end event
```

Important points: - `reset_fs_spin()` prepares for free spins (e.g., sets `self.fs = 0` and any free spin specific state) [184] . - `update_freespin()` increments the `self.fs` counter and emits an event (often called `"updateFreespin"` event) to inform the UI of the free spin count [185] [186] . - `draw_board()` uses the freegame reel strips (the engine automatically uses `self.freegame_type` to pick reels like "FR0") [185] . - Wins are evaluated similarly; after each free spin, `check_fs_condition` can catch retriggers (e.g., scatters during free spins) and `update_fs_retrigger_amt` will increase `self.tot_fs` accordingly [187] [188] . - After all free spins (`self.fs` reaches `self.tot_fs`), `end_freespin()` emits an event signaling the feature ended (often used to display summary or transition back to base game) [189] [190] .

The base and freegame flows are kept separate through `self.gametype` (likely set to "basegame" or "freegame") and separate wallet tracking. All free spins events and wins also get recorded in the same book events list, but they are distinguished by an `"gameType": "freegame"` field in events like `reveal` [191] . This way, the frontend knows those events happened in the bonus round.

- **Finalize spin**: After base (and any free spins), `evaluate_finalwin()` tallies the total payout for this simulation (including base + feature wins) and likely logs a `finalWin` event [192] . It might also check if the max win (`wincap`) was hit and set a flag `wincap_triggered` (used to stop free spins early, etc., if needed).
- **Repeat check**: As described, `check_repeat()` will compare the outcome to the criteria. If criteria was not met, the function will *not* set `self.repeat` to False (meaning the `while` continues), effectively discarding the spin and trying again [114] . If the criteria was satisfied, `self.repeat` is set False allowing the loop to exit. Note that `check_repeat` can check multiple conditions – e.g., if a spin needed to be both a freegame trigger and a max win, it would verify both.
- **Record the win**: When a spin is accepted, `self.imprint_wins()` is called [115] . This finalizes the simulation's data: it appends the `book` (with all events and payout) to the GameState's library, updates cumulative counters in the WinManager (`total_cumulative_wins`, etc.), and resets any temp storage for force records. Specifically, `update_end_round_wins()` in WinManager adds this round's base/free wins to running totals and prints thread RTP info if in multi-thread mode [193] .

This entire sequence happens for each simulation ID in each thread via the `run_sims()` function of GameState [194] . `run_sims` essentially loops over the assigned simulations (split by thread) and calls `run_spin(sim)` for each, managing threading and repeats.

By following this design, you can see how **separation of concerns** is achieved: - **Config** defines what should happen (paytables, triggers, quotas). - **GameState/Executables** define when/how to execute these actions (spin reels, start feature). - **WinManager** tracks the money. - **Events module** logs everything that happened in a structured way.

Required Math Output File Formats

After running simulations (and optimization, if enabled), the SDK produces three types of files which you will upload to Stake's RGS, as mentioned earlier: **Index JSON**, **Lookup Table CSV(s)**, and **Game Logic JSONL** (often called "books files"). These must follow strict formats for the RGS to accept them [195] [196]. Let's recap each:

- **Index File (`index.json`):** Lists the game modes in the package and their file names [197]. Stake requires this file at the root of your game package with exactly this structure:

```
{
    "modes": [
        {
            "name": "<mode_name>",
            "cost": <cost_multiplier>,
            "events": "<logic_file>.jsonl.zst",
            "weights": "<lookup_table>.csv"
        },
        ...
    ]
}
```

For example, for a game with a base mode and a bonus mode, index.json might contain:

```
{
  "modes": [
    { "name": "base", "cost": 1.0, "events": "books_base.jsonl.zst",
"weights": "lookUpTable_base_0.csv" },
    { "name": "bonus", "cost": 100.0, "events": "books_bonus.jsonl.zst",
"weights": "lookUpTable_bonus_0.csv" }
  ]
}
```

The `"cost"` is the bet cost multiplier (so 100.0 means the bonus buy costs 100× the base bet) [198] [199]. The filenames must match exactly your output files. The index.json is how the RGS knows which files correspond to which mode (and the cost for each mode).

- **Lookup Table CSV:** Each mode has a CSV summarizing outcomes. The format is **three columns, no header**: `simulation number, round probability, payout multiplier` [92]. The probability column is typically an integer representing the weight or count of that outcome (in practice these are scaled up to avoid fractions – the SDK uses large integers such that dividing by sum gives the probability) [92]. For example:

```
1,199895486317,0
2,25668581149,20
3,126752606,140
...
```

This means simulation 1 has weight 199895486317 and pays 0, sim 2 weight 25668581149 pays 20, sim 3 weight 126752606 pays 140, etc. [200]. **Important:** The payout values in the third column must exactly match those in the game logic file for the same simulation IDs [92]. The RGS will hash and cross-verify that every `payoutMultiplier` in the books appears in the CSV to ensure integrity [92]. By using only non-negative integers (payouts are multipliers, never negative) in the CSV, the system avoids any rounding issues – everything is integer-based for safety [201].

If optimization was run, you will typically use the optimized table (e.g. `lookUpTable_base_0.csv`) in the index, which has adjusted weights summing to a value corresponding to 100% probability at your target RTP [96]. If not optimized, the initial table (with all weights 1) might sum to an RTP higher than desired (because forced events inflate it), so optimization is expected in production.

- **Game Logic file ("events" JSONL):** This is the large file containing one JSON object per simulation, detailing the event sequence (the "book"). It must be in **compressed JSON-Lines with zstandard compression** (file extension `.jsonl.zst`) when uploaded [202]. Each JSON entry must include at least the keys `"id"`, `"payoutMultiplier"`, and `"events"` [203] [204]. The SDK ensures these keys are present for every round [205]. The example minimal format given is:

```
{
    "id": 1,
    "events": [ {}, ... ],
    "payoutMultiplier": 1150
}
```

This means simulation 1 had a 1150× payout [206]. In practice, as we saw, there will be actual event objects in the events list describing what happened. The **RGS /play API** essentially fetches the appropriate JSON object (by simulation ID) from this file and returns it to the frontend [207]. That is why everything needed to render the outcome (reveals, wins, etc.) must be in those events.

A **minimum math package** for a single-mode game thus includes: one index.json, one lookup CSV, one compressed book file [195] [208]. Multi-mode games include one of each per mode. If any file is missing or misnamed, the upload will be rejected. The Stake Engine performs checks on these math files: - It will parse index.json and attempt to load each referenced file [209]. - It verifies that for every entry in the events file, there's a matching line in the CSV and vice versa (payouts exactly match) [92]. - It may also compute a quick RTP from the CSV and display it for reference on the backend [209]. - If anything doesn't match format or the values seem off (e.g. negative payout which shouldn't happen), it fails the validation.

In short, **follow the format strictly**. The SDK's `generate_configs()` will output a correct index and config files for you, so use those as-is [210] [211]. Do not edit the weights or events files manually, as it could break the hashes. If you need to tweak RTP, re-run the optimization tool rather than editing CSV by hand.

Math SDK Source Modules – Key Components

We've mentioned various source files (Board, Tumble, Events, Executables, State, WinManager, etc.). Here we summarize their roles and highlight any important functions or code snippets a developer should know when extending the engine:

- **Board (src.calculations.board):** Handles generating the game board layout each spin. The `Board` class inherits from GameState (actually from a `GeneralGameState` helper) and provides `create_board_reelstrips()` which picks a reelset and stopping positions for each reel [161] [162]. It then constructs the symbols via the Symbol class:
- The `Symbol` class (in src.events or src.symbols) takes the `name` and checks it against config.special_symbols. Any matching special property is set as an attribute on the Symbol (e.g. symbol.wild = True) and if any special property was found, `symbol.special` is set True [212] [213]. It also calls `assign_paying_bool(config)` to mark `symbol.is_paying=True` and attach its paytable info if the name is in paytable [214]. So every symbol object knows if it's part of a pay combination and its payout values [215].
- The Board class uses these Symbol objects to populate `self.board` (a matrix of symbols). It also collects indices of special symbols in `self.special_symbols_on_board`, categorized by property [216] [217]. For example, after drawing, `self.special_symbols_on_board['scatter']` might be a list of coordinates where scatter symbols appear [216] [218]. This is used for `check_fs_condition()` (if length >= trigger count) [217], and also available for custom logic (e.g. a feature where you need to know positions of a special symbol).
- Board also stores `self.reelstrip_id` (which reelset was used, chosen via weighted random) [219]. If the chosen positions would exceed the strip length (like you want to show padding symbol beyond end of reel), it wraps around using modulo [220].

- Special effect: If `include_padding=True` in config, Board identifies the symbol just above row0 on each reel as `top_symbols` and just below bottom as `bottom_symbols` [221]. These are included in the `reveal` event under `paddingPositions` [222], so the front-end can show symbols partially off-screen (commonly the top of the reel above the first row).

- **Tumble (src.calculations.tumble):** Manages cascading mechanics. The `Tumble` class extends Board and provides `tumble_board()` which removes symbols marked with `explode=True` (these are symbols that were part of a win in cluster or scatter games) [223] [224]. It then causes above symbols to drop into those gaps:

- It scans each reel for exploded symbols count, then for each reel, it appends that many symbols from above (taking from either the reel's top buffer or generating new ones) [225] [226]. If padding symbols exist, the first vacancy is filled by the symbol from `top_symbols` (which simulates the next symbol falling in) [227]. Additional new symbols are generated from continuing up the reel strip beyond the visible window (the engine simulates an "infinite" upward reel by looping around or tracking an index) [220] [226].
- After filling, it can emit a `tumble_board_event` that logs which symbols dropped and from where [228] [229], as well as an updated `reveal_event` for the new board if needed (the SDK might reuse the reveal for initial spin only, and rely on front-end animating the drop).

- The Tumble process repeats: in cluster games, after tumbling, it will evaluate wins again on the new board. The sample scatter game loop in `run_spin` pseudocode illustrates this: while there were wins and not wincap triggered, it tumbles and evaluates again [230] [231]. Each cascade's win is emitted via an `update_tumble_win_event` and accumulated into `self.tumble_win` etc., until no more wins [231].

- **Win Calculations (Lines, Ways, Scatter, Cluster):** Each of these is implemented in `src.calculations`. They all follow the pattern: find groups of like symbols meeting criteria, check if that combination exists in paytable (taking wilds into account), and accumulate their payout.

- **LinesWins:** Iterates through each payline defined in `config.paylines` (which maps line index to row positions per reel) [232]. For each line, it scans from reel0 onward while symbols match; wilds count as matching any symbol but have special logic to decide final symbol:
  - The engine uses default wild symbol `"W"` and wild attribute `"wild"` unless configured otherwise [233]. If a wild combination is found (e.g. W-W-W-L4-L4), it compares the payout of treating them as wilds vs as the underlying symbol sequence [233] [234]. In the example earlier, three W's vs five L4's – it chose the higher payout combination [235] [236]. Essentially, if `(kind, "W")` is in paytable, it checks if that pays more than the actual symbol win and uses the better.
  - You can configure the wild pay behavior by paytable entries. A common approach (if you don't want partial wild wins overriding real wins) is to only give Wild a payout for the full line (e.g. only (5,"W") exists in paytable), so that wilds only pay by themselves if they fill entire line [237] [238]. Otherwise mixed wins count as the non-wild symbol.
  - The result is a list of all line wins, each with a `{'symbol': sym, 'kind': n, 'win': value, 'positions': [...], 'meta': {'lineIndex': i, ...} }` [173]. If multipliers exist on symbols or global, the engine uses `wins/multiplier_strategy.apply_mult()` to adjust the win:
  - It has strategies: `"symbol"` (multiply by sum of multipliers on involved symbols), `"global"` (multiply by global multiplier once), or `"combined"` (do both) [239]. By default, if you use symbol multipliers in ways or cluster, the engine *adds* them together for line/cluster wins, but for ways it *multiplies the count of ways* [169] [240]. This is handled such that games with multiplier symbols can just rely on apply_mult to correctly compute enhanced payout [239] [241].
- **WaysWins:** Finds, for each symbol type, if it appears on 3+ reels in a row anywhere. For each symbol (excluding scatters and usually excluding wild on reel1), count how many on reel1, reel2, reel3 until a reel has none. The total ways = product of counts per reel. For example, H1 on reels 1,2,3 with counts 1×2×1 yields 2 ways [242]. If a multiplier symbol is present, ways use a different approach: instead of summing multipliers, they multiply the ways count by the multiplier value [243] [242]. The doc example: if a 3× multiplier is on reel3 for H1, and it was 1×2×1 ways = 2 ways normally, they do (1*2*3) = 6 ways [244]. The returned win entries include `'ways': N` in meta.

- **Scatter/Cluster:** These count symbols anywhere or in clusters. They often have pay ranges, so you might see the engine having to translate, say, 12 symbols into the highest pay group via `convert_range_table`. The `overlay` meta in cluster is the "center of mass" position of the cluster for nice animation positioning [174] [245].

- **Config (src.config.config):** We covered most features. Additionally, Config defines the output directory names (in `output_filenames.py`). For instance, when `generate_configs(gamestate)` is called at end of run, it uses the templates in `OutputFiles` class to name files like `books_{mode}.jsonl` or `lookUpTable_{mode}.csv` and places them in appropriate `library` subfolders [246] [247]. It ensures directories exist and might compute a hash for each file for the backend to verify changes [246].

- Config also allows custom **win_levels**: e.g. `self.win_levels = {"standard": 1, "big": 20, "mega": 100}` indicating thresholds of bet multipliers for categorizing wins. The `set_win_event` uses these to mark win level key (like `'winLevel': 2` meaning maybe "big win") which the frontend can use for different animations [248]. This was indicated where `set_win_event(winlevel_key='standard')` is default [248], but bonus end might use `'endFeature'` as a win level for final big win screen.

- **Events (src.events.events):** This module defines a variety of event constructors that the engine calls to append events to the book [249] [248]. Each function typically takes the GameState (`gamestate`) and possibly some params, and does:

  ```
  event = {"index": n, "type": "...", ...fields...}
  gamestate.book["events"].append(copy.deepcopy(event))
  ```

  where `n` is the next event index (they likely increment an internal counter for each event) [250] [251]. They use deep copy to ensure if the same event object is reused it doesn't mutate past events [252]. The result is an ordered list of events from index 0 up that describe the whole round.

Notable events functions include: - `reveal_event(gamestate)`: Records the symbols revealed at the start of a spin. It typically stores `board` (matrix of symbol names), `paddingPositions` (top/bottom symbol names if padding), `gameType` ("basegame" or "freegame"), and `anticipation` (the delay array for scatters) [250] [251]. The frontend uses this to display the initial spin result. - `win_info_event(gamestate, include_padding_index=True)`: Creates an event summarizing wins after a spin or cascade. It sets `"totalWin": X` and `"wins": [ ... ]` with details of each win (as stored in `gamestate.win_data['wins']`) [248] [253]. If `include_padding_index`, it adjusts the row indices in win positions to account for padded top symbols (because if row indexing started at 1 due to padding as mentioned [254], the event needs to shift them back for frontend). Generally, leave this True so that the positions in events match the indices the frontend uses. - `set_win_event(gamestate, winLevel_key='standard')`: Announces the end of a spin's wins with a categorized level. It often contains `"amount": spin_win` and `"winLevel": <level>` [253]. For example, after a spin, you might get:

```
{ "index": 2, "type": "setWin", "amount": 10, "winLevel": 2 }
```

if 10x corresponds to level 2 ("big win") [179]. - `set_total_event(gamestate)`: If multiple spins (like free spins), this might set the running total. Or in a base game, it might be the same as setWin. - `update_freespin_event(gamestate)`: Emitted at start of each free spin to update the spin count. It

might contain `"current": self.fs, "total": self.tot_fs` to show “Free Spin 3 of 8” on UI [255]. - `freespin_end_event(gamestate, winLevel_key='endFeature')`: Emitted at end of free spins feature. Could contain the total feature win and a special winLevel for feature end. Frontend can use this to show a summary screen (e.g. “Bonus Won 200x!”). - `final_win_event(gamestate)`: Marks the final total win of the round (especially if feature was involved). Often the same amount as `set_total_event` but a distinct event type to signal completion to UI [192]. - `update_global_mult_event(gamestate)`: If the game has a global multiplier that changes (some games have increasing multipliers each cascade), this event would log the new multiplier so UI can update it [256]. - `tumble_board_event(gamestate)`: As mentioned, logs which symbols fell in a tumble cascade [228]. Frontend might not always explicitly use this, but it’s there if needed for complex cascades.

The **Usage Notes** for events emphasize: each function appends to `book['events']`, using deep copies so that once an event is added, subsequent changes don’t affect it [252]. This ensures the event log is immutable after recording. The events system is crucial for a “transparent, trackable game state” – anyone can read the events and understand exactly how the outcome came about [257].

- **Executables (src.executables.executables):** The Executables class groups many of the high-level game actions we discussed in `run_spin`:
- `draw_board(emit_event=True)`: calls Board.create_board and optionally emits the reveal_event [164].
- `force_special_board(criteria, num_force_syms)`: finds positions for forcing certain special symbols on the board (searches reelstrips for that symbol) [164] [258]. Could be used to guarantee e.g. a certain number of Wilds for a feature.
- `emit_wayswin_events()`, `emit_linewin_events()`, `emit_tumble_win_events()`: convenience methods to emit the appropriate winInfo and setWin events after evaluating wins in those games [259].
- `tumble_game_board()`: wraps the Tumble.tumble_board and likely calls `emit_tumble_win_events` for cascades [260]. It may loop until no more cascades.
- `evaluate_wincap()`: checks if the current cumulative win >= wincap and if so sets a flag (so the game can stop giving free spins, etc.). Possibly triggers `wincap_event` to log that max win was hit.
- `count_special_symbols(attr)`: returns how many symbols with a given special attribute are on the board (uses `special_symbols_on_board` which Board prepared) [261].
- `check_fs_condition(scatter_key="scatter")`: returns True if enough scatters present to trigger feature (compares `len(self.special_symbols_on_board[scatter_key])` to trigger count) [262].
- `check_freespin_entry(scatter_key="scatter")`: likely similar to check_fs_condition, or it might involve buy-in logic.
- `run_freespin_from_base()` / `update_freespin_amount()` / `update_fs_retrigger_amt()` / `update_freespin()` / `end_freespin()`: manage the free spin flow as described earlier [263]. The base implementations handle the common case (logging triggers, setting tot_fs from freespin_triggers, etc.), but games can override them (as we saw with doubling free spins in scatter game) [81] [264].
- `evaluate_finalwin()`: adds up base spin and feature wins for the final payout and might emit finalWin event.
- `update_global_mult()`: updates a global multiplier state (for games with progressive multiplier).

Dependencies and Usage: The Executables functions call upon Events functions to emit events, and call Calculations functions to evaluate wins. They form the bridge between the math logic and event logging.

- **State (src.state.state & src.state.gamestate):** The `GeneralGameState` or base `GameState` class provides the overall framework:
- **Properties:** It instantiates a `WinManager` for the simulation batch [265]. It holds `self.book` (current sim events), `self.library` (list of all books) implicitly via how outputs are written, `self.criteria` (the criteria name for current simulation), and others like `self.sim` (simulation index), `self.gametype` (current mode).
- **Important Methods:**
  - `create_symbol_map()`: Possibly builds a mapping of symbol names to Symbol instances or indices for quick lookup.
  - `assign_special_sym_function()`: Abstract method meant to be overridden as discussed, so base may do nothing.
  - `reset_book()`: Initializes `book` dict as described and sets `self.repeat=False` or True appropriately [266]. Base version sets repeat False, but many overrides set it True after calling base (because they plan to loop until acceptance) [79].
  - `reset_seed(sim)`: Seeds random using a combination of sim ID and maybe a global seed to ensure independence across threads but determinism in sequence [160].
  - `reset_fs_spin()`: Prepares for entering free spin mode. Base likely sets `self.fs = 0` and flags like `self.in_feature = True`.
  - `get_betmode(name)` and `get_current_betmode()`: Retrieve the BetMode object by name or current one (if simulation criteria tied to a mode).
  - `get_current_distribution_conditions()`: Returns the dict of conditions for the distribution of the current simulation [267] (this is how `draw_board` knows what reel_weights to use and `assign_mult_property` knows what `mult_values` to draw from [144]).
  - `get_wincap_triggered()`: Returns if the win cap was hit in the current round (so external functions can check).
  - `in_criteria(*args)`: Possibly checks if current criteria name matches any of provided (for convenience in code).
  - `record(description: dict)`: This implements the event force recording: it pushes the given description (a small dict like `{"gametype":"basegame","symbol":"scatter","kind":5}`) and the current book ID into a temp list for forces [268] [182]. The description acts as a key in force JSON output with list of bookIds that had that event [182] [269]. (The actual writing to file is handled when simulation ends, see `imprint_wins()`).
  - `check_force_keys(description)`: Likely ensures the description dict has all expected keys for consistency in force file.
  - `combine(modes, betmode_name)`: Possibly a helper to merge results from multiple sub-modes (not commonly used unless a game had a sub-mode segmentation beyond what we've seen).
  - `imprint_wins()`: Called at end of accepted simulation – it finalizes the book:
  - Appends the `book` to a list (or writes it to file stream if streaming) [270].

    - Writes any accumulated `self.temp_wins` (force events) into the force record for this mode, but crucially it only writes now because if a simulation was repeated, we don't want to log the failed attempts [270] [271].
    - Resets `temp_wins` after imprinting.
    - Calls `self.win_manager.update_end_round_wins()` to add this spin's wins to cumulative totals and perhaps print debug output [193].
    - `update_final_win()`: Could set some final tally or move `wallet_manager.running_bet_win` into a final variable.
    - `check_repeat()`: Implements the acceptance criteria check. It probably looks at the predetermined criteria for this spin (GameState likely has something like `sim_to_criteria` map prepared at start) and checks:
    - If criteria is "0" and `wallet_manager.running_bet_win > 0`, then still had a win, so repeat [114].
    - If criteria is "freegame" and maybe `not self.freegame_triggered`, then repeat [114].
    - If criteria is "wincap" and `not self.wincap_triggered`, repeat [114].
    - Else set `self.repeat=False`.
    - Note: The SDK warns that having too strict a criteria can cause very long repeats (e.g., if you demanded a natural max win without forcing it, you could loop indefinitely) [118].
    - `run_spin(self, sim)` and `run_freespin(self)`: Abstracts meant to be overridden by game if needed, but typically the base might have a default as described earlier. In sample games, these are overridden (in `game_calculations.py` or `game_override.py`) to implement the specific loop logic, especially for cascading games where `while win -> tumble` needs to be inserted (as shown in scatter sample snippet) [230].
    - `run_sims(...)`: Manages dividing simulations across threads and running them. Likely uses Python threading or multiprocessing to start `num_threads` workers each running a portion of the simulations. Each thread creates its own GameState or copies (the SDK documentation suggests passing a `betmode_copy_list` for threads) [194]. It collects all results at the end.

- **WinManager (src.wins.wallet_manager):** (It's called Wallet Manager in docs) This class is constructed at the start of simulations with the identifiers for base and free modes [265]. It tracks:

- `spin_win` – the current spin's win (resets every spin or free spin) [272].
- `running_bet_win` – cumulative win in the current simulation (spin + any free spins) [273].
- `basegame_wins` and `freegame_wins` – accumulators that reset each simulation: as you call `update_gametype_wins(gametype)`, it will add the current spin's outcome to either base or free accumulator and then possibly reset the `spin_win` for next spin [274].
- `total_cumulative_wins`, `cumulative_base_wins`, `cumulative_free_wins` – these accumulate over the entire simulation batch (all spins) to compute overall RTP by mode and in total [275]. Every time a round ends (`update_end_round_wins()`), it adds that round's base and free totals to these [275].
- Functions:
    - `update_spinwin(amount)`: Adds a win amount to `spin_win` (used when multiple wins in one spin, e.g. line wins from multiple lines, or cascade wins) [176].
    - `set_spinwin(amount)`: Sets `running_bet_win = amount` and also updates `running_bet_win` (I suspect `set_spinwin` is called after free spins to directly set final total).

- `update_gametype_wins(game_type)`: If `game_type` is base, it adds current `spin_win` to `basegame_wins`, if free, to `freegame_wins`, then resets `spin_win` to 0 for next spin context [276] [277]. The note in docs: "Important! as part of final payout verification, self.final_win and sum(basegame_wins + freegame_wins) must match, else runtime error" [277] – this is a safety check to ensure no win is lost between base and free or double-counted. Essentially, at end of simulation, `final_win` should equal `basegame_wins + freegame_wins`. The engine probably checks this and throws if they diverge.
- `update_end_round_wins()`: Called after each simulation, it takes the completed round's basegame_wins and freegame_wins and adds them to the grand totals `cumulative_base_wins` and `cumulative_free_wins`, and also adds their sum to `total_cumulative_wins` [275]. It then might print the thread's RTP info as we saw (e.g., "Thread 0 ... RTP [baseGame: X, freeGame: Y]") [278].
- Possibly `reset` functions to clear spin and running values at appropriate times.

The WinManager basically segregates wins by category and ensures final consistency. By having separate base vs free tallies, we can compute RTP contributions of base vs bonus (the RGS also logs these from the CSV segmented file) [91].

Now, stepping back: As a developer, do you need to modify these core modules? Usually *no* – you can rely on configuration and the game-level override classes to adjust behavior. But you should understand their responsibilities in case you observe something in output and need to pinpoint where it came from. For example, if an event isn't appearing as expected, you'd check the Events module function corresponding to it. Or if a particular win combination isn't recognized, check the Lines or Ways logic and ensure your config was set properly (like wild definitions or paytable entries).

Example Games and Templates

The SDK's **Example Games** serve as both tutorials and starting templates. Each demonstrates a specific common slot mechanic:

- **Lines Game (0_0_lines):** A classic slot with paylines and a free-spin feature. The documentation likely describes its basegame (paying on lines) and its freegame triggered by scatters [279]. The basegame uses `win_type="lines"` and defines 20 paylines in config. The freegame might have a different reelset or multiplier. You can see how the game override handles scatter triggers to call free spins, and how events like anticipation are used.
- **Ways Game (0_0_ways):** Pays on adjacent symbols regardless of line position (243 ways on a 3x5 reelset). It presumably triggers free spins similarly. This example shows how to configure a ways game (simply set win_type "ways" and provide reelstrips; no paylines needed, and maybe set a multiplier strategy in config if needed).
- **Cluster Game (0_0_cluster):** Wins occur for clusters of 5+ touching symbols. Typically these games cascade. The sample likely has basegame clusters and possibly a separate cluster-based freegame with increasing multipliers (common in popular games). The example's notes likely mention how the `convert_range_table` was used to define pay groups (like 5,6-7,8-10,11+ clusters) [280] [281]. It will also illustrate implementing cascades: e.g. in `game_calculations.py` of cluster game, `run_spin` might loop while wins exist (similar to scatter).

- **Scatter-Pays Game (0_0_scatter):** Similar to cluster but positions don't matter – any 8+ same symbols anywhere pay (like "pay anywhere" games). Often these also have cascades and a free spin with multipliers. The example likely covers that and has detailed event flows for tumbles.
- **Expanding Wilds + Superspin (0_0_expwilds):** A more complex example combining features: a lines game where certain symbols expand to wild reels and collected prizes trigger a "Super Spin" after normal free spins. This shows how to implement more complex state – e.g., maybe they use `Context` or extra events to handle the superspin mode. It demonstrates adding a **third mode** (superspin) beyond base and free, how to set `is_feature` or such for it, and custom events to handle transitions (like an event to start superspin mode).

Each example game section in docs breaks down: - **Basegame**: rules of the base mode (lines or ways, special symbols like wild or scatter). - **Freegame rules**: what triggers it, any differences (e.g., free spins might have multipliers or a different reelset) [282] . - **Superspin or Bonus**: if applicable, explanation. - **Notes**: particular implementation details, perhaps how events map or edge cases (like if free spins win cap). - **Event descriptions**: possibly a walk-through of the event sequence for a sample round of that game [283] .

Using Examples: When starting a new game, find the example closest to your concept and use it as a base. For instance, if you want a tumble mechanic, start from the scatter or cluster example so you have the repeat-and-tumble loop logic. If you want a buyable bonus, see how the expanding wilds game set up a superspin mode with `is_buybonus`. The examples also illustrate how to use the SDK's features properly (like recording events for force files using `self.record`, or issuing custom events via `game_events.py` if needed).

Optimization Algorithm – Balancing the RTP

After generating raw outcomes, the **Optimization Algorithm** comes into play to tune the distribution of outcomes to the desired RTP (and possibly hit rate constraints). Stake's SDK includes a Rust-based iterative optimizer which treats the lookup table as a search space for weight adjustments [284] [285] .

The optimizer is essentially a **genetic algorithm / iterative sampler** that: - Takes the initial lookup table (with all weights = 1) and target RTP (the one you set in config). - It will adjust weights of outcomes up or down to try to reach exactly that RTP for each mode, while also possibly respecting any "Conditions" (like ensure 0-win frequency is X%, etc.) [286] . - It runs multiple generations or iterations trying combinations (this is why in the sample output thread logs, you see some threads reporting an RTP higher than expected – the optimizer will heavily down-weight those high outcomes in final table) [36] . - The result is new weight values (often fractional probabilities turned into big integers).

To use it, you have to provide **optimization parameters** in the `run.py` or config: - The SDK references an `OptimizationSetup` class for game-specific parameters [287] . This likely includes: - **Conditions:** e.g., ensure basegame portion of RTP is 40% and freegame 60%, or ensure a certain hit rate. These conditions can be fed so the optimizer tries to satisfy them besides just RTP [288] . - **Scaling:** possibly how aggressively to adjust or if to use logarithmic scales. - **Parameters:** number of iterations, population size, mutation rate, etc. The doc snippet suggests "number of distributions to trial before combination, minimum …" but got cut [289] .

These settings are written to a `setup.txt` for the Rust binary to consume [290]. The math SDK will generate `config_math.json` with game parameters and `setup.txt` with optimization parameters, then call the optimization binary with those as input [291].

- **Executing the optimizer:** The `run.py` sets `run_conditions["run_optimization"] = True` to invoke this after simulations. It will spawn the Rust binary (`optimize.exe` or so) with the newly created `math_config.json` and `setup.txt`. The binary then outputs optimized lookup tables (with "_0" suffix) [96], and possibly an updated `math_config.json` with results.

The documentation even provides a research paper about the algorithm [284]. We won't dive into GA theory, but key points: - It's iterative weighted sampling – likely adjusting outcome frequencies in proportion to error between current RTP and target. It might also ensure that extremely large wins keep a minimum frequency to preserve max win probability (hence requiring wincap spins in simulation). - **Rust Implementation:** Because it's computationally heavy (a large table could have millions of rows), it's done in Rust for speed. The first run or code change triggers a compile of the Rust code to a binary, which then runs quickly on subsequent uses [285]. - The output is reliable if your simulation had enough data. If your simulation was too small (e.g. you only simulated 1000 spins but your quotas demanded something rare), you might get a weird distribution that the optimizer then heavily skews. Aim for large sim counts and realistic quotas so the optimizer just fine-tunes rather than doing extreme adjustments.

After optimization, a new `lookUpTable_mode_0.csv` is produced with weights that yield the target RTP exactly [292]. The SDK also outputs a `lookUpTableIdToCriteria_mode.csv` (maps each sim ID to which criteria it belonged) and `lookUpTableSegmented_mode.csv` (maps each sim ID to how much was base vs free) [293]. These extra files help in analysis (e.g., you can see what portion of RTP comes from free games by summing freeGameWins of all entries, etc.), but you typically do **not** upload them – only the main CSV with weights is uploaded [294] [295].

Frontend SDK – Overview and Development Workflow

While the math SDK creates the "brain" of the game, the **frontend SDK** builds the "body" – the visual and interactive part players see. Stake's frontend framework is built on **PixiJS** (a 2D WebGL rendering library) and **Svelte** (a reactive UI framework), combined through a custom `pixi-svelte` integration [296] [297]. It also leverages **SvelteKit** for development server and build tooling, and **Turborepo** to manage a monorepo structure (likely splitting core components, game-specific components, etc.) [296] [298].

Dependencies and Stack

Key frontend dependencies include [296]: - **PixiJS:** High-performance rendering of 2D graphics (used for spinning reels, animations, etc.) [299]. - **Svelte:** A modern UI framework that compiles reactive components (used for game UI layout, state management) [300]. - **pixi-svelte (in-house):** A custom library that allows Svelte components to easily use PixiJS elements declaratively [297]. This likely provides Svelte components for Pixi containers, sprites, text, etc., bridging the gap between Svelte reactivity and Pixi's imperative rendering. - **SvelteKit:** Framework for building Svelte applications, provides dev server, routing, and building to a static site (the game ultimately is a static bundle served via CDN) [301]. - **Turborepo:** Manages multiple packages in the repository. The front-end might be organized as a monorepo with packages like `core`, `components`, `games/lines`, `games/shared`, etc., and Turborepo helps run builds/tests across them. - **Storybook:** A development environment for UI components. They use it to run the game in isolation and test various

states (spin, pre-spin, etc.) [302] [303] . - Other supporting libraries might include GSAP (for animations), Howler (for sound), etc., though not explicitly listed in the snippet.

As a developer, familiarity with HTML/CSS/JS and specifically Svelte and Pixi is important. If new to Svelte or Pixi, the user is advised to understand those (the docs link to official resources) [304] .

Setting Up the Frontend Environment

To get started with the frontend SDK sample games: - Use **VS Code** as recommended (with likely Svelte extensions) [305] . - Install Node.js v18 (the project specifies Node 18.18.0) [306] . Using NVM is suggested for version management [307] . Once Node is set up (`node -v` shows 18.18.0) [308] , you can proceed. - Run `npm install` or the Turborepo equivalent (`npm run install` which runs through all packages). - The tutorial likely says to start the Storybook:

```
npm run storybook
```

This should launch Storybook at a local URL where you can see a list of game components and game states.

The **Getting Started tutorial** walks through these steps in detail, including NVM installation commands (as shown) and perhaps how to navigate Storybook once it's up [307] .

Once Storybook is running, you can select a game and scenario from its UI: - e.g., "COMPONENTS/Lines Game/component" to see the entire game component without loading screen [302] . - "preSpin" to simulate state before spin (bets etc.) [303] . - "emitterEvent: boardHide" to test an event that hides the board (maybe after spinning) [309] . - etc. They have stories for Symbol component too (to test rendering each symbol) [310] .

This is extremely useful: you can feed the Storybook particular event data (like a specific `board` outcome) and verify the UI displays correctly. It's essentially a sandbox for your game's UI.

Frontend Architecture and Flow

The frontend is designed to consume the math SDK outputs (the JSON events from `/play`) and render the game accordingly. **Flowchart** documentation illustrates the lifecycle when a round is played:

1. **Game Launch:** The game loads via SvelteKit, reading query params (sessionID, rgs_url, etc.) [311] . It likely establishes a connection or uses fetch to the RGS.
2. **Player Input:** When player presses spin (or auto-spin triggers), the frontend calls the RGS `/play` endpoint with the selected bet mode (base or bonus) and session, etc., to request an outcome.
3. **RGS Response:** The RGS responds with a JSON containing the events for that round (basically one of the entries from books file). The frontend receives this.
4. **Processing Events:** The SDK uses a function like `playBookEvents(events)` to iterate through the list [312] . This function (from `createPlayBookUtils.ts`) likely goes event by event (index 0 upwards) and for each:
5. It triggers a corresponding **bookEvent** handler from a `bookEventHandlerMap` [313] [314] . For example, for `"type":"reveal"`, the handler might set up the reels to display those symbols. For

`"winInfo"`, handler might update the UI win meter. These handlers are defined in code, mapping event types to Svelte component methods or Pixi actions.

6. Some events (like `reveal`) might be visual and synchronous (just show symbols), others (like `winInfo` or `finalWin`) might require waiting for an animation to finish.
7. The system might use an **eventEmitter** (perhaps an instance of Node's EventEmitter or a Svelte store) to emit **emitter events** when certain book events are done [313] [315]. For instance, after finishing processing all reveal events, it could emit an `"spinComplete"` event internally.
8. The **emitterEventHandlerMap** then maps those internal events to UI updates. For example, an internal `"boardHide"` event might tell the Symbol components to fade out (this was referenced in storybook as a scenario) [309].

The flowchart doc lists `bookEvent`, `bookEventHandlerMap`, `eventEmitter`, `emitterEvent`, `emitterEventHandlerMap` as key pieces [313]. We interpret: - *bookEvent:* a single event from the RGS outcome. - *bookEventHandlerMap:* a dictionary where keys are event types ("reveal", "winInfo", etc.) and values are functions to handle them [313]. - *eventEmitter:* a system to broadcast when certain states happen (could be the Svelte component context or a library). - *emitterEvent:* an internal UI event triggered by code (like "symbolDropComplete", "bigWinAnimationEnd"). - *emitterEventHandlerMap:* mapping of those internal events to any follow-up actions (like if "endFeature" event triggers showing a summary, etc.).

The **simplified flowchart** presumably shows that after receiving the play result, the game calls `playBookEvents()`, which: - Loops through each bookEvent and uses `bookEventHandlerMap` to handle it in sequence [316]. - Handlers might push tasks to Pixi (like start reel spin animation with target stops). - Some handlers might register listeners for when animation finishes, at which point they emit an emitterEvent into `eventEmitter`. - Those emitterEvents then trigger their handlers (via emitterEventHandlerMap) which might immediately fire the next bookEvent or chain additional animations.

Essentially, the front-end is **event-driven**, just like the math. The math produced a timeline of events; the front-end maps that timeline to a timeline of visuals and sounds.

For example: - **Reveal Event Handling:** On "reveal" bookEvent: - The handler sets each reel's symbols to the ones in `board` and starts a spinning animation that lands on those symbols. - It might set up an anticipation delay on reels 3,4 if anticipation array is [0,0,1,2,3] (so reels 2,3,4 spin slower). - When all reels stop, it emits an emitterEvent "reelsStopped". - **Win Event Handling:** On "winInfo": - The handler highlights winning symbols (maybe flashing paylines or clusters). - It updates a running win counter on screen to show the amount. - After a short delay or on player click, it emits "winDisplayed". - **Feature Trigger Event Handling:** On a "fs_trigger_event": - Handler might show a "Free Spins Won!" banner. - Possibly also immediately calls `playBookEvents()` recursively for the free spin events that follow, or more likely, the free spins are already part of the events list after trigger event, so continue processing. - Or, maybe they treat free spins as a nested sequence: emitterEvent "startFreeSpins" triggers switching UI mode to free spins (different background, etc.). - **End Feature Event:** On "freespin_end_event": - Handler stops any free spin counter UI and shows a summary of total free spins win (this might correspond to a winLevel 'endFeature' event as well). - It could emit "featureEnd" event when done, which is handled to transition back to base game mode visuals. - **FinalWin Event:** On "finalWin": - Possibly triggers any final coin animation or just signals that round is over, allowing player to spin again (emitterEvent "roundEnd" enabling the Spin button).

The **Task Breakdown** likely enumerates tasks to implement when developing a new game's front-end: - Create art assets (symbol images, background, etc.). - Define Svelte components for Reel, Symbol, etc., or reuse core ones. - Configure payline drawing or cluster highlighting logic. - Implement any custom animations (e.g., expanding wild). - Hook up all math events to visual outcomes: - E.g., if you add a new math event type "explodeSymbolEvent", you must add a handler in bookEventHandlerMap to handle it (like play an explosion animation). - The "Adding New Events" section [317] likely instructs how to extend the front-end event handling when you introduce custom events in math (for example, the expanding wilds game might have a custom event for triggering the super spin).

Adding New Events: If your math emits a custom `events` type not in base set, you need to: - Add a handler in the front-end `bookEventHandlerMap`. - Possibly add a corresponding emitter event if needed (for intermediate state). - Create a Storybook story to test it.

For instance, if you have a "bonusPick" event where player picks a prize, the front-end must pause and wait for player input. The handler for "bonusPick" could: - Display a pick UI (several face-down cards). - Then *wait* for player to click one. On click, send selection to RGS (if RGS required, or if outcome already decided, just reveal). - Emit "bonusPickComplete" event internally after selection, which the emitterEventHandlerMap uses to resume processing remaining events.

File Structure: The front-end likely has: - `packages/` containing reusable libraries (like `pixi-svelte`, `utils-book` which has playBookEvents, etc.). - `apps/` containing each sample game as a SvelteKit app. - Shared UI components in a core library (buttons, meters). - Game-specific components in their own folder (for theming). - The `File Structure` doc will outline how they organize code and where to put new game code.

Context and UI Modules: - The *Context* might refer to Svelte's context API usage for passing down game state or Pixi application instance to components [317]. - The *UI* section probably describes the main Svelte structure: e.g., a `Game.svelte` component that uses `<PixiCanvas>` and places symbols, and how reactive values (like current balance, current bet, win count) are handled (possibly via Svelte stores).

Running on Stake Platform

When the front-end is built (likely `npm run build`), it produces a static bundle (index.html, assets, JS) to be served via CDN. The game will be loaded by Stake using a URL format (as given in RGS details):

```
https://TEAM.cdn.stake-engine.com/GAMEID/GAMEVERSION/index.html?
sessionID=...&lang=...&device=...&rgs_url=...
```

The front-end uses these query params to communicate back with the RGS (through `rgs_url`). The **RGS Technical Details** specify endpoints like: - `POST /sessions` to validate sessionID perhaps, - `POST /bets` or `/play` to start a round, - `POST /endround` to close a round if needed, - Perhaps `/balance` to fetch updated balance [318] [319].

The URL structure shows `GameID` and `GameVersion`, meaning multiple versions can be hosted (for updates). The provider (you) would have a TeamName (like a subdomain).

The RGS doc goes into how to integrate: - The query params and endpoints to call. - Likely an example of a typical round trip request/response JSON.

For game developers, ensure your front-end calls the correct RGS endpoints: - Use the provided `rgs_url` parameter for API calls (this will point to Stake's RGS cluster). - Include `sessionID` for authentication in those calls. - Use HTTPS and proper request format (the doc likely gives a JSON schema for requests and responses).

Conclusion

By following this guide: - **Math Developers** can configure every aspect of the slot's mechanics in Python, simulate to verify math performance, and output the required static files [197] [203]. They can utilize built-in win calculators and override as needed for custom features, and ensure all outcomes are balanced and verified. - **Frontend Developers** can use the SDK's scaffolding to render outcomes frame-by-frame exactly as described by the math events, giving players an engaging visual experience that matches the logic. The use of modern web frameworks (Svelte) combined with PixiJS provides both flexibility and performance, and Storybook integration allows rapid development and testing of game states in isolation [302] [303]. - **Integration Engineers** have clear specifications for packaging and uploading the game. By producing the correct files (index.json, etc.) and implementing the RGS API calls properly, the game can be deployed on the Stake Engine platform [197] [211].

In summary, the Stake Engine SDK provides a **complete pipeline** for slot game development: from defining reels and probabilities, through simulating millions of rounds and optimizing payout distribution, to rendering the game client with rich animations – all in a structured, transparent, and testable way. By adhering to the SDK's formats and using the provided tools, developers (and even AI assistants guiding them) can efficiently create complex, high-quality slot games ready to be launched on Stake. Happy building and good luck for the big wins!

Sources:

- Stake Math SDK Documentation [2] [92] [178] [146] [140] (engine and file format specifics)
- Stake Frontend SDK Documentation [296] [302] [316] (frontend architecture and usage)
- Stake RGS API Documentation [203] [311] (integration requirements for deployment)

---

[1] [2] [3] [4] [5] [6] [207] [317] Stake Development Kits
https://stakeengine.github.io/math-sdk/

[7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] Engine Setup - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/general_overview/

[18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [178] [179] [191] [192] [222] [278] Quickstart Guide - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/quickstart/

[41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] SDK Directory - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/directory/

62 63 64 76 77 210 Game Structure - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/overview_section/game_struct/

65 66 67 68 69 70 71 72 73 74 75 78 79 80 81 82 83 84 85 86 87 88 89 90 91 94 95 96 97 183 264 266 State Machine - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/overview_section/state_overview/

92 93 195 196 197 198 199 200 201 202 203 204 205 206 208 209 Required Math File Format - Stake Development Kits
https://stakeengine.github.io/math-sdk/rgs_docs/data_format/

98 99 100 112 113 114 118 119 122 123 Simulation Acceptance - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/repeat_info/

101 102 103 104 120 121 Distribution - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/configuration_section/betmode_dist/

105 106 107 108 109 110 111 146 147 148 149 150 BetMode - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/configuration_section/betmode_overview/

115 116 117 125 126 127 128 151 152 153 156 157 158 159 160 180 181 184 185 186 187 188 189 190 Game Format - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/overview_section/game_format/

124 154 155 Config - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/config_info/

129 130 131 132 133 134 135 136 137 138 Configs - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/configuration_section/config_overview/

139 140 161 162 163 165 219 220 267 Board - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/board_info/

141 142 143 144 145 212 213 214 215 Symbols - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/syms_board_section/symbol_info/

164 177 258 259 260 261 262 263 Executables - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/executables_info/

166 167 232 233 234 235 236 237 238 Lines - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/lines_info/

168 169 175 240 242 243 244 Ways - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/ways_info/

170 171 172 173 174 176 193 239 241 245 272 273 274 275 276 277 Wins - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/win_info/

182 268 269 270 271 Force Files - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/force_info/

194 State - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/state_info/

211 246 247 292 293 294 295 Outputs - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/file_info/

216 217 218 221 254 Board - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/syms_board_section/board_info/

223 224 225 226 227 Tumble - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/tumble_info/

228 229 248 249 252 253 256 257 Events - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/event_info/

230 231 Scatter - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/scatter_info/

250 251 255 Events - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/gamestate_section/events_info/

265 Win Manager - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/win_manager/

279 282 283 Example Games - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/sample_section/sample_games/

280 281 Cluster - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/source_section/cluster_info/

284 285 286 287 288 289 290 291 Optimization Algorithm - Stake Development Kits
https://stakeengine.github.io/math-sdk/math_docs/optimization_section/optimization_algorithm/

296 297 298 299 300 301 304 Dependencies - Stake Development Kits
https://stakeengine.github.io/math-sdk/fe_docs/dependencies/

302 303 309 310 Storybook - Stake Development Kits
https://stakeengine.github.io/math-sdk/fe_docs/explore_sb/

305 306 307 308 Getting Started - Stake Development Kits
https://stakeengine.github.io/math-sdk/fe_docs/get_started/

311 318 319 RGS Technical Details - Stake Development Kits
https://stakeengine.github.io/math-sdk/rgs_docs/RGS/

312 313 314 315 316 Flowchart - Stake Development Kits
https://stakeengine.github.io/math-sdk/fe_docs/flowchart/