API Documentation

The Stake Development Kit is a comprehensive framework designed to simplify the creation, simulation, and optimization of slot games. Whether you're an independent developer or part of a dedicated studio, the SDK empowers you to bring your gaming vision to life with precision and efficiency. By leveraging the Carrot Remote Gaming Server (RGS), developers can seamlessly integrate their games on Stake.com, facilitating smooth and scalable deployments.

What Does the SDK Offer?

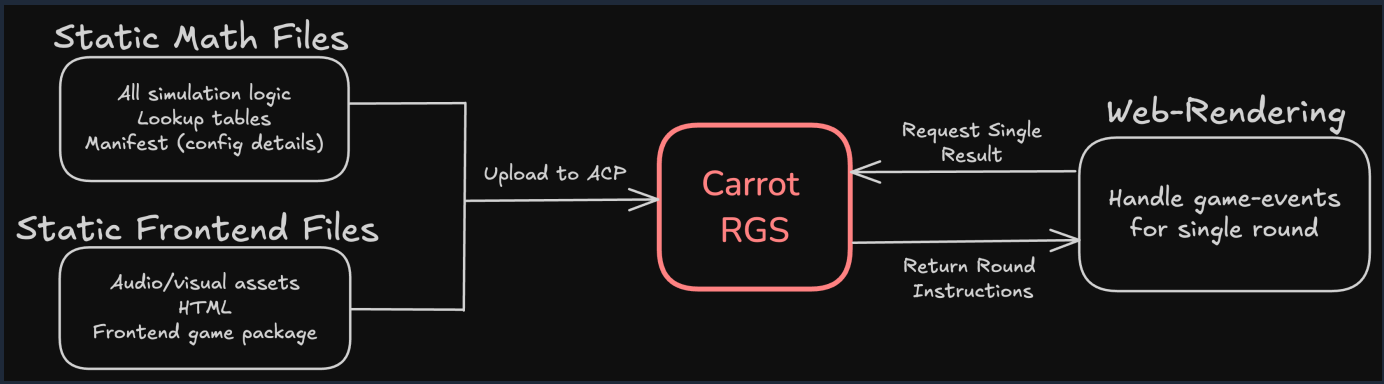
The SDK is an optional software package handling both the client-side rendering of games in-browser, and the generation of static files containing all possible game results.

1. **Math Framework**: A Python-based engine for defining game rules, simulating outcomes, and optimizing win distributions. It generates all necessary backend and configuration files, lookup tables, and simulation results.
2. **Frontend Framework**: A PixieJS/Svelte-based toolkit for creating visually engaging slot games. This component integrates seamlessly with the math engine's outputs, ensuring consistency between game logic and player experience.

[illegible]

[illegible] of static files. Developers utilizing their own frontend and/or math solutions are welcome to upload compatible file-formats to the Admin Control Panel (ACP). All possible game-outcomes must be contained within compressed game-files, typically separated out by modes. Each outcome must be mapped to a corresponding CSV file summarizing a single game-round by a simulation number, probability of selection, and final payout multiplier. When a betting round is

initiated a simulation number is selected at a frequency proportional to the simulation weighting, and the corresponding game events are returned though the `/play` API response.



General Requirements

When a game is submitted for approval, Stake Engine technical support will review its suitability for publication on Stake's platform. Approval requests will be actioned for a specific frontend and math version. Our team will inspect the game for functionality, clarity, communication, and technical performance. These factors determine the suitability of your game for publication. Stake Engine will communicate any issues or concerns, providing requested changes and reasoning for approval or rejection on a case-by-case basis.

All submissions must include a publicly accessible Google Drive or Dropbox link containing high resolution game assets (characters, symbols, backgrounds, etc..) with your approval request. These will be used to create a game tile conforming to Stake's design requirements.

Approval requests must be accompanied by a short blurb describing your game theme and mechanics for use in promotional material and the game description tag.

Key Restrictions

- Stake Engine games are strictly *stateless*: Each bet must be independent of previous outcomes. Games cannot include jackpots, gamble features, continuation, or early cashout options.
- Team names, game titles, and assets must comply with intellectual property/copyright law. Infringement is grounds for rejection.
- Games must be original designs. Pre-purchased or licensed games existing on other third-party websites will not be permitted.
- Game assets cannot include material with Stake™ branding or themes.

- Approval is at the discretion of the reviewer. Games deemed offensive, explicit, in poor taste, or of insufficient quality may be rejected.
- Games will be automatically considered for publication on stake.us under the condition that they abide by ***strict language requirements*** (see *Jurisdiction Requirements* below). Stake Engine offers a *social mode* setting within the play modal to test social languages.

Post Release Notes

Ensure that when submitting a review request, **the game is finalized and ready for publication**.

Once a game has been approved for publication on Stake/Stake-US, only minor updates to address visual issues are permitted, unless otherwise requested by the Stake Engine team. Changes to the underlying math model, the addition of new game modes, or modifications to gameplay mechanics will not be allowed.

Frontend and Communication

Frontend checks will include reviewing in-game performance and display to ensure the game is free of visual bugs, has the necessary industry-standard User Interface (UI) components, and behaves as described in the game rules.

Game Communication

Game Display

- Submitted games must use unique audio and visual assets. Assest such as backgrounds, symbols and/or animations provided with the *web-sdk* sample games will not be approved for publication.
- Ensure the game is free of visual bugs, including broken or missing assets or animations.
- Popout view support: Stake offers players to option to use the ‘mini-player’ modal to play games in the background. Games must support this small view without the active game board been visibly distorted.
- The game must support mobile view for commonly used devices, with all UI functionality remaining usable during screen scaling.
- All images and fonts must be loaded from the Stake Engine Content Delivery Network (CDN).

Rules and Paytable

- Game information must be accessible from the UI, including a detailed description of all game rules.

- If multiple game modes are available, provide a description of the cost of each bet and the actions being purchased.
- The RTP of the game (and each mode, if applicable) must be clearly communicated to the player.
- The maximum win amount for each mode must be clearly displayed.
- Payout amounts for all symbol combinations must be presented.
- If the game includes special symbols (e.g., cash prizes or multipliers), list all obtainable values.
- For feature modes (e.g., triggered by Scatter symbols), describe how to access them. Example: “3 Scatters award 10 free spins; 4 Scatters award 15 spins …”

UI Components

- Game must include a User Interface guide, briefly describing the functionality of UI buttons.
- The game must allow players to change the bet size.
- Player must be able to use all bet-levels returned within RGS auth/ response.
- The player’s current balance must be displayed.
- Final win amounts must be clearly shown for non-zero payout results.
- If an outcome contains multiple winning actions, the payout amount must incrementally update to match the final payout multiplier.
- The UI must include an option to disable sounds.
- The spacebar must be mapped to the *bet button*.
- If an ‘autoplay’ feature is present, the player must confirm the autoplay action, games are not allowed to automatically place consecutive bets with one click.

Other Checks

- The game must include the Stake Engine animation loader.
- Check the network tab to ensure no errors or game information is being logged.
- Playtest the game to verify it behaves as described in the rules (e.g., validating payout combinations).
- Game will be tested with various combinations of currencies and languages.
- If the game has a ‘fastplay’ option: wins amounts, winning symbol combinations and pop-up information and must still be legible to player.

Game Quality Rankings

All games deemed suitable for publication on Stake Engine receive a Quality Ranking from **1** to **3** ★'s, where **1 is the lowest** and **3 is the highest**.
This ranking determines a game's visibility and positioning eligibility.

Ranking Tiers

Rank	Description	Promotion & Visibility
★★★	Awarded only to studio-quality games showing exceptional creativity, uniqueness and attention to detail.	• Optimal positioning and eligible for prominent display in Burst Games, Stake Exclusives, and/or the *featured* section of New Releases.
★★	Given to games that show considerable creativity or originality. While they may lack polish compared to more established studios, they still demonstrate strong development quality and attention to detail.	• Can appear in Burst Games or Stake Exclusives if driven by user popularity. • Placement in New Releases depends on space and demand.

Rank	Description	Promotion & Visibility
★	Games of lower polish that still meet publishing requirements.	• Published with limited visibility. Always placed at the bottom of New Releases. • Not included in promotional categories unless driven by exceptional user demand.

Review Priority

When a game is submitted for review, it will first receive an initial star rating prior to a comprehensive evaluation. Both new and ongoing reviews are prioritised according to the game's current star rating.

Category Placement Guidelines (updated weekly)

New Releases

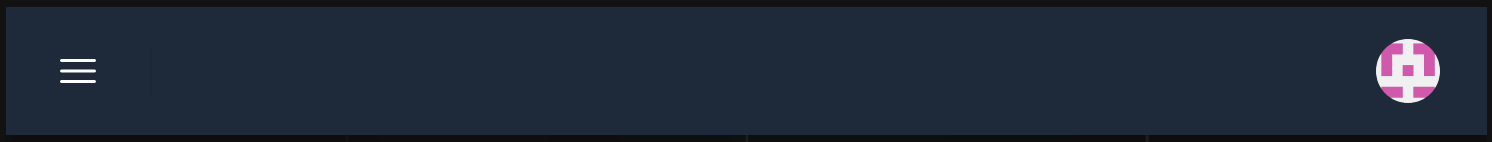
- All games receive a New Release tag.
- **Rank 3**: Prioritized placement in *featured* and top positioning of New Releases.
- **Rank 2**: May appear in weekly releases if space allows, otherwise placed lower.
- **Rank 1**: Always remain at the bottom of New Releases, unless sorted by Newest.

Burst Games

- Priority given to Rank 3 games.
- Rank 2 games may appear in this category if popularity drives demand.

Stake Exclusives

- Priority given to Rank 3 games.
- Rank 2 games may appear within this section if driven by popularity.



Game Tile Visual Asset Requirements

With each game submission, they must include the submission of visual assets to be used to create the game tile.

It's important to include high quality, visually appealing assets in order to create a game tile that will appeal to players and entice them to click the game. Games with artwork that look to be of low quality or are visually unappealing often result in lower player trust, lower interest and ultimately lower game engagement.

For the creation of each game tile, we require the following assets:

- **Background image**
- **Foreground image**
- **Provider Logo**

Background image

- An environmental background that shows the world of the game
- File format: High resolution **PNG** or **JPG** file
- Minimum size dimensions: **1200px x 1200px @ 72dpi**
- Naming convention: `` `GameTitle-BG.format` `` (e.g., `` `CrownConquest-BG.png` `` or `` `PixelCastle-BG.jpg` ``)

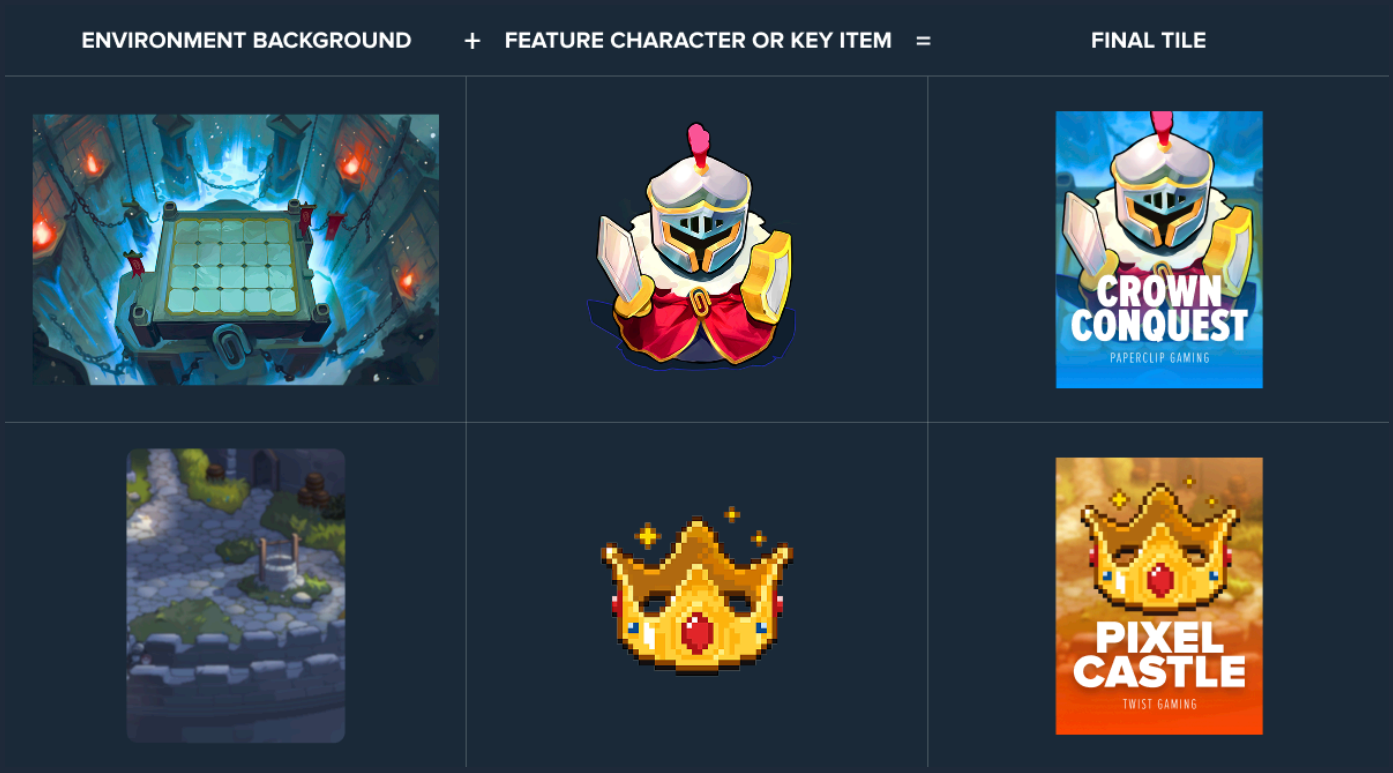
Foreground image

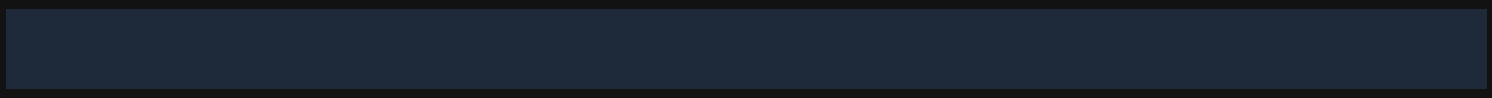
- A feature character or key item that represents the game

- File format: High resolution **PNG** with a transparent background
- Minimum size dimensions: **1200px x 1200px @ 72dpi**
- Naming convention: `GameTitle-FG.png` (e.g., `CrownConquest-FG.png`)

Provider Logo

- The official logo of the game provider or studio
- File format: High resolution **PNG** with a transparent background
- Naming convention: `ProviderName-Logo.png` (e.g., `ZuckGames-Logo.png`)
- Should be clear and legible at small sizes





General Game Disclaimer

The game rules/information popup must include a brief disclaimer regarding game operation. Since Stake Engine utilises pre-calculated game results, payouts are dictated purely by the RGS response and are not influenced by events on the frontend. You are able to use our template disclaimer, or your own, so long as the same message is clearly conveyed.

General Disclaimer:

Malfunction voids all wins and plays. A consistent internet connection is required. In the event of a disconnection, reload the game to finish any uncompleted rounds. The expected return is calculated over many plays. The game display is not representative of any physical device and is for illustrative purposes only. Winnings are settled according to the amount received from the Remote Game Server and not from events within the web browser. TM and © 2025 Stake Engine.

Jurisdiction Requirements

For games to be avaliable on stake.us, US requriements prohibit the use of certain gambling terms. This predominantly applies to game rules but also potentially extends to images and UI elments. For your game to be approved for release on stake.us, your game cannot contain any of the terms listed below.

The RGS uses the URL query parameter `social=true/false` to indicate wheather or not the game is loaded in a 'social' casino. We reccomend using an additional language file with the prefix: `sweeps_<lang>` to handle phrase changes.

A table of prohibited terms is given below, along with suggested replacement phrases:

Restricted Phrase	Replacement Phrase
win feature	play feature
pay out	win / won
paid out	win
stake	play amount
pays out	won
betting	play / playing
total bet	total play
bet	play
bets	plays

Restricted Phrase	Replacement Phrase
cash	coins
payer	winner
pay	win
pays	wins
paid	won
money	coins
buy	play
bought	instantly triggered
purchase	play
at the cost of	for
rebet	respin
cost of	can be played for
credit	coins
buy bonus	get bonus
gamble	play
wager	play
deposit	get coins
withdraw	redeem
bonus buy	bonus / feature
be awarded to player’s accounts	appear in player’s accounts
betting	playing
total bet	play
pay out	win / won

Restricted Phrase	Replacement Phrase
paid out	won
place your bets	come and play / join in the game
pays out	win
win feature	play feature
bet/s	play/s
currency	token

Math Verification

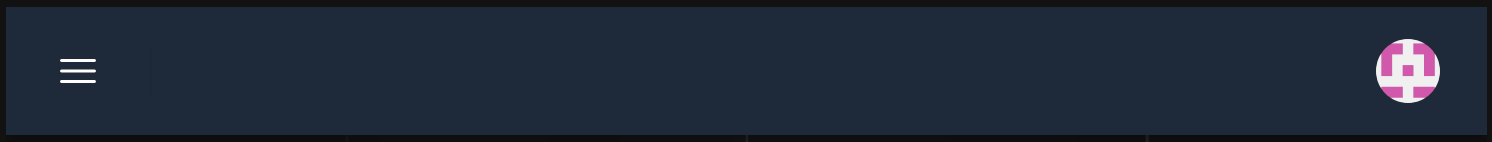
Summary statistics and hit-rate tables will be analyzed to ensure the game adheres to industry standards for chance-based casino games and is not misleading.

Summary Statistics

- Verify the mode cost is correctly represented in the game rules for each mode.
- The calculated Return to Player (RTP) must be within 90.0%–99.0%. For multiple modes, all must fall within a 0.5% variation (e.g., base game at 97% RTP requires other modes to be between 96.5% and 97.5%).
- Ensure the maximum win amount matches the description in the game rules for each mode.
- The maximum win must be realistically obtainable (typically more frequent than 1 in 10,000,000, depending on payout amount).
- For slot-type games, run 100,000–1,000,000 simulations to ensure sufficient outcome diversity and avoid repeated results in a single session.
- A reasonable portion of simulations should yield paying results (e.g., 90,000 non-paying results out of 100,000 may be grounds for rejection).
- The hit-rate of the most likely single simulation should not be overwhelmingly dominant if there is a visual expectation that results are sufficiently varied.

Other Considerations

- The hit-rate of non-zero wins should align with industry standards (<1 in 20 bets, or more frequent).
- For “BASE” modes (1x cost), the standard deviation should be within industry norms to ensure reasonable volatility for slot-type games.
- List the number of non-zero weight payouts. Zero-weight payouts should not dominate the provided simulations.
- Inspect hit-rates for win-ranges to avoid gaps where expected win amounts are unobtainable (e.g., intermediate wins should exist between small payouts and the maximum payout amount).



Remote Game Server (RGS) Communication

Session authentication and bet transactions are handled exclusively through the Stake Engine RGS. The RGS manages session token generation, *play/* responses, and optional parameters like supported currencies and languages.

RGS Authentication

- **Bet Level Verification:** The *authenticate* HTTP response returns default bet levels, supported bet levels for a specified currency, and minimum/maximum bet amounts. The frontend must respect these values. Example: If the default bet size is 1 unit but the session uses JPY (minimum bet size: 10 units), the *play/* request will fail.
- Bet increments must reflect allowed values within *authenticate/config/minStep*.
- Minimum and maximum bet levels must be available for selection as dictated by the RGS.

Cross-Site-Scripting (XSS)

- Stake Engine enforces a strict XSS policy. The game build must consist only of static files and cannot reach external sources. Common issues include downloading fonts from external servers, which logs console errors.

RGS URL

- The game must use the *rgs_url* query parameter to determine the server to call.

Currency and Language

English is the only required language. If only English (en) is supported, on-screen text must not corrupt when other language parameters are passed.

Supported Languages

Language	Abbreviation
Arabic	ar
German	de
English	en
Spanish	es
Finnish	fi
French	fr
Hindi	hi
Indonesian	id
Japanese	ja
Korean	ko
Polish	po
Portuguese	pt
Russian	ru
Turkish	tr
Chinese	zh
Vietnamese	vi

Supported Currencies

Currency	Abbreviation	Display	Example
United States Dollar	USD	$	$10.00
Canadian Dollar	CAD	CA$	CA$10.00
Japanese Yen	JPY	¥	¥10
Euro	EUR	€	€10.00
Russian Ruble	RUB	₽	₽10.00
Chinese Yuan	CNY	CN¥	CN¥10.00
Philippine Peso	PHP	₱	₱10.00
Indian Rupee	INR	₹	₹10.00
Indonesian Rupiah	IDR	Rp	Rp10
South Korean Won	KRW	₩	₩10
Brazilian Real	BRL	R$	R$10.00
Mexican Peso	MXN	MX$	MX$10.00
Danish Krone	DKK	KR	10.00 KR
Polish Złoty	PLN	zł	10.00 zł
Vietnamese Đồng	VND	₫	10 ₫
Turkish Lira	TRY	₺	₺10.00
Chilean Peso	CLP	CLP	10 CLP
Argentine Peso	ARS	ARS	10.00 ARS
Peruvian Sol	PEN	S/	S/10.00
Nigerian Naira	NGN	₦	₦10.00
Saudi Arabia Riyal	SAR	SAR	10.00 SAR

Currency	Abbreviation	Display	Example
Israel Shekel	ILS	ILS	10.00 ILS
United Arab Emirates Dirham	AED	AED	10.00 AED
Taiwan New Dollar	TWD	NT$	NT$10.00
Norway Krone	NOK	kr	kr10.00
Kuwaiti Dinar	KWD	KD	KD10.00
Jordanian Dinar	JOD	JD	JD10.00
Costa Rica Colon	CRC	₡	₡10.00
Tunisian Dinar	TND	TND	10.00 TND
Singapore Dollar	SGD	SG$	SG$10.00
Malaysia Ringgit	MYR	RM	RM10.00
Oman Rial	OMR	OMR	10.00 OMR
Qatar Riyal	QAR	QAR	10.00 QAR
Bahraini Dinar	BHD	BD	BD10.00
Stake Gold Coin	XGC	GC	10.00 GC
Stake Cash	XSC	SC	10.00 SC

Find code examples for displaying these values at https://stake-engine.com/docs/rgs

Stake Engine Software Development Kit

Frontend - SDK

Why Use the frontend SDK?

The frontend-sdk is a PixieJS/Svelte package used for developing web-based slot games in a declarative way. This package walks though how to utilize powerful tools such as Turborepo and Storybook to test and publish slot games. Sample slot games are provided which consume outputs provided by the math-sdk, though the repo is customizable and can be tailored to accommodate custom events for slot games covering all levels of complexity.

See Frontend SDK Technical Details for more details.

Steps to Add a New BookEvent

For example, we have a game `/apps/lines` already. Assume that we have added a new bookEvent `updateGlobalMult` to the bonus game mode (`MODE_BONUS`) in math, so that we have a new global multiplier feature for the game. Based on that, here we will go through the steps together to implement this new bookEvent and add it to the game. Along the way we will introduce part of our file structure as well.

- `/apps/lines/src/stories/data/bonus_books.ts`: This file includes the an array of bonus books that story `MODE_BONUS/book/random` will randomly pick at. This is to simulate requesting data from RGS. All we need to do is to copy/paste data from our new math package and format it.a

```
// bonus_books.ts

{
  type: 'updateGlobalMult',
  globalMult: 3,
},
```

- `/apps/lines/src/stories/data/bonus_events.ts`: This file includes the an object of every type of bookEvent that story `MODE_BONUS/bookEvent/<BOOK_EVENT_TYPE>` uses. All we need to do is to copy/paste data from our new math package and format it.

```
// bonus_events.ts

export default {
  ...,
  updateGlobalMult: {
    type: 'updateGlobalMult',
```

```
    globalMult: 3,
  },
  ...,
}
```

- `/apps/lines/src/stories/ModeBonusBookEvent.stories.svelte`: This file implements all the sub stories in story set `MODE_BONUS/bookEvent`. With the following code added in this file, you will see the a new story `MODE_BONUS/bookEvent/updateGlobalMult` that is added in our storybook with an `Action` button. Now if we click on it and nothing would happen, but it is a good start because we set up the testing environment first. Next step is to add code of bookEventHandler to handle it.

```
// ModeBonusBookEvent.stories.svelte

<Story
  name="updateGlobalMult"
  args={templateArgs({
    skipLoadingScreen: true,
    data: events.updateGlobalMult,
    action: async (data) => await playBookEvent(data, { bookEvents: [] }),
  })}
/>
```

- `/apps/lines/src/game/typesBookEvent.ts`: This file contains typescript types of all the bookEvents. Let is add the type of our new bookEvent to get the intellisense from typescript for the following step.
  - `type BookEvent` is a `union type` (typescript union type) of BookEvent types.

```
// typesBookEvent.ts

type BookEventUpdateGlobalMult = {
  index: number;
  type: 'updateGlobalMult';
  globalMult: number;
};

export type BookEvent =
```

```
  | ...
  | BookEventUpdateGlobalMult
  | ...
;
```

- `/apps/lines/src/game/bookEventHandlerMap.ts` : This file includes all the bookEventHandlers. Let is add a new one for the new bookEvent. Check the intellisense that the previous step brings, it provides a better developer experience.

- `/apps/lines/src/components/GlobalMultiplier.svelte` : This file is created as our target svelte component for updateGlobalMulti bookEvent. Technically speaking, all the jobs that is related to global multiplier of the game should only be in this svelte component. Similar to the bookEvent types, let is add the typescript types for new emitterEvents first.
  - `type EmitterEventGlobalMultiplier` is a union type of EmitterEvent types.

```
// GlobalMultiplier.svelte

<script lang="ts" module>
  export type EmitterEventGlobalMultiplier =
    | { type: 'globalMultiplierShow' }
    | { type: 'globalMultiplierHide' }
    | { type: 'globalMultiplierUpdate'; multiplier: number };
</script>
```

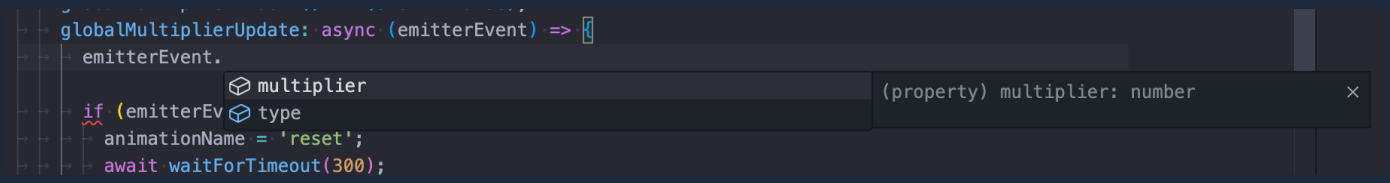
- `/apps/lines/src/game/typesEmitterEvent.ts` : This file has typescript types of all the emitterEvents of the game. Let is add the type of our new emitterEvents for intellisense.
  - `type EmitterEventGame` is a union type of EmitterEvent types.

```
// typesEmitterEvent.ts

...
import type { EmitterEventGlobalMultiplier } from '../components/GlobalMultiplier.svelte';
...

export type EmitterEventGame =
  | ...
  | EmitterEventGlobalMultiplier
  | ...
;
```

- `/apps/lines/src/game/eventEmitter.ts` : This file exports the eventEmitter, it uses the `EmitterEventGame` and other EmitterEvent types to compose `type EmitterEvent` .
  - `type EmitterEvent` is a union type of EmitterEvent types.

```
// eventEmitter.ts

...
import type { EmitterEventGame } from './typesEmitterEvent';
export type EmitterEvent = EmitterEventUi | EmitterEventHotKey | EmitterEventGame;
export const { eventEmitter } = createEventEmitter<EmitterEvent>();
```

- `/apps/lines/src/components/GlobalMultiplier.svelte` : Back to our component file, the intellisense is there. Let is add the code to process the values with a spine animation as well.




```
// GlobalMultiplier.svelte

<script lang="ts" module>
  export type EmitterEventGlobalMultiplier =
    | { type: 'globalMultiplierShow' }
    | { type: 'globalMultiplierHide' }
    | { type: 'globalMultiplierUpdate'; multiplier: number };
</script>

<script lang="ts">
  ...

  context.eventEmitter.subscribeOnMount({
    globalMultiplierShow: () => (show = true),
    globalMultiplierHide: () => (show = false),
    globalMultiplierUpdate: async (emitterEvent) => {
      console.log(emitterEvent.multiplier)
    },
  });
</script>

<SpineProvider key="globalMultiplier" width={PANEL_WIDTH}>
  ...
  <SpineTrack trackIndex={0} {animationName} />
</SpineProvider>
```

- Test it individually `(MODE_BONUS/bookEvent/updateGlobalMult)`: Run storybook and we should see this a new story "updateGlobalMult" has been added.
  - Now click on the `Action` button and we should see the `<GlobalMultiplier \/>` (`/apps/lines/src/components/GlobalMultiplier.svelte`) component animates correctly followed by the "ⓘ Action is resolved ✅" message, otherwise we need to go back to the component and figure out what is wrong until it is resolved.

- If you find out the component hard to debug, we'd better start creating a new story `` `COMPONENTS/<GlobalMultiplierSpine>/component` `` . `` `<GlobalMultiplierSpine />` `` component will purely take props and achieve its duty instead of being controlled by emitterEvents. This way it becomes more friendly for testing the component with the storybook controls.

- Test it in books `(MODE_BONUS/book/random)`: Final step is to test it in a book environment by switching to this book story. In a previous step we have updated `/apps/lines/src/stories/data/bonus_books.ts`, so the new bookEvent will appear if we keep hitting the `Action` button in this story.

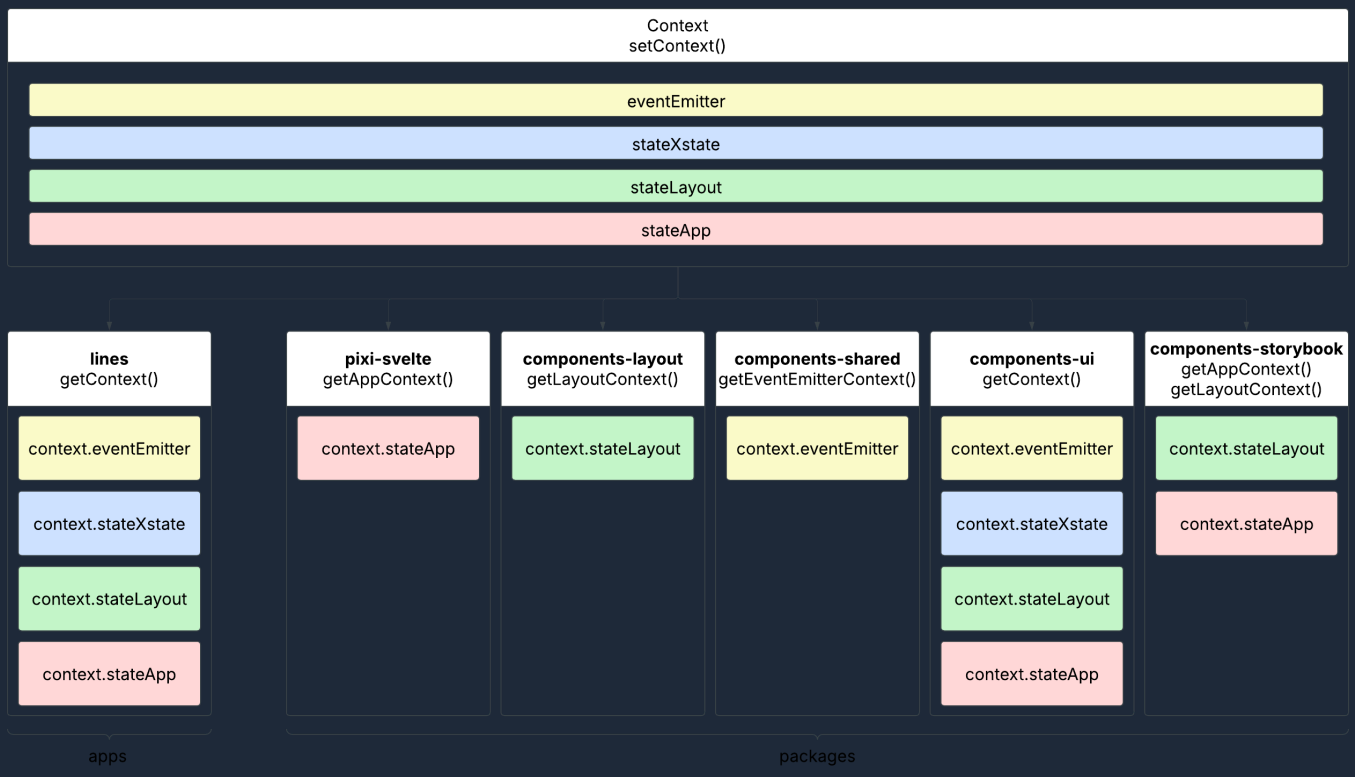
Context

- ContextEventEmitter
- ContextLayout
- ContextXstate
- ContextApp

[svelte-context](https://svelte.dev/docs/svelte/context) is a useful feature from svelte especially when a shared state requires some inputs/types to create. Here it shows the structure of context of sample game `/apps/lines`. As showed before, `setContext()` is called at entry level component. For example, `apps/lines/src/routes/+page.svelte` or `apps/lines/src/stories/ComponentsGame.stories.svelte`. It sets four major contexts from the packages by this:

```
// context.ts - Example of setContext in apps

export const setContext = () => {
  setContextEventEmitter<EmitterEvent>({ eventEmitter });
  setContextXstate({ stateXstate, stateXstateDerived });
  setContextLayout({ stateLayout, stateLayoutDerived });
  setContextApp({ stateApp });
};
```

Different apps and packages require different contexts.



ContextEventEmitter

`eventEmitter` is created by `packages/utils-event-emitter/src/createEventEmitter.ts`. We have covered eventEmitter in the previous content.

ContextLayout

`stateLayout` and `stateLayoutDerived` are created by `packages/utils-layout/src/createLayout.svelte.ts`. It provides canvasSizes, canvasRatio, layoutType and so on. Because we have a setting `resizeTo: window` for PIXI.Application, we use the sizes of window from svelte-reactivity as `canvasSizes`.

For html, the tags will auto-flow by default. However, in the canvas/pixijs we need to set positions manually to avoid overlapping. The importance of LayoutContext is that it provides us the values of boundaries (canvasSizes), device type based on the dimensions (layoutType) and so on. For example:

- Set a pixi-svelte component to the left edge of the canvas:
  - `` `<Component x={0} />` ``
- Set a pixi-svelte component to the right edge of the canvas:
  - `` `<Component x={context.stateLayoutDerived.canvasSizes().width} anchor={{ x: 1: y: 0 }} />` ``
  - It works when `` `<App />` `` is the parent of the component, otherwise it will be determined by its parent `` `<Container />` ``.
  - The reason why we set `` `anchor` `` is because that the drawing is always go from top-left to bottom-right in pixijs.

```
// createLayout.svelte.ts

import { innerWidth, innerHeight } from 'svelte/reactivity/window';

...

const stateLayout = $state({
  showLoadingScreen: true,
});

const stateLayoutDerived = {
  canvasSizes,
  canvasRatio,
  canvasRatioType,
  canvasSizeType,
  layoutType,
  isStacked,
  mainLayout,
  normalBackgroundLayout,
  portraitBackgroundLayout,
};
```

ContextXstate

`stateXstate` and `stateXstateDerived` are created by `packages/utils-xstate/src/createXstateUtils.svelte.ts`. It provides a few functions to check the state of finite state machine, also known as `gameActor`, which is created by `packages/utils-xstate/src/createGameActor.svelte.ts`.

```
// createXstateUtils.svelte.ts

import { matchesState, type StateValue } from 'xstate';

...

const stateXstate = $state({
  value: '' as StateValue,
});

const matchesXstate = (state: string) => matchesState(state, stateXstate.value);

const stateXstateDerived = {
  matchesXstate,
  isRendering: () => matchesXstate(STATE_RENDERING),
  isIdle: () => matchesXstate(STATE_IDLE),
  isBetting: () => matchesXstate(STATE_BET),
  isAutoBetting: () => matchesXstate(STATE_AUTOBET),
  isResumingBet: () => matchesXstate(STATE_RESUME_BET),
  isForcingResult: () => matchesXstate(STATE_FORCE_RESULT),
  isPlaying: () => !matchesXstate(STATE_RENDERING) && !matchesXstate(STATE_IDLE),
};
```

`gameActor`: To avoid using massive “if-else” conditions in the code, we use npm/xstate to create a finite state machine to handle the complicated logic and states of betting. It provides a few pre-defined mechanics like one-off `bet`, `autoBet` with a count down, `resumeBet` to continue an unfinished bet and so on.

```
// createGameActor.svelte.ts

import { setup, createActor } from 'xstate';

...
```

```
const gameMachine = setup({
  actors: {
    bet: intermediateMachines.bet,
    autoBet: intermediateMachines.autoBet,
    resumeBet: intermediateMachines.resumeBet,
    forceResult: intermediateMachines.forceResult,
  },
}).createMachine({
  initial: 'rendering',
  states: {
    [STATE_RENDERING]: stateRendering,
    [STATE_IDLE]: stateIdle,
    [STATE_BET]: stateBet,
    [STATE_AUTOBET]: stateAutoBet,
    [STATE_RESUME_BET]: stateResumeBet,
    [STATE_FORCE_RESULT]: stateForceResult,
  },
});

const gameActor = createActor(gameMachine);
```

This is highly useful when it comes to the interactions with UI, for example disable the bet button when the a game is playing.

```
// BetButton.svelte - Example of interaction between xstate and UI

<script lang="ts">
  import { getContext } from '../context';

  const context = getContext();
</script>

<SimpleUiButton disabled={context.stateXstateDerived.isPlaying()} />
```

AppContext

`stateApp` is created by `packages/pixi-svelte/src/lib/createApp.svelte.ts`. `loadedAssets` contains the static images, animations and sound data that is processed by `PIXI.Assets.load` with `stateApp.assets`. `loadedAssets` can be digested by pixi-svelte components directly as showed in pixi-svelte component `\<Sprite /\>` (`/packages/pixi-svelte/src/lib/components/Sprite.svelte`).

```ts
// createApp.svelte.ts

const stateApp = $state({
  reset,
  assets,
  loaded: false,
  loadingProgress: 0,
  loadedAssets: {} as LoadedAssets,
  pixiApplication: undefined as PIXI.Application | undefined,
});
```

Dependencies

Besides basic web skills (html, css and javascript), here it shows a list of npm dependencies of this repo. It would be great to start with understanding them before kicking off.

- pixijs: https://www.npmjs.com/package/pixi.js and more...
- svelte: https://www.npmjs.com/package/svelte and more...
- turborepo: https://www.npmjs.com/package/turbo and more...
- pixi-svelte: https://www.npmjs.com/package/pixi-svelte and more...
    - This is an in-house npm package. It combines pixi and svelte together and uses pixijs in a declarative way.
- sveltekit: https://www.npmjs.com/package/@sveltejs/kit and more...
- storybook: https://www.npmjs.com/package/storybook and more...
- xstate: https://www.npmjs.com/package/xstate and more...
- typescript: https://www.npmjs.com/package/typescript and more...
- pnpm: https://www.npmjs.com/package/pnpm and more...

File Structure

The file structure is in a way of structure of TurboRepo to achieve a monorepo. Besides the files for the configurations of TurboRepo, sveltekit, eslint, typescript, git and so on, here is a list of of key modules of (`/apps`) and `/packages` .

```
root
  |_apps
  |  |_cluster
  |  |_lines
  |  |_price
  |  |_scatter
  |  |_ways
  |
  |_packages
     |_config-*
     |_constants-*
     |_state-*
     |_utils-*
     |_components-*
     |_pixi-*
```

/apps

For each game, it has an individual folder in the apps, for example `/apps/lines` .

- `/apps/lines/package.json` : Find the module name of the app here.

```
{
  "name": "lines",
```

```
  ...
}
```

- To run the app in DEV mode instead of in the storybook: Run `pnpm run dev --filter=<MODULE_NAME>` in the terminal.

```
pnpm run dev --filter=lines
```

- `/apps/lines/src/routes/%2Bpage.svelte`: This is the entry file of sample game apps/lines in a sveltekit way. It is a combination of two things:
  - `setContext()` (`/apps/lines/src/game/context.ts#L14`): A function that sets all the svelte-context required and used in this app and in the `/packages`. As we already know, only children-level components can access the context. That is why we set the context at the entry level of the app.
  - `<Game \/>` (`/apps/lines/src/components/Game.svelte`): The entry svelte component to the game. It includes all the components of the game.

```
// +page.svelte

<script lang="ts">
  import Game from '../components/Game.svelte';
  import { setContext } from '../game/context';

  setContext();
</script>

<Game />
```

- `/apps/lines/src/stories/ComponentsGame.stories.svelte`: You will find the same pattern in this storybook or other `Mode<GAME_MODE>Book.stories.svelte` and `Mode<GAME_MODE>BookEvent.stories.svelte`.

```
// ComponentsGame.stories.svelte

<script lang="ts">
  ...
  import Game from '../components/Game.svelte';
  import { setContext } from '../game/context';

  ...
  setContext();
</script>

<Story name="component (loadingScreen)">
  <StoryLocale lang="en">
    <Game />
  </StoryLocale>
</Story>
```

- We can render `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component in the app or in the storybook. Either way it requires the context to set in advance, otherwise the children or the descendants will throw errors if they use the `getContext()` (`/apps/lines/src/game/context.ts#L21`) from `/apps` or `getContext()` (`/packages/components-ui-pixi/src/context.ts#L8`) from `/packages` .

/packages

For every TurboRepo local package, you can import and use them in an app or in another local package directly without publishing them to npm. Our codebase benefits considerably from a monorepo because it brings reusability, readability, maintainability, code splitting and so on. Here is an example of importing local packages with `workspace:*` in `/apps/lines/package.json` :

```
// package.json

{
  "name": "lines",
  ...,
  "devDependencies": {
    ...,
```

```
    "config-ts": "workspace:*",
  },
  "dependencies": {
    ...,
    "pixi-svelte": "workspace:*",
    "constants-shared": "workspace:*",
    "state-shared": "workspace:*",
    "utils-shared": "workspace:*",
    "components-shared": "workspace:*",
  }
}
```

The naming convention of packages is a combination of `<PACKAGE_TYPE>`, hyphen and `<SPECIAL_DEPENDENCY>` or `<SPECIAL_USAGE>`. For example, `components-pixi` is a local package that the package type is “components” and the special dependency is `pixi-svelte`.

- `config-*`:
  - `/packages/config-lingui`: This local package contains reusable configurations of npm package lingui.
  - `/packages/config-storybook`: This local package contains reusable configurations of npm package storybook.
  - `packages/config-svelte`: This local package contains reusable configurations of npm package svelte.
  - `/packages/config-ts`: This local package contains reusable configurations of npm package typescript.
  - `/packages/config-vite`: This local package contains reusable configurations of npm package vite.
- `pixi-*`
  - `/packages/pixi-svelte`: This local package contains reusable svelte components/functions/types based on pixijs and svelte.
    - It creates `stateApp` and `AppContext` as a svelte-context.
    - It also builds and publishes pixi-svelte of npm.
  - `packages/pixi-svelte-storybook`: This is a storybook for components in `pixi-svelte`.

- `constants-*` :
  - `/packages/constants-shared` : This local package contains reusable global constants.
- `state-*` :
  - `/packages/state-shared` : This local package contains reusable global svelte-$state.
- `utils-*` :
  - `/packages/utils-book` : This local package contains reusable functions/types that are related to book and bookEvent.
  - `/packages/utils-fetcher` : This local package contains reusable functions/types based on fetch API.
  - `/packages/utils-shared` : This local package contains reusable functions/types, except for lodash and lingui.
  - `/packages/utils-slots` : This local package contains reusable functions/types for slots game, for example creating reel and spinning the board.
  - `/packages/utils-sound` : This local package contains reusable functions/types based on npm package howler for music and sound effect.
  - `/packages/utils-event-emitter` : This local package contains reusable functions/types to achieve our event-driven programming.
    - It creates `eventEmitter` and `ContextEventEmitter` as a svelte-context
  - `/packages/utils-xstate` : This local package contains reusable functions/types based on npm package xstate.
    - It creates `stateXstate`, `stateXstateDerived` and `ContextXstate` as a svelte-context
  - `/packages/utils-layout` : This local package contains reusable functions/types for our layout system of pixijs.
    - It creates `stateLayout`, `stateLayoutDerived` and `ContextLayout` as a svelte-context
- `components-*` :

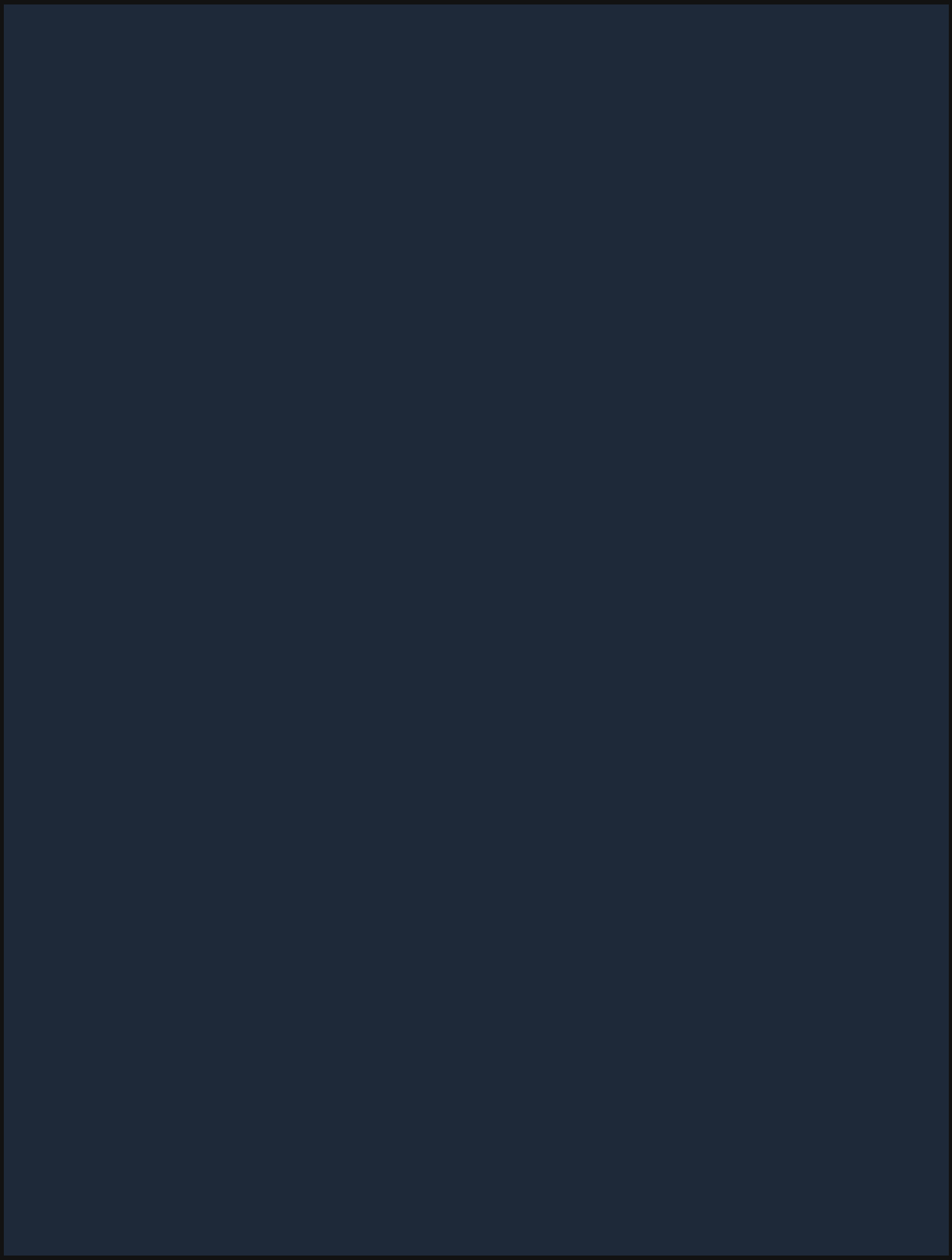
- `/packages/components-layout` : This local package contains reusable svelte components based on another local package `utils-layout`.
- `/packages/components-pixi` : This local package contains reusable svelte components based on `pixi-svelte`.
- `/packages/components-shared` : This local package contains reusable svelte components based on `html`.
- `/packages/components-storybook` : This local package contains reusable svelte components for storybooks.
- `/packages/components-ui-pixi` : This local package contains reusable svelte pixi-svelte components for the game UI.
- `packages/components-ui-html` : This local package contains reusable svelte html components for the game UI.

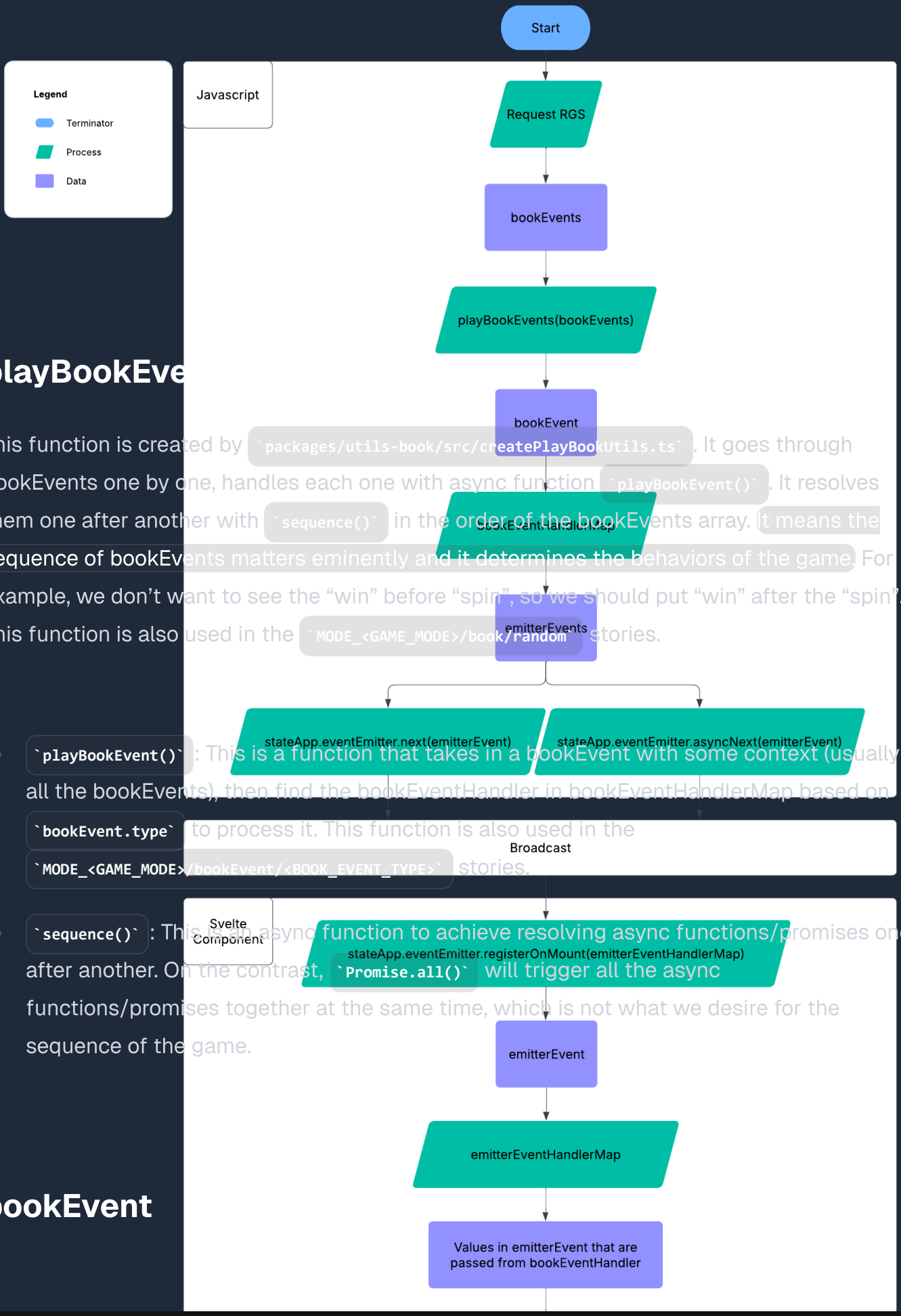
For `*-shared` packages, they are created to be reused as much as possible by other apps and packages. Instead of having a special dependency or usage, they should have a minimum list of dependencies and a broad set of use cases.

`pixi-svelte`, `utils-event-emitter`, `utils-layout` and `utils-xstate` they have functions to create corresponding svelte-context. For the contexts, they can be used by either an app or a local `components-*` package by just calling the `getContext<CONTEXT_NAME>()`. For example, components in `components-layout` use `getContextLayout()` from `utils-layout`. In this way, we can regard `pixi-svelte` as an integration of “utils-pixi-svelte” and “components-pixi-svelte”.

Flow Chart

Here it is a simplified flow chart of steps how a game is processed after RGS request. The real situation might be more complicated, but it follows the same idea.



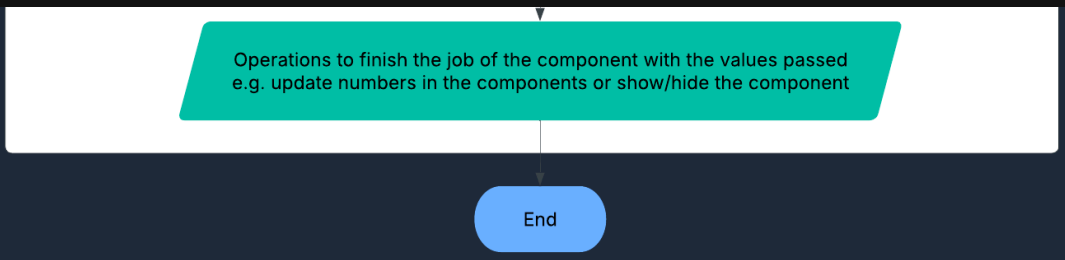


playBookEve

This function is created by `packages/utils-book/src/createPlayBookUtils.ts`. It goes through bookEvents one by one, handles each one with async function `playBookEvent()`. It resolves them one after another with `sequence()` in the order of the bookEvents array. It means the sequence of bookEvents matters eminently and it determines the behaviors of the game. For example, we don't want to see the "win" before "spin", so we should put "win" after the "spin". This function is also used in the `MODE_<GAME_MODE>/book/random` stories.

- `playBookEvent()`: This is a function that takes in a bookEvent with some context (usually all the bookEvents), then find the bookEventHandler in bookEventHandlerMap based on `bookEvent.type` to process it. This function is also used in the `MODE_<GAME_MODE>/bookEvent/<BOOK_EVENT_TYPE>` stories.
- `sequence()`: This is an async function to achieve resolving async functions/promises one after another. On the contrast, `Promise.all()` will trigger all the async functions/promises together at the same time, which is not what we desire for the sequence of the game.

bookEvent



- `book`: A book is a json data that is returned from the RGS (Remote Game Server) for each game requested. It is mainly composed by bookEvents.

```
// base_books.ts - Example of a base game book

{
  id: 1,
  payoutMultiplier: 0.0,
  events: [
    {
      index: 0,
      type: 'reveal',
      board: [
        [{ name: 'L2' }, { name: 'L1' }, { name: 'L4' }, { name: 'H2' }, { name: 'L1' }],
        [{ name: 'H1' }, { name: 'L5' }, { name: 'L2' }, { name: 'H3' }, { name: 'L4' }],
        [{ name: 'L3' }, { name: 'L5' }, { name: 'L3' }, { name: 'H4' }, { name: 'L4' }],
        [{ name: 'H4' }, { name: 'H3' }, { name: 'L4' }, { name: 'L5' }, { name: 'L1' }],
        [{ name: 'H3' }, { name: 'L3' }, { name: 'L3' }, { name: 'H1' }, { name: 'H1' }],
      ],
      paddingPositions: [216, 205, 195, 16, 65],
      gameType: 'basegame',
      anticipation: [0, 0, 0, 0, 0],
    },
    { index: 1, type: 'setTotalWin', amount: 0 },
    { index: 2, type: 'finalWin', amount: 0 },
  ],
  criteria: '0',
  baseGameWins: 0.0,
  freeGameWins: 0.0,
}
```

- `bookEvent`: A bookEvent is a json data that is one of the element of the `book.events` array.

```
// base_books.ts - Example of a "reveal" bookEvent

{
  index: 0,
  type: 'reveal',
  board: [
    [{ name: 'L2' }, { name: 'L1' }, { name: 'L4' }, { name: 'H2' }, { name: 'L1' }],
    [{ name: 'H1' }, { name: 'L5' }, { name: 'L2' }, { name: 'H3' }, { name: 'L4' }],
    [{ name: 'L3' }, { name: 'L5' }, { name: 'L3' }, { name: 'H4' }, { name: 'L4' }],
    [{ name: 'H4' }, { name: 'H3' }, { name: 'L4' }, { name: 'L5' }, { name: 'L1' }],
    [{ name: 'H3' }, { name: 'L3' }, { name: 'L3' }, { name: 'H1' }, { name: 'H1' }],
  ],
```

```
  paddingPositions: [216, 205, 195, 16, 65],
  gameType: 'basegame',
  anticipation: [0, 0, 0, 0, 0],
}

// base_books.ts - Example of a setTotalWin bookEvent

{ index: 1, type: 'setTotalWin', amount: 0 },
```

- `bookEventHandler`: An async function that takes in a bookEvent and do some operations with it. Usually it broadcasts some emitterEvents, so the components will receive and handle.

bookEventHandlerMap

An object that the key is `bookEvent.type` and value is a `bookEventHandler`. We can find an example in `/apps/lines/src/game/bookEventHandlerMap.ts`.

```
// bookEventHandlerMap.ts - Example of "updateFreeSpin" bookEventHandler

export const bookEventHandlerMap: BookEventHandlerMap<BookEvent, BookEventContext> = {
  ...,
  updateFreeSpin: async (bookEvent: BookEventOfType<'updateFreeSpin'>) => {
    eventEmitter.broadcast({ type: 'freeSpinCounterShow' });
    eventEmitter.broadcast({
      type: 'freeSpinCounterUpdate',
      current: bookEvent.amount,
      total: bookEvent.total,
    });
  },
  ...,
}
```

- In simple terms, a book is composed by multiple bookEvents. Different combinations of bookEvents will determine the different behaviours of a game e.g. win/lose, a big/small win, a base/bonus game, 1/10/15 spins and so on.

eventEmitter

It achieves event-driven programming for the development. It can either broadcast or subscribe to emitterEvents. It connects the javascript scope and svelte component scope with emitterEvents instead of passing the different states as svelte component props directly. The three most used functions are:

- `` `eventEmitter.broadcast()` ``
- `` `eventEmitter.broadcastAsync()` ``
- `` `eventEmitter.subscribeOnMount()` ``

emitterEvent

An emitterEvent is a json data that `` `eventEmitter.broadcast(emitterEvent)` `` or `` `eventEmitter.broadcastAsync(emitterEvent)` `` broadcasts, so that a component which has `` `eventEmitter.subscribeOnMount(emitterEventHandlerMap)` `` can receive the data and deal with it in a synchronous or asynchronous way.

For a game we have many animations, so sometimes we need to "await" for those animations to finish before going to the next step.

Conceptually a bookEvent is composed by emitterEvents. Nevertheless, the flexibility lies in that the emitterEvents composing a bookEvent can come from multiple different svelte components. This way we can achieve and control the interactions and timing between different svelte components for the same bookEvent, ultimately, to achieve our games.

```
// bookEventHandlerMap.ts - Example of an emitterEvent

{
  type: 'freeSpinCounterUpdate',
  current: undefined,
```

```
  total: bookEvent.totalFs,
}
```

- `EmitterEventHandler (Synchronous)`: A sync function that takes in an emitterEvent. It usually deals with some sync operations e.g. show/hide component, tidy up, update some numbers and so on.

```
// bookEventHandlerMap.ts - Example of broadcast

eventEmitter.broadcast({
  type: 'freeSpinCounterUpdate',
  current: undefined,
  total: bookEvent.totalFs,
});

// FreeSpinCounter.svelte - Example of receiving

context.eventEmitter.subscribeOnMount({
  ...,
  freeSpinCounterUpdate: (emitterEvent) => {
    if (emitterEvent.current !== undefined) current = emitterEvent.current;
    if (emitterEvent.total !== undefined) total = emitterEvent.total;
  },
  ...,
});
```

- `EmitterEventHandler (Asynchronous)`: An async function that takes in an emitterEvent. It usually deals with some async operations e.g. wait for fading in/out component, wait for animations to finish, wait for numbers to increase/decrease with svelte-tween and so on.

```
// bookEventHandlerMap.ts - Example of broadcastAsync

await eventEmitter.broadcastAsync({
  type: 'freeSpinIntroUpdate',
  totalFreeSpins: bookEvent.totalFs,
});

// FreeSpinIntro.svelte - Example of receiving
```

```
context.eventEmitter.subscribeOnMount({
  ...,
  freeSpinIntroUpdate: async (emitterEvent) => {
    freeSpinsFromEvent = emitterEvent.totalFreeSpins;
    await waitForResolve((resolve) => (oncomplete = resolve));
  },
  ...,
});
```

emitterEventHandlerMap

An object that the key is `` `emitterEvent.type` `` and value is an `` `emitterEventHandler` ``. We can find this object in each component. For example, (`` `/apps/lines/src/components/FreeSpinCounter.svelte` ``).

- Each emitterEventHandler can do a lot or a little, but we prefer each emitterEventHandler just doing a minimum job to achieve the duty that is described by its type. This way we follow the [Single Responsibility Principle of SOLID](https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#single-responsibility-principle). For example, `freeSpinCounterShow` just shows this component and does nothing more.

```
// FreeSpinCounter.svelte and its emitterEventHandlers

<script lang="ts" module>
  export type EmitterEventFreeSpinCounter =
    | { type: 'freeSpinCounterShow' }
    | { type: 'freeSpinCounterHide' }
    | { type: 'freeSpinCounterUpdate'; current?: number; total?: number };
</script>

<script lang="ts">
  ...

  context.eventEmitter.subscribeOnMount({
    freeSpinCounterShow: () => (show = true),
    freeSpinCounterHide: () => (show = false),
    freeSpinCounterUpdate: (emitterEvent) => {
      if (emitterEvent.current !== undefined) current = emitterEvent.current;
      if (emitterEvent.total !== undefined) total = emitterEvent.total;
```

```
    },
  });
</script>

<MainContainer>
  ...
</MainContainer>
```

Get started

Here is a complete tutorial to start our sample games in the storybook. Please ignore those steps that you already know or done.

- It is preferred to use VS Code as IDE. download
- Install node with version 18.18.0. download

```
# Download and install nvm:
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash

# in lieu of restarting the shell
\. "$HOME/.nvm/nvm.sh"

# Download and install Node.js:
nvm install 18.18.0

# Verify the node versions. Should print "v18.18.0".
node -v
```

- Install pnpm with version 10.5.0.

```
# Install pnpm
npm install pnpm@10.5.0 -g

# Verify the pnpm versions. Should print "v10.5.0"
pnpm -v
```

- Clone the repo to your local in VS Code terminal or others.

```
git clone &lt;REPO_CLONE_URL&gt;
cd web-sdk
```

- Install dependencies.

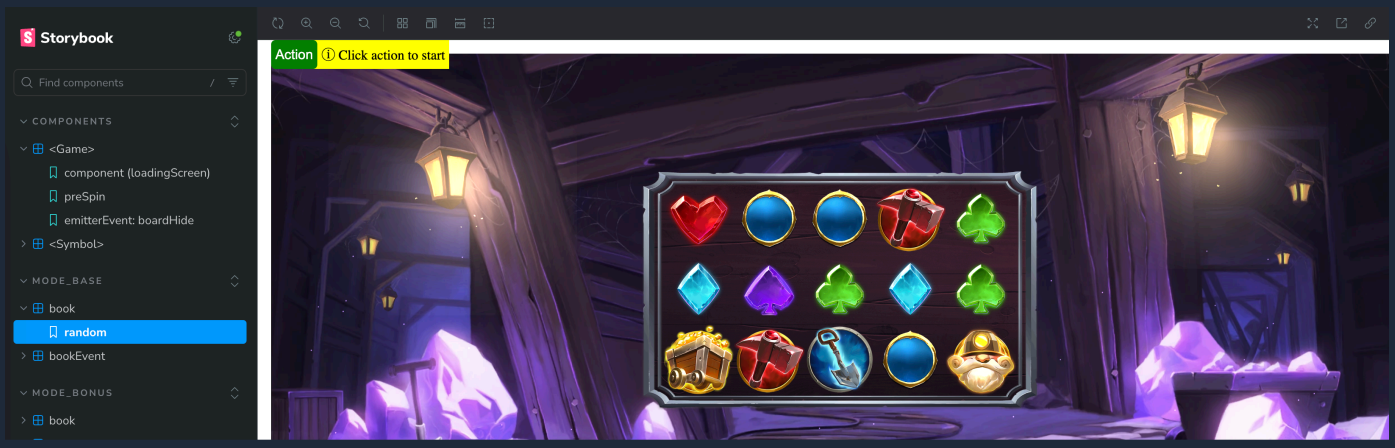
```
pnpm install
```

- Run `pnpm run storybook --filter=<MODULE_NAME>` in the terminal to see the storybook of a sample game in a TurboRepo way. `<MODULE_NAME>` is the name in the package.json file of a module in apps or packages folders.
- For example, we have `"name": "lines"` in the `/apps/lines/package.json`, so we can find it and run its storybook by:

```
pnpm run storybook --filter=lines
```

- You should see this:

- Now switch to `MODE_BASE/book/random` in the left sidebar, you will see an `Action` button appear on the left right conner of the game.



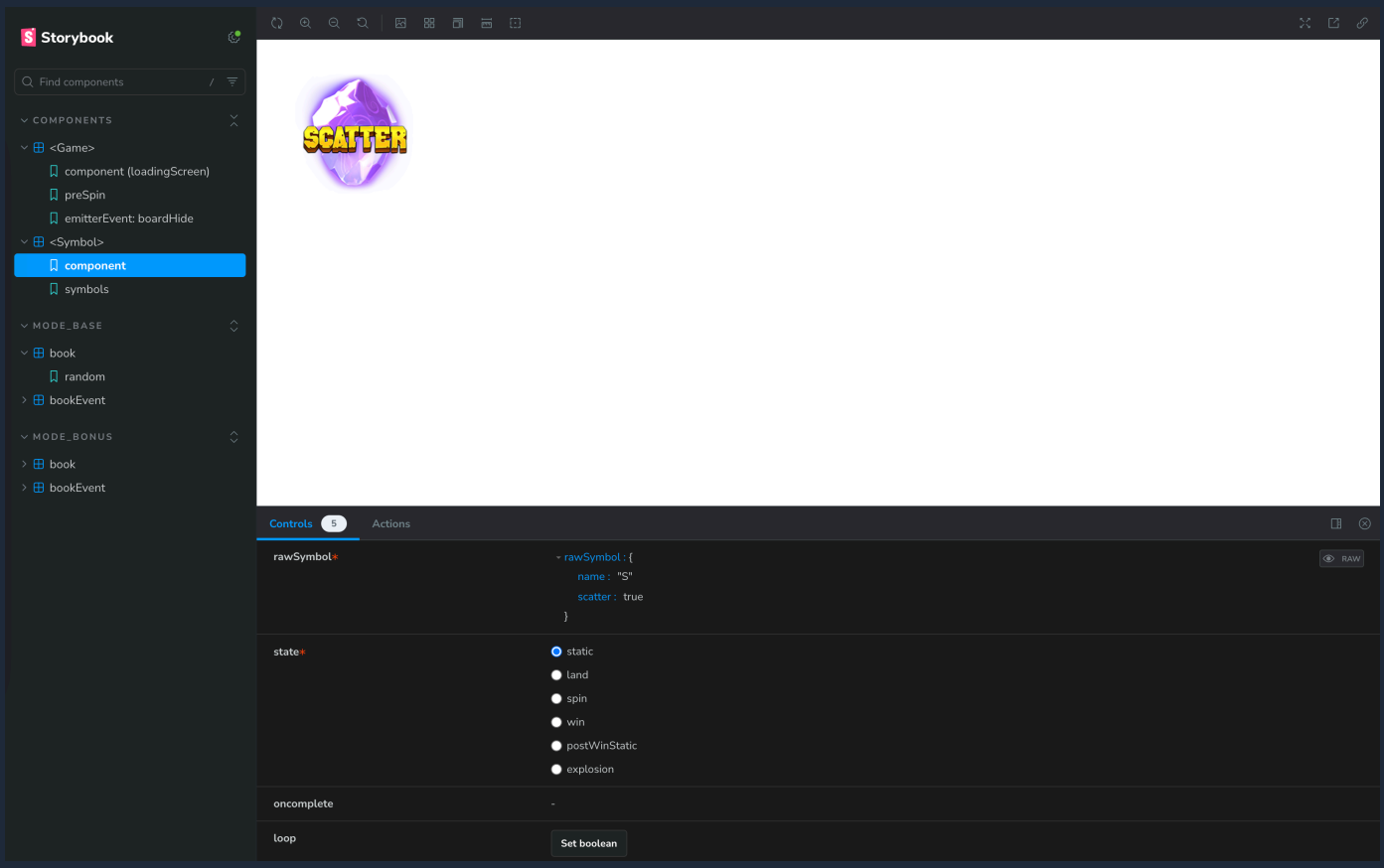
- Click on the `` `Action` `` button and wait for a base game to finish.
- Congratulations! You are now in the zone of game development with us now.

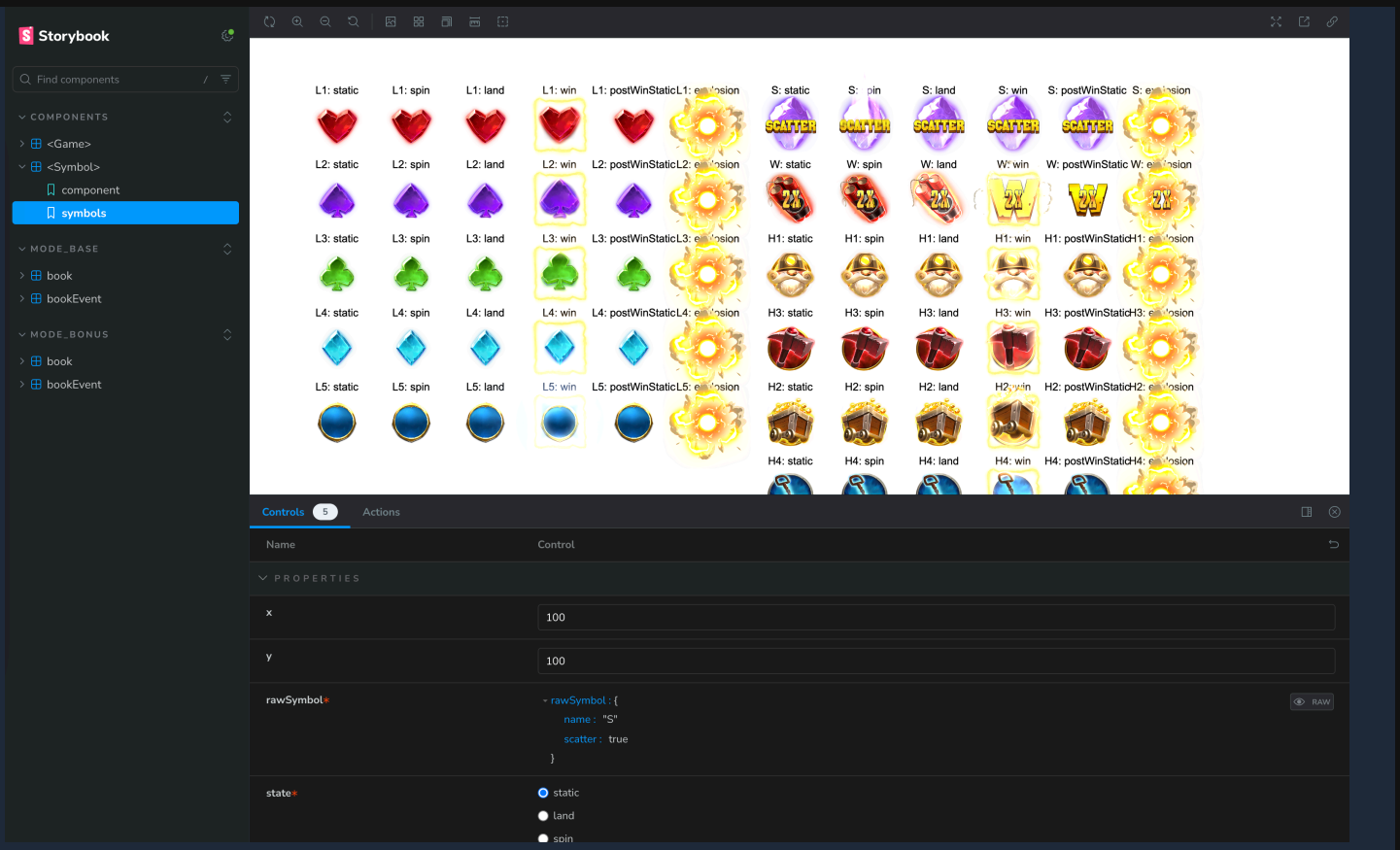
Explore Storybook

Storybook is a powerful and handy tool to test our games. For example:

- `COMPONENTS/<Game>/component` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component. In this case, it doesn't skip the loading screen.
- `COMPONENTS/<Game>/preSpin` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with the preSpin function.
- `COMPONENTS/<Game>/emitterEvent` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with an emitterEvent "boardHide".
- ...
- `COMPONENTS/<Symbol>/component` : It tests the `<Symbol \/>` (`/apps/lines/src/components/Symbol.svelte`) component with controls e.g. state of the symbol.
- `COMPONENTS/<Symbol>/symbols` : It tests the `<Symbol \/>` (`/apps/lines/src/components/Symbol.svelte`) component with all the symbols and all the states.
- ...
- `MODE_BASE/book/random` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with a random book of base mode.
- `MODE_BASE/bookEvent/reveal` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with a "reveal" bookEvent of the base mode. It will spin the reels.
- ...
- `MODE_BONUS/book/random` : It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with a random book of bonus mode.

- `MODE_BONUS/bookEvent/reveal`: It tests the `<Game \/>` (`/apps/lines/src/components/Game.svelte`) component with a “reveal” bookEvent of the bonus mode. It will spin the reels.
- ...





With all the stories above and the stories that created and customised by yourself, we are able to test the whole game, intermediate components and atomic components.

We are also able to test our game with a book, a sequence of bookEvents and a single bookEvent.

If each bookEvent is implemented well with emitterEvents and its story is resolved properly, the game is technically finished.

Task Breakdown

There is one single idea that is been applied across the whole carrot-game-sdk that is **Task Breakdown**.

To extend a bit more of the topic above, if an emitterEventHandler does too much work, then it is better we consider to split it into smaller emitterEventHandlers as a process of task-breakdown.

For example, “tumbleBoard” bookEvent is a fairly complicated bookEvent. Instead of having one “tumbleBoard” emitterEvent, we split it into “tumbleBoardInit”, “tumbleBoardExplode”, “tumbleBoardRemoveExploded”, “tumbleBoardSlideDown”.

This way we can implement a big and complicated emitterEvent step by step. More importantly, we can test the implementations one by one in storybook of `` `COMPONENTS/<Game>/emitterEvent` ``.

```
// bookEventHandlerMap.ts - Example of task-breakdown

{
  ...,
  tumbleBoard: async (bookEvent: BookEventOfType<'tumbleBoard'>) => {
    eventEmitter.broadcast({ type: 'tumbleBoardShow' });
    eventEmitter.broadcast({ type: 'tumbleBoardInit', addingBoard: bookEvent.newSymbols });
    await eventEmitter.broadcastAsync({
      type: 'tumbleBoardExplode',
      explodingPositions: bookEvent.explodingSymbols,
    });
    eventEmitter.broadcast({ type: 'tumbleBoardRemoveExploded' });
    await eventEmitter.broadcastAsync({ type: 'tumbleBoardSlideDown' });
    eventEmitter.broadcast({
      type: 'boardSettle',
      board: stateGameDerived
        .tumbleBoardCombined()
```

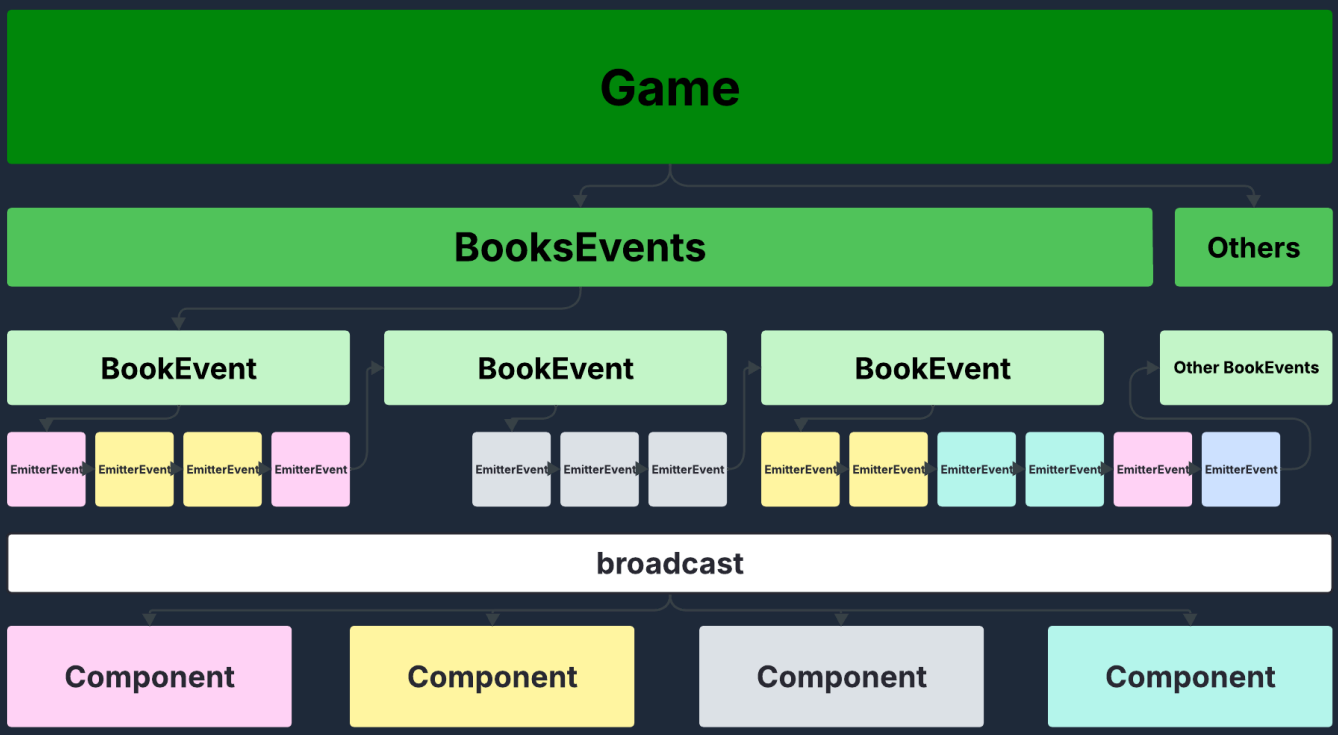
```
        .map((tumbleReel) => tumbleReel.map((tumbleSymbol) => tumbleSymbol.rawSymbol)),
    });
    eventEmitter.broadcast({ type: 'tumbleBoardReset' });
    eventEmitter.broadcast({ type: 'tumbleBoardHide' });
  },
  ...,
}
```

```
// TumbleBoard.svelte - Example of task-breakdown

context.eventEmitter.subscribeOnMount({
  tumbleBoardShow: () => {},
  tumbleBoardHide: () => {},
  tumbleBoardInit: () => {},
  tumbleBoardReset: () => {},
  tumbleBoardExplode: () => {},
  tumbleBoardRemoveExploded: () => {},
  tumbleBoardSlideDown: () => {},
});
```

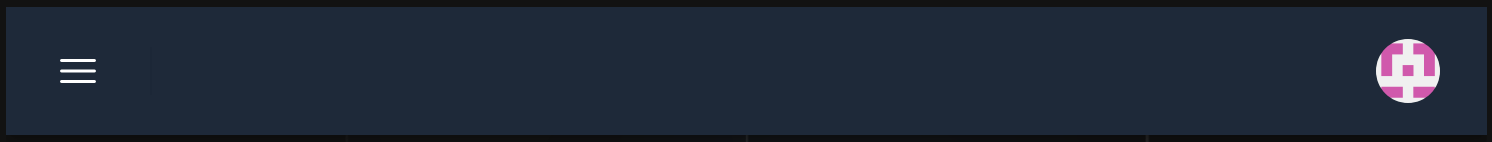
Stateless games can be complicated as well (vs. stateful games). For example, a slots game can have different types of spins, number of spins, win rules, number of bookEvents, game modes, global multiplier, multiplier symbols and so on.

- Stateless games: A single request to the RGS will finish the job of playing a game. For example, it requires only one request to play and finish a slots game.
- Stateful games: It requires multiple requests to the RGS to be able to finish the job. For example, a mines game.

However with the data structure of math and the functions we have, we are able to break down a complicated game into small and atomic tasks (emitterEvents). It enables us to test the atomics independently as well. Visually it is something like this:



The colors of the emitterEvents under a bookEvent can be different, which means they are from different svelte components.

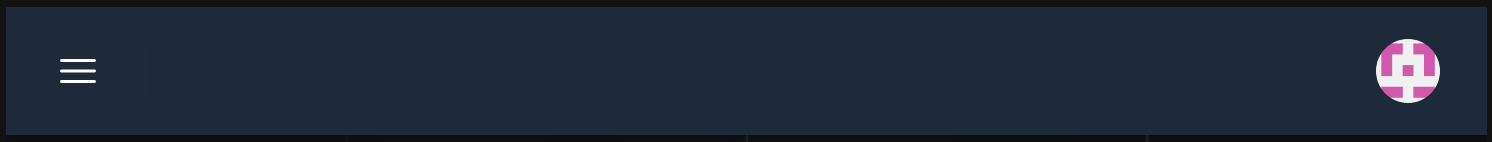


UI

We have provided solutions for the UI, which are `/packages/components-ui-pixi` and `/packages/components-ui-html`. They are functional with a few features like auto gaming, turbo mode, bonus button, responsiveness and so on, but they are not as beautiful.

```
<script lang="ts">
    import { UI, UiGameName } from 'components-ui-pixi';
    import { GameVersion, Modals } from 'components-ui-html';
</script>

<App>
  <UI>
    {#snippet gameName()}
      <UiGameName name="LINES GAME" />
    {/snippet}
    {#snippet logo()}
      <Text
        anchor={{ x: 1, y: 0 }}
        text="ADD YOUR LOGO"
        style={{
          fontFamily: 'proxima-nova',
          fontSize: REM * 1.5,
          fontWeight: '600',
          lineHeight: REM * 2,
          fill: 0xffffff,
        }}
      />
    {/snippet}
  </UI>
</App>

<Modals>
    {#snippet version()}
        <GameVersion version="0.0.0" />
    {/snippet}
</Modals>
```

For the branding purpose, we recommend you to regard them as just an example of UI packages instead of applying them directly to your final product. It would be a good choice to use them as a starting point and add more style to them to build your UI. It is completely fine to ignore them and build your own UI from scratch.



Why Use the Math SDK?

Traditionally, developing slot games involves navigating complex mathematical models to balance payouts, hit rates, and player engagement. This process can be time-consuming and resource-intensive. The Carrot Math SDK eliminates these challenges by providing:

- **Predefined Frameworks**: Start with customizable templates or sample games to accelerate development.
- **Mathematical Precision**: Simulate and optimize win distributions using discrete outcome probabilities, ensuring strict control over game mechanics.
- **Seamless Integration**: Outputs are formatted to align with the Carrot RGS, enabling quick deployment to production environments.
- **Scalability**: Built-in multithreading and optimization tools allow for efficient handling of large-scale simulations.

Who Is This For?

The Carrot Math SDK is ideal for developers looking to:

- Create custom slot games with unique mechanics.
- Optimize game payouts and hit rates without relying on extensive manual calculations.
- Generate detailed simulation outputs for statistical analysis.
- Publish games on Stake.com with minimal friction.

Static File Outputs

Physical slot-machines (and many of those used in iGaming) generate results in real time by programming game-logic onto the RGS/backend. When a game is requested, a

cryptographically secure random number generator selects a random reel-stop position for every active reel, and the game-logic flows from the starting board position. The drawbacks of this method is that since a single reel-strip could easily have 100+ symbols, with typically 5 reels, there are 100^5 (10 billion) unique board combinations.Explicitly calculating game payouts or Return to Player (RTP) is often infeasible, so extensive simulations are used to estimate outcomes. Stake Engine requires all game-outcomes to be known at the time of publication. Storing instructions for all possible game outcomes is impractical. Instead, a subset of results is used to define the game.

These outputs are broken up into two main parts: 1. game logic files and 2. CSV payout summaries. The game-logic files contain an ordered list of critical game details such as symbol names, board positions, payout amounts, winning symbol positions etc... Accompanying each simulation detailed in the game logic files is a CSV entry listing the simulation number, probability of selection, and payout amount. So upon a game round request, the RGS will consult the CSV/lookup table to select a simulation number, then return a JSON response from the game-logic file for this simulation number to the frontend, telling the web-client what to render, while also updating the players wallet with the payout amount. Breaking up these two files also allows us to exactly calculate the games RTP and essential win-distribution statistics at time of publication.

Get Started Today

Dive into the technical details and explore how the Carrot Math SDK can transform your game development process. With powerful tools, sample games, and detailed documentation, you'll have everything you need to create engaging and mathematically sound games.

See Math SDK Technical Details for more details.

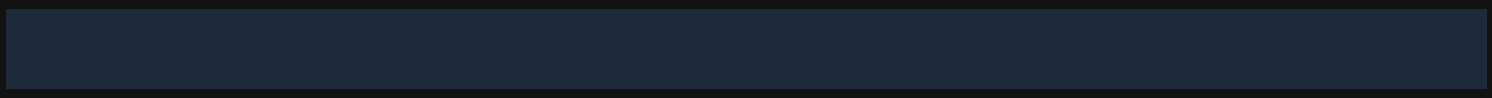
All valid bet-modes are defined in the array `self.bet_modes = [ ...]` The `BetMode` class is an important configuration for when setting up game the behavior of a game.This class is used to set maximum win amounts, RTP, bet cost, and distribution conditions. Additional noteworthy tags are:

1. `auto_close_disabled`
    - When this flag is `False` (default) the RGS endpoint API `/endround` is called automatically to close out the bet for efficiency. When the bet is closed however, the player cannot resume their bet. It may be desirable in bonus modes for example, to set this flag to `True` so that the player can resume interrupted play even if the payout is `0`. This means that the front-end will have to manually close out the bet in this instance.
2. `is_feature`
    - When this flag is true, it tells the frontend to preserve the current bet-mode without the need for player interaction. So if the player changes to `alt_mode` where this mode has `is_feature = True`, every time the spin/bet button is pressed, it will call the last selected bet-mode. Unlike in bonus games, where the player needs to confirm the bet-mode choice after each round completion.
3. `is_buybonus`
    - This is a flag used for the frontend framework to determine if the mode has been purchased directly (and hence may require a change in assets).

For example, the BetMode class for a bonus/buy feature is taken from the sample ***lines*** game:

```
BetMode(
    name="bonus",
    cost=100.0,
```

```
        rtp=self.rtp,
        max_win=self.wincap,
        auto_close_disabled=False,
        is_feature=False,
        is_buybonus=True,
        distributions=[
            Distribution(
                criteria="wincap",
                quota=0.001,
                win_criteria=self.wincap,
                conditions={
                    "reel_weights": {
                        self.basegame_type: {"BR0": 1},
                        self.freegame_type: {"FR0": 1, "WCAP": 5},
                    },
                    "mult_values": {
                        self.basegame_type: {1: 1},
                        self.freegame_type: {2: 10, 3: 20, 4: 50, 5: 60, 10: 100, 20: 90, 50: 50},
                    },
                    "scatter_triggers": {4: 1, 5: 2},
                    "force_wincap": True,
                    "force_freegame": True,
                },
            ),
            Distribution(
                criteria="freegame",
                quota=0.999,
                conditions={
                    "reel_weights": {
                        self.basegame_type: {"BR0": 1},
                        self.freegame_type: {"FR0": 1},
                    },
                    "scatter_triggers": {3: 20, 4: 10, 5: 2},
                    "mult_values": {
                        self.basegame_type: {1: 1},
                        self.freegame_type: {2: 100, 3: 80, 4: 50, 5: 20, 10: 10, 20: 5, 50: 1},
                    },
                    "force_wincap": False,
                    "force_freegame": True,
                },
            ),
        ],
    ),
```



Game Configuration Files

The GameState object requires certain parameters to be specified, and should be manually filled out for each new game. These elements are all defined in the `__init__` function. Full details of the expected inputs and data-types are given in the config info section.

General aspects of the game setup which should be considered when creating a `game_config.py` are:

Game-types

Several parts of the engine such as win amount verification, special symbol triggers/attributes and win-levels require the engine to know if the current state of the game is in the *basegame* or *freegame*. For example it is common to perform a weighted draw of some value:

```
#Within game config:
self.multiplier_values = {
   "basegame":{1:100, 2:50, 3: 10},
   "freegame":{2:20, 3:50, 5: 20, 10:10, 20:1}}
....
#Within gamestate:
multiplier = get_random_outcome(self.config.multiplier_values[self.gametype])
```

Typically special rules apply when the player enters a freegame. The configuration file allows the user to specify the key corresponding to each gametype. By default this is set to `basegame` and `freegame` respectively. All simulations will start in the basegame mode unless otherwise specified, and the transition to the freegame state is handled in the default `reset_fs_spin()` function, which is called as soon as the `run_freespin()` function is entered.

Reels

Most games will use distinct reelstrips for different game-types. It is commonplace for game-modes to have multiple possible reels per mode. One method of adjusting the overall RTP of a game is to have a multiple reelstrips with varying RTP, which can be selected from a weighted draw when calling `self.create_board_from_reelstrips()`. Reelstrips are stored as a dictionary in the `self.config.reels` object. The reelstrip key and csv file name should be specified:

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

Reelstrip weightings are required distribution conditions. An example of using multiple reelstrips for each gametype can be applied as:

```
conditions={
    "reel_weights": {self.basegame_type: {"BR0": 2, "BR1": 1}, self.freegame_type: {"FR0":5, "FR1": 1}},
},
```

Scatter triggers and Anticipation

Freegame entry from the basegame or retriggers in the freegame should be specified in the format `{num_scatters: num_spins}`,

```
self.freespin_triggers = {
    self.basegame_type: {3: 10, 4: 15, 5: 20},
    self.freegame_type: {2: 4, 3: 6, 4: 8, 5: 10},
}
```

Symbol initialization

A symbol is determined to be valid if the name exists either in `self.paytable` or in `self.special_symbols`. If a symbol that does not exist in either of these fields is detected when loading reelstrips, a `RuntimeError` is raised.

Symbol values

Winning symbols are determined from the `self.paytable` dictionary object in the game configuration. The expected format is:

```
self.paytable = {
    (kind[int], name[str]): value[float],
    ...
}
```

Where `kind` is the number of winning symbols. For cascading games, or other circumstances where multiple winning symbol numbers pay the same about, for example in the scatter pays example game where 13+ symbols pay the same amount, `self.pay_group` can be defined. By then calling `self.paytable = self.convert_range_table(pay_group)` a paytable of the expected format is generated. The format of the pay-group objects (inclusive of both values in the kind-range) is given as:

```
self.pay_group = {
    ((min_kind[int],max_kind[int]), name[str]): value[float],
    ...
}
```

Special symbols

Special symbol attributes are assigned based on names appearing in `self.special_symbols = {attribute[str]: [name[str], ...]}`. Multiple symbols can share attributes and multiple attributes can be applied to the same symbol. Most games will at least have a `wild` and `scatter` attribute. Once the symbol is initialized, the value of the attribute is accessed through `symbol.attribute` or symbol.get_attribute(attribute) see Symbols for more information regarding symbol object structures. By default the attribute is set to `True`, unless otherwise overridden using the `gamestate.special_symbol_functions`, defined in the gamestate override.

Distribution Conditions

Within each `BetMode` there is a set of `Distribution` Classes which determine the win-criteria within each bet-mode. Required fields are:

1. Criteria
    - A shorthand name describing the win condition in a single word
2. Quota
    - This is the amount of simulations (as a ratio of the total number of bet-mode simulation) which need to satisfy the corresponding criteria. The quota is normalized when assigning criteria to simulations, so the sum of all quotas does not need to be 1. There is a minimum of 1 simulation assigned per criteria.
3. Conditions
    - Conditions can have an arbitrary number of keys. Though the required keys are:
        - `reel_weights`
        - `force_wincap`
        - `force_freegame`

    Note that `force_wincap` and `force_freegame` are set to `False` by default and do not have to be explicitly added.

    The most common use for the Distribution Conditions is when drawing a random value using the BetMode's built-in method `get_distribution_conditions()`. i.e.

```
multiplier = get_random_outcome(betmode.get_distribution_conditions()['mult_values'])
```

Or to check if a board forcing the `freegame` should be drawn with:

```
if get_distribution_conditions()['force_freegame']:
    ...
```

4. Win criteria (optional)

   There is also a `win_criteria` condition which incorporates a payout multiplier into the simulation acceptance. The two commonly used conditions are `win_criteria = 0.0` and `win_criteria = self.wincap`. When calling `self.check_repeat()` at the end of a simulation, if `win_criteria` is not `None` (default), the final win amount must match the value passed.

   The intention behind betmode distribution conditions is to give the option to handle game actions in a way which depends on the (known) expected simulation. This is most clear if for example a simulation is known to correspond to a `max-win` scenario. Instead of repeated drawing random outcomes which are most likely to be rejected, we can alter the probabilities of larger payouts occurring by biasing a particular reelset, weighting larger prize or multiplier values etc..

Simulation Acceptance Criteria

When setting up the game configuration file each mode is split into different win-criteria. Given a total number of simulations for a given bet-mode, the number of simulations required for each criteria is set using a `quota`, which determines the ratio of the total number of simulations satisfying a particular win criteria.

Following the example used in the Sample Games, the win criteria has been split into the following unique conditions:

1. `0` win amounts
2. `basegame` wins
3. `freegame` scenarios
4. `max-win` scenarios

The purpose of segmenting these game outcomes is to ensure that there are sufficiently many simulations scenarios satisfying a certain criteria. For example if the hit-rate for a max-win is 1% of the available RTP for a game with a 5000x payout would be 1 in 500,000 outcomes. Though if we are only producing 1 Million simulations in total for this mode, we would like to have more than 2 simulations in total which result in the maximum win amount. This reduces the possibility of any players seeing the same outcomes for a specific win amount.

In the aforementioned list `0` dictates that the payout multiplier is ==0 for that simulation number. `basegame` is essentially any basegame spin where the payout is >0 and the `freegame` is not triggered. `freegame` is any scenario where the `freegame` is triggered from the basegame. `max-win` is any outcome where the maximum payout multiplier is awarded.

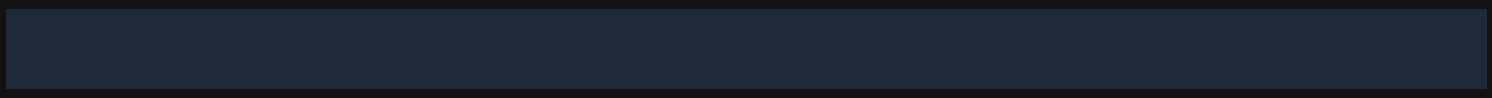
This segmentation of wins is also used by the optimization algorithm.

Pertinent to this section though, the simulation acceptance criteria is integral to the `repeat` condition implemented in all sample games. When the `GameState` is setup, the acceptance criteria is assigned to a specific simulation number before any simulations are carried out. So simulation 10, for example, is predetermined to be a simulation which triggers a `freegame`.

When the `run_spin()` function is called and the game-round ends, whether or not the simulation is recorded and added to the state overview is partially determined by the final win condition. If the only condition is that the simulation must be a `0` payout, then the `final_win` value is checked. If this condition is satisfied the `self.repeat = False` and the outcome is saved. Likewise if a particular simulation is determined to be `freegame` criteria, at the end of the spin we verify if the freegame has been triggered and accept the simulation result if so. There can be as many conditions are required in the `self.check_repeat()` function. Just be aware that the more stringent the criteria, the longer a simulation will likely take to run. This time can be quite substantial if the required criteria is unlikely to be achieved naturally. For the `max-win` scenarios for example, generally a specifically made reelstrip is used, and the probability if achieving higher multipliers, prizes etc.. is dictated in the bet-mode distribution.

Predetermining Acceptance

While it would be useful to run the simulations first and then assign the distribution criteria afterwards, this can cause issues when multi-threading larger simulation batches. Simulations relating to max-wins for example typically take substantially longer to succeed than say `0` win simulations. This means that all criteria except the max-win are likely to be filled first, leaving the final thread to deal with many or all of the max-win simulations. For this reason, the `quota` in the BetMode distribution conditions is used in conjunction with the total number of simulations.



Standard Game Setup Requirements

Without diving into specific functions, this section is intended to walkthrough how a new slot game would generally be setup. In practice it is recommended to start with one of the sample games which closest resemble the game being made, or otherwise starting from the template.

Configuration file

Game parameters should all be set in the `GameConfig` `__init__()` function. This is where to set the name name, RTP, board dimensions, payouts, reels and various special symbol actions. All required fields are listed in the `Config` class and should be filed out explicitly for each new game. Next the `BetMode` classes are defined. Generally there would be at a minimum a (default) `base` game and a `freegame`, which is usually purchased.

```
class GameConfig(Config):
    def __init__(self):
        super().__init__()
        self.game_id = ""
        self.provider_number = 0
        self.working_name = ""
        self.wincap = 0
        self.win_type = "lines"
        self.rtp = 0

        self.num_reels = 0
        self.num_rows = [0] * self.num_reels
        self.paytable = {
            (kind, symbol): payout,
        }

        self.include_padding = True
        self.special_symbols = {"property": ["sym_name"],...}
```

```
        self.freespin_triggers = {
        }
        self.reels = {}
        self.bet_modes = []
```

Each `BetMode` should likewise be set explicitly, defining the cost, rtp maximum win amounts and various gametype flags. We would like to define different win criteria within each betmode. In the sample games we define distinct criteria for any game-aspects where we would like to control either the hit-rate and/or RTP allocation. In this example we would like to control the basegame hit-rate, max-win hit-rate and freegame hit-rate. Therefore we need to specify unique `Distribution` criteria for each of these special conditions.

```
    BetMode(
        name="base",
        cost=1.0,
        rtp=self.rtp,
        max_win=self.wincap,
        auto_close_disabled=False,
        is_feature=True,
        is_buybonus=False,
        distributions=[
            Distribution(
                criteria="winCap",
                quota=0.001,
                win_criteria=self.wincap,
                conditions={
                    "reel_weights": {
                        self.basegame_type: {"BR0": 1},
                        self.freegame_type: {"FR0": 1},
                    },
                    "force_wincap": True,
                    "force_freegame": True,
                },
            ),
            Distribution(
                criteria="freegame",
                quota=0.1,
                conditions={
                    "reel_weights": {
                        self.basegame_type: {"BR0": 1},
                        self.freegame_type: {"FR0": 1},
                    },
                    "force_wincap": False,
                    "force_freegame": True,
```

```
                },
            ),
            Distribution(
                criteria="0",
                quota=0.4,
                win_criteria=0.0,
                conditions={
                    "reel_weights": {self.basegame_type: {"BR0": 1}},
                },
            ),
            Distribution(
                criteria="basegame",
                quota=0.5,
                conditions={
                    "reel_weights": {self.basegame_type: {"BR0": 1}},
                },
            ),
        ],
    )
```

Gamestate file

When any simulation is run, the entry point will be the `run_spin()` function, which lives in the `GameState` class. `GameExecutables` and `GameCalculations` are child classes of `GameState` and also deal with game specific logic.

The generic structure would follow the format:

```
def run_spin(self, sim):
    self.reset_seed(sim) #seed the RNG with the simulation number
    self.repeat = True
    while self.repeat:
        self.reset_book() #reset local variables
        self.draw_board() #rraw board from reelstrips

        #evaluate win_data
        #update win_manager
        #emit relevant events

        self.win_manager.update_gametype_wins(self.gametype) #update cumulative basegame wins
        if self.check_fs_condition(): #check scatter conditions
```

```
            self.run_freespin_from_base() #run freegame

        self.evaluate_finalwin()
        self.check_repeat() #Verify betmode distribution conditions are satisfied

    self.imprint_wins() #save simulation result
```

For reproducibility the RNG is seeded with the simulation number. Betmode distribution criteria are preassigned to each simulation number, requiring the `self.repeat` condition to be initially set until the spin has completed and it can be checked that any criteria-specific conditions or win amounts are satisfied. Note that `self.repeat = False` is set in the `self.reset_book()` function. This function will reset all relevant `GameState` properties to default values.

Generally the first steps will be to use the reelstrips provided in the configuration file to draw a board from randomly chosen reelstop positions. Wins are evaluated from one of the provided win-types for the active board, and the wallet manager is updated. After this game-logic is completed the relevant events (such as `reveal` and `winInfo`) are emitted. All sample games follow these three steps:

1. Calculate current state of the board
2. Update wallet manager
3. Emit events

To keep track of which gametype wins are allocated, the wallet manger is again invoked once all basegame actions are complete. If the game have a freegame mode and the triggering conditions are satisfied the `run_freespin()` function is invoked. This mode will have a similar structure:

```
def run_freespin(self):
    self.reset_fs_spin() #reset freegame variables
    while self.fs < self.tot_fs: #account for multiple freegame spins
        self.update_freespin() #update spin number and emit event
        self.draw_board() #draw a new board using freegame reelstrips

        #evaluate win_data
```

```
        #update win_manager
        #emit relevant events

        if self.check_fs_condition(): #check retrigger conditions
            self.update_fs_retrigger_amt()

        self.win_manager.update_gametype_wins(self.gametype) #update cumulative freegame win amounts

    self.end_freespin() #emit event to indicate end of freegame
```

While it is possible to perform all game actions within these functions, for clarity functions from `GameExecutables` and `GameCalculations` are typically invoked and should be created on a game-by-game basis depending on requirements.

Runfile

Finally to produce simulations, the `run.py` file is used to create simulation outputs and config files containing game and simulation details.

```
if __name__ == "__main__":

    num_threads = 1
    rust_threaeds = 20
    batching_size = 50000
    compression = False
    profiling = False

    num_sim_args = {
        "base": int(10),
        "bonus": int(10),
    }

    config = GameConfig()
    gamestate = GameState(config)

    create_books(
        gamestate,
        config,
        num_sim_args,
        batching_size,
```

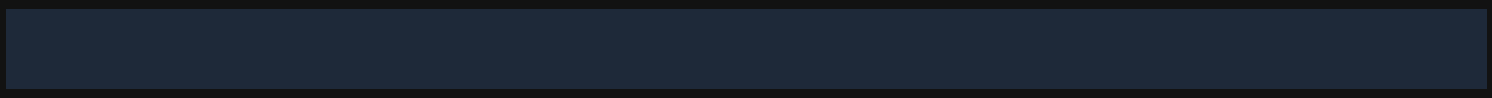
```
        num_threads,
        compression,
        profiling,
    )
    generate_configs(gamestate)
```

The `create_books` function handles the allocation of win criteria to simulation numbers, output file format and multi-threading parameters.

Outputs

Simulation outputs are placed in the `game/library/` folder. `books/books_compressed` is the primary data-file containing all events and payout multipliers. `lookup_tables` hold the summary simulation-payout values in `.csv` format which is consumed by the optimization algorithm. Additionally for game analysis, lookup table mapping of which simulations belong to which win criteria and which gametype wins arise from are produced. `force/` file outputs contain all information used by the `.record()` function, which is again useful for analyzing the frequency and average win amounts for specific events. The optimization algorithm also uses the recorded `force` data to identify which simulations correspond to specific win criteria. Finally `config/` files contain information required by the frontend such as symbol and betmode information, backend information such as file hash values and a configuration file for the optimization algorithm.

The optimization algorithm consumes the lookup table and outputs a copy of the file, but with modified weights. To assist with setting optimization parameters, there are two other files with the prefix `lookUpTableIdToCriteria` and `lookUpTableSegmented`. These files are used to identify which bet-mode sub-type that specific simulation number belongs to (such as max-wins, 0-wins, freegame entry etc..), and what gametype (usually basegame or freegame) contributes to the final payout multiplier.



Intended Engine Usage

Game Files

As seen in the example games, all games follow a recommended structure, which should be copied from the games/template folder.

```
game/
├── library/
|----- books/
|----- books_compressed/
|----- configs/
|----- forces/
|----- lookup_tables/
├── reels/
├── readme.txt
├── run.py
├── game_config.py
├── game_executables.py
├── game_calculations.py
├── game_events.py
├── game_override.py
└── gamestate.py
```

Sub-folders within library/ are automatically generated if they do not exist at the completion of the si

While all commonly used engine functions are handled by classes within their respective src/ directory,

The game_config/executables/calculations/events/override files offer extensions on actions defined in th

Run-file

This file is used to set simulation parameters, specifically the configuration and `GameState` classes.

Parameter	Type	Description
`num_threads`	`int`	Number of threads used for multithreading

```
| `rust_threads` | `int`        | Number of threads used by the Rust compiler |
| `batching_size`| `int`        | Number of simulations run on each thread |
| `compression`  | `bool`       | `True` for `.json.zst` compressed books, `False` for `.json` format |
| `profiling`    | `bool`       | `True` outputs and opens a `.svg` flame graph |
| `num_sim_args` | `dict[int]`  | Keys must match bet mode names in the game configuration |


All simulations are passed to the `create_books()` function which carries out all the simulations and ha

Once the simulations are completed, the **gamestate** is passed to `generate_configs(gamestate)` which h

## Library Folders

#### books/books_compressed
Depending on the **compression** tag passed to `create_books()` the `books/` or `books_compressed/` fold

#### configs
This will consist of three `.json` files for the math, frontend and backend.

#### lookup_tables
Once any given simulation is compete the events associated are stored within the books, and the correspo

| Simulation | Weight  | Payout |
|------------|---------|--------|
|   `int`    |  `int`  | `float`|

All simulations start with an assigned weight of `1`, which is then modified if the optimization algorit

### Configs

The **GameConfig** inherits the **Config** class. All information defined in the *__init__* function are

### Gamestate

Every game has a *gamestate.py* file, where independent simulation states are handled. The *run_spin()*

### Executables

Commonly used groups of game-logic and event emission is provided in this location. Functions called in

Functions currently in this class include drawing random or forced game-boards, handling game-logic for

### Misc. Calculations

The **Executables** class inherits all miscellaneous game-logic and board-actions. Primarily this includ
* Lines
* Ways
* Scatter (pay anywhere)
* Cluster
```

```
* Expanding wild + prize collection

Additionally other classes attached to **Executables** are tumbling/cascading of winning symbols and **C
```

The State Machine

Introduction

The **GameState** class serves as the central hub for managing all aspects of a simulation batch. It handles:

- Simulation parameters
- Game modes
- Configuration settings
- Simulation results
- Output files

The entry point for all game simulations is the `run.py` file, which initializes parameters through the config class and creates a GameState object. The **GameState** ensures consistency across simulations and provides a unified structure for managing game logic and outputs.

Key Responsibilities of `GameState`

Simulation Configuration

- Compression
- Tracing
- Multithreading
- Output files

- Cumulative win manager

Game Configuration

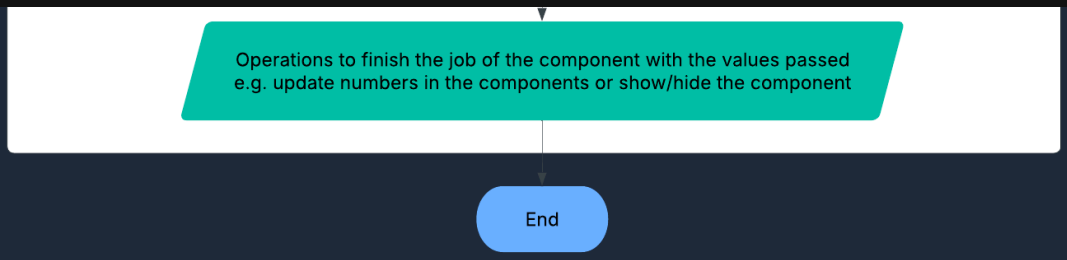
- Betmode details (costs, names, etc.)
- Paytable
- Symbols
- Reelsets

These **global** `GameState` **attributes** remain consistent across all game modes and simulations. When a simulation runs, the `run_spin()` method creates a sub-instance of the **GeneralGameState**, allowing modifications to game data directly through the `self` object. This design reduces the need for passing objects between functions, streamlining game logic development.

At a high level, the structure of the engine is shown below:

Extending Core

The `GameState` class acts as a super-class containing core functionality. Custom games can extend or override this functionality using Python's Method Resolution Order (MRO). Once simulations are complete, the relevant output files are generated sequentially for each BetMode. These outputs can then be optimized and uploaded to the Admin Control Panel (ACP).



Class Inheritance

Why Use Class Inheritance?

Class inheritance ensures flexibility, allowing developers to access core functions while customizing specific behaviors for each game. Core functions are defined in the source files and can be overridden at the game level.

GameStateOverride (game/game_override.py)

This class is the first in the **Method Resolution Order (MRO)** and is responsible for modifying or extending actions from the `state.py` file. For example, all sample games override the `reset_book()` function to accommodate game-specific parameters:

```
def reset_book(self):
    super().reset_book()
    self.reset_grid_mults()
```

```
    self.reset_grid_bool()
    self.tumble_win = 0
```

GameExecutables (game/game_executables.py)

This class groups commonly used game actions into executable functions. These functions can be overridden to introduce new mechanics at the game level. For example, triggering freespins based on scatter symbols:

```
config.freespin_triggers = {3: 8, 4: 10, 5: 12}

def update_freespin_amount(self, scatter_key: str = "scatter") -> None:
    self.tot_fs = self.config.freespin_triggers[self.gametype][self.count_special_symbols(scatter_key)]
    fs_trigger_event(self, basegame_trigger=True, freegame_trigger=False)
```

However in the `0_0_scatter` sample game, we would instead want to assign the total spins to be 2x the number of active Scatters. Therefore we can override the function in the `GameExecutables` class:

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

GameCalculations (games/game_calculations.py)

This class handles game-specific calculations, inheriting from **GameExecutables**.

Books and Libraries

What is a “Book”?

A “book” represents a single simulation result, storing:

- The payout multiplier

- Events triggered during the round
- Win conditions

Each simulation generates a Book object, which is stored in a library. The library is a collection of all books generated during a simulation batch. These books are attached to the global GameState object and are used for further analysis and optimization.

Example JSON structure:

```
[
    {
        "id": int,
        "payoutMultiplier": float,
        "events": [ {}, {}, {} ],
        "criteria": str,
        "baseGameWins": float,
        "freeGameWins": float
    }
]
```

Resetting the Book

At the start of a simulation, the book is reset to ensure a clean state:

```
def reset_book(self) -> None:
    self.book = {
        "id": self.sim + 1,
        "payoutMultiplier": 0.0,
        "events": [],
        "criteria": self.criteria,
    }
```

Lookup Tables

What are Lookup Tables?

Lookup tables provide a summary of all simulation payouts, offering a convenient way to calculate win distribution properties and Return To Player (RTP) values. Each table is stored as a CSV file and contains the following columns:

Simulation Number	Simulation Weight	Payout Multiplier
1	1	0.0
2	1	92.3
…	…	…

The **payoutMultipler** attached to a **book** represents the final amount paid to the player, inclusive or *basegame* and *freegame* wins. The **LookUpTable** *csv* file is a summary of all simulation payouts. This provides a convenient way to calculate win distribution properties and Return To Player calculations. All lookup tables will be of the format:

Purpose of Lookup Tables

- Win Distribution Analysis: Analyze payout distributions across simulations.
- RTP Calculation: Calculate the overall RTP for a game mode.
- Optimization: Serve as input for the optimization algorithm, which adjusts simulation weights to achieve desired payout characteristics.

File Naming Convention

- Initial Lookup Tables: lookUpTable_mode.csv
- Optimized Lookup Tables: lookUpTable_mode_0.csv

The optimization algorithm modifies the weight values in the lookup table, which are initially set to 1. These optimized tables are then used for further analysis or deployment.

Math verification

When uploading static math files to the RGS, Stake Engine will carry out preliminary checks to ensure ensure game-logic is of the expected format. The corresponding payout multipliers and probabilities are analyzed as a means of providing a quick summary of game statistics on the backend.

Minimum file requirements

For a game with one game-mode, there will be 3 files required for the Math to be published successfully.

- Index file (must be called ***index.json** and contain the mode name, cost multiplier and logic/CSV filenames)
- Lookup table (CSV file, with each line containing ID, Probability, Payout)
- Game logic (zStandard compressed JSON-lines (__.jsonl.zst))

Index file format

When selecting a directory to upload from for the Stake Engine math there must exist a JSON-encoded file called ***index.json*** with the strictly enforced form:

```
{
    "modes": [
        {
            "name": <string>,
```

```
            "cost": <float>,
            "events": <string>"<logic_file>.jsonl.zst",
            "weights": <string>"<lookup_table>.csv"
        },
        ...
    ]
}
```

For example, for a game with 2-modes:

```
{
    "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"
        }
    ]
}
```

CSV format

When calculating various statistical values on the RGS side, it is much more efficient and robust to work with unsigned integer values (since no payouts or probabilities will ever be negative). This avoids misinterpreting values due to rounding or floating-point errors. For every game-round uploded within the game-logic there must a summary CSV table containing rows of `uint64` values. We require the payoutMuliplier value in the third column to exactly match those provided in the game-logic file. There values are extracted and hashed to ensure identical `payoutMultiplier` values.

```
    simulation number, round probability, payout multiplier
```

For example:

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

Game logic format

Round information returned through the ***/play*** API corresponds to a single simulation outcome returned in JSON format. For efficiency, we require this data to be stored in compressed ***.jsonl*** format. Currently zStandard (.zst) encoding must be used, though this will be expanded upon in the near future. In order to identify simulation IDs, payouts and logic we enforce the condition that every simulation contains the key fields:

```
    "id": <int>,
    "events" <list<dict>>,
    "payoutMultiplier": <int>
```

For example, at a minimum the game round, printed to ***jsonl*** before compression will have the format:

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

Where the payoutMultiplier value corresponds to an 11.5x payout for a base game round (costing 1.0x). **The three JSON key fields: id, events, payoutMultipler are required for every**

round returned.

Running your first game

There are several example games provided within `/games/`, showing how common slot mechanics may be implemented. As an example let's look at `games/0_0_lines/`, a 3-row, 5-reel game paying on 20 win-lines. Wins involving 3 or more like symbols will award an amount described by `GameConfig.(paytable/payines)`.

Run-file

Simulation parameters including number of simulations, payout statistics, optimization conditions, which modes to run etc.. are all handled within `run.py`.

Using the default settings, running:

```
make run GAME=0_0_lines
```

(or calling the script manually after activating your virtual-environment)

```
python3 games/0_0_lines/run.py
```

will output all the files required by the RGS. All required files to publish math results are found within the `library/publish_files/` folder. Even if this math-sdk is not being used to generate math results, the *books*, *lookup-tables* and *index* file are required for publication.

Testing Game Outputs

To see example output files in human-readable form, lets simulate 100 results without compression in order to inspect the JSON output, we can alter the following variables within `run.py` :

```
num_threads = 1
compression = False

num_sim_args = {
    "base": 100,
    "bonus": 100,
}

run_conditions = {
    "run_sims": True,
    "run_optimization": False,
    "run_analysis": False
}
```

When setting `num_sim_args` , we are essentially running the function `run_spin()` within `gamestate.py` 100 times, with simulation criteria being assigned within the `GameConfig()` class. We can see which criteria (basegame, 0-wins, feature games, max-wins etc..) have been applied to which simulation within the `library/lookup_tables/lookUpTableIdToCriteria_<mode>.csv` file. Here, we will not run the optimization or analysis because 100 results does not give a large enough range of results to approach large-scale statistics. We only need 1 CPU thread, so we can change this from 10 to 1 since it should only a take a second or two to run. Inspecting the output file, `library/books/books_base.jsonl` shows each simulation, identified by `id` (1-100). Each simulation-id has an `events` tag, which communicates to the front-end framework which symbols are revealed, win positions and amounts, and any of game-specific logic. Each simulation has a `payoutMultiplier` value which is the final payout amount for that round. This value directly corresponds to the value in `library/lookup_tables/lookUpTable_base.csv` . When a round-response is returned from by the RGS from the `play/` API, it is the contents of the `events` tag which is returned in the response body.

If we look at the results for, say, simulation 58:

```
{
    "id": 58,
    "payoutMultiplier": 10,
    "events": [
        {
            "index": 0,
            "type": "reveal",
            "board":[...],
            "paddingPositions": [...],
            "gameType": "basegame",
            "anticipation": [...]
        },
        {
            "index": 1,
            "type": "winInfo",
            "totalWin": 10,
            "wins": [
                {
                    "symbol": "L5",
                    "kind": 3,
                    "win": 10,
                    "positions": [...],
                    "meta": {}
                }
            ]
        },
        {
            "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 tells up what board-symbols to reveal, the winning positions on this board, the payout amount, and sets the win counters. If we now open the lookup-table file and search for simulation number 58 we see the result: `58,1,10`, matching what was given to us within the `books` file. Note that all simulations are initially given a selection weight of `1` (the second value in each CSV row). The optimization program is what sets these weights to ensure that the game-mode is balanced to a specified RTP.

Larger simulation batches

When starting with a new game, it is suggested to start by running a small number of simulations saved in uncompressed JSON format for debugging. Once satisfied with the gamestate output, larger simulations should be run. For a production-ready game it is typically recommended to run 100k+ simulations per mode to ensure that there is a diverse range of payout multipliers to optimize over, and to significantly reduce the chances of any single player receiving the same round result more than once. We set the following parameters indicating that we want to use 20 threads for simulating the game-logic for 10,000 simulations per mode, output in compressed (.json.zst) format, we will then use 20 threads when running the optimization algorithm (this will produce modified lookup-tables such as `lookUpTable_base_0.csv`).

```
num_sim_args = {
    "base": int(1e4),
    "bonus": int(1e4),
}

run_conditions = {
    "run_sims": True,
    "run_optimization": True,
    "run_analysis": True,
    "upload_data": False,
}
```

In the terminal you should seethe game RTP printed out as each thread finishes

```
Thread 0 finished with 1.632 RTP. [baseGame: 0.043, freeGame: 1.588]
```

Flor the `bonus` mode, this is telling us that thread 0/10 finished with a total RTP of 163.2%, with 4.3% coming from the basegame (wins on the reveal of Scatter symbols), and 158.8% RTP coming from freegame wins. This is higher than our expected 97%, though we are forcing significantly more max-win simulations than will naturally be awarded, so this is okay. The optimization algorithm will adjust these weights to balance the game properly.

By setting `run_analysis: True` we are indicating that we would like to generate a PAR sheet, summarizing key game statistics and hit-rates. This program will use the `library/lookup_tables/lookUpTableSegmented_<mode>.csv` file to determine which game-types contributed to the final round wins, in conjunction with the pay-table and `library/forces/force_record_<mode>.json` files to generate frequency and average-win statistics for specific events or win combinations.

Next steps

These outputs corresponding directly with example Storybook packages within the `web-sdk`. It is recommended to take look through this pack to see how these math events are passed and displayed on the frontend. If you have your own game in mind you can use one of the sample games provided as a template and implement your own unique rules within the `games/<game_name>/` directory. You will likely need to specify configuration values for things like multipliers, prize-value etc.. wtihin `game_config.py`. Then any unique calculations and events should be handled within the games `game_executables/game_calculation` files. Generally speaking, reusable functions, events or calculation should like with `/src/`, which one-off game functionality belongs within that games folder `/games/<game_id>/`.

Repository Directory Overview

This repository is organized into several directories, each focusing on a specific aspect of the game creation process. Below is a breakdown of the main directories and their purposes:

Main Directories

- `games/`
  - Contains sample slot games showcasing widely used mechanics and modes:
    - `0_0_cluster`: Cascading cluster-wins game.
    - `0_0_lines`: Basic win-lines example game.
    - `0_0_ways`: Basic ways-wins example game.
    - `0_0_scatter`: Pay-anywhere cascading example game.
    - `0_0_expwilds`: Expanding Wild-reel game with an additional prize-collection feature.
- `src/`
  - Core game setup functions, game mechanics, frontend event structures, wallet management, and simulation output control. This directory contains reusable code shared across games. **Edit with caution.**
  - Subdirectories:
    - `calculations/`: Board and symbol setup, various win-type game logic.

    - `config/`: Generates configuration files required by the RGS, frontend, and optimization algorithm.
    - `events/`: Data structures passed between the math engine and frontend engine.
    - `executables/`: Commonly used groupings of game logic and events.
    - `state/`: Tracks the game state during simulations.
    - `wins/`: Wallet manager handling various win criteria.
    - `write_data/`: Handles writing simulation data, compression, and force files.
- `utils/`
    - Contains helpful functions for simulation and win-distribution analysis:
        - `analysis/`: Constructs and analyzes basic properties of win distributions.
        - `game_analytics/`: Uses recorded events, paytables, and lookup tables to generate hit-rate and simulation properties.
- `tests/`
    - Includes basic PyTest functions for verifying win calculations:
        - `win_calculations/`: Tests various win-mechanic functionality.
- `uploads/`
    - Handles the data upload process for connecting and uploading game files to an AWS S3 bucket for testing.
- `optimization_program/`
    - Contains an experimental genetic algorithm (written in Rust) for balancing discrete-outcome games.
- `docs/`
    - Documentation files written in Markdown.

Detailed Subdirectory Breakdown

`src/`

- `calculations/` : Handles board and symbol setup, along with various win-type game logic.
- `config/` : Creates configuration files required by the RGS, frontend, and optimization algorithm.
- `events/` : Defines data structures passed between the math engine and frontend engine.
- `executables/` : Groups commonly used game logic and events for reuse.
- `state/` : Tracks the game state during simulations.
- `wins/` : Manages wallet functionality and various win criteria.
- `write_data/` : Writes simulation data, handles compression, and generates force files.

`games/`

- `0_0_cluster/` : Sample cascading cluster-wins game.
- `0_0_lines/` : Basic win-lines example game.
- `0_0_ways/` : Basic ways-wins example game.
- `0_0_scatter/` : Pay-anywhere cascading example game.
- `0_0_expwilds/` : Expanding Wild-reel game with an additional prize-collection feature.

`utils/`

- `analysis/` : Constructs and analyzes basic properties of win distributions.
- `game_analytics/` : Generates hit-rate and simulation properties using recorded events, paytables, and lookup tables.

`tests/`

- `win_calculations/` : Tests various win-mechanic functionality.

`uploads/`

- Handles the process of uploading game files to an AWS S3 bucket for testing.

`optimization_program/`

- Experimental genetic algorithm (written in Rust) for balancing discrete-outcome games.

Setup and installation

Running the math-sdk requires Python3 and PIP to be installed!

Rust/Cargo must also be installed for the optimization algorithm to run!

Clone the Math SDK repository to get started

```
git@github.com:StakeEngine/math-sdk.git
```

Makefile (recommended)

Assuming Make and a recent version of Python3 is installed on your machine, the easiest method of setting up the SDK is using the terminal to invoke:

```
make setup
```

This will setup and activate a Python virtual environment, installing all necessary packages as defined within ***requirements.txt***, and install an editable math-sdk module.

Once the relavent parameters are set for a particular game, execute the run.py file using:

```
make run GAME=&lt;game_id&gt;
```

Installing Cargo (Only if using Optimization Algorithm)

If the optimization algorithm is being utilized, **Rust** and **Cargo** should be installed.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Manual installation

*Note: This installation is for Mac operating systems, Windows OS uses the prefix python (instead of python3)

Create and Activate a Virtual Environment

It's recommended to use a virtual environment to manage dependencies. Using the Virtual Environment manager (*venv*), install Python version >=3.12 using:

```
python3 -m venv env
```

If you are using Mac, activate the env with:

```
source env/bin/activate
```

If using a Windows computer use:

```
  env\Scripts\activate.bat
```

Install Dependencies

Use `pip` to install dependencies from the `requirements.txt` file:

```
python3 -m pip install -r requirements.txt
```

Install the Package in Editable Mode

Using the `setup.py` file, the package should be installed it in editable mode (for development purposes) with the command:

```
python3 -m pip install -e .
```

This allows modifications to the package source code to take effect without reinstallation.

Verify Installation

You can check that the package is installed by running:

```
python3 -m pip list
```

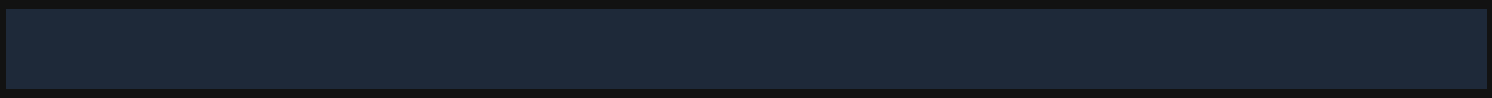
or testing the package import in Python:

```
python
>>> import your_package_name
```

Deactivating the Virtual Environment

When finished, deactivate the virtual environment with:

```
deactivate
```



Game Board

The `Board` class inherits the `GeneraGameState` class and handles the generation of game boards. Most commonly used is the `create_board_reelstrips()` function. Which selects a reelset as defined in the `BetMode.Distribution.conditions` class. For each reel a random stopping position is chosen with uniform probability on the range *[0,len(reelstrip[reel])-1]*. For each reelstop a 2D list of `Symbol` objects are created and attached to the GameState object.

Additionally, special symbol information is included (*special_symbols_on_board*) along with the reelstop values (*reel_positions*), padding symbols directly above and below the active board (*padding_positions*) and which reelstrip-id was used.

The is also an *anticipation* field which is used for adding a delay to reel reveals if the number of Scatters required for trigging the freegame is almost satisfied. This is an array of values initialized to `0` and counting upwards in `+1` value increments. For example if 3 Scatter symbols are needed to trigger the freegame and there are Scatters revealed on reels 0 and 1, the array would take the form (for a 5 reel game):

```
self.anticipation = [0, 0, 1, 2, 3]
```

If the selected reel_pos + the length of the board is greater than the total reelstrip length, the stopping position is wrapped around to the 0 index:

```
self.reelstrip[reel][(reel_pos - 1) % len(self.reelstrip[reel])]
```

The reelset used is drawn from the weighted possible reelstrips as defined in the `BetMode.betmode.distributions.conditions` class (and hence is a required field in the `BetMode` object):

```
    self.reelstrip_id = get_random_outcome(
        self.get_current_distribution_conditions()["reel_weights"][self.gametype]
    )
```

Specific stopping positions can also be forced given a reelstrip-id and integer stopping values from `force_board_from_reelstrips()`. If no integer value are provided for a reel, a random position is chosen. This function is typically used in conjunction with `executables.force_special_board`, which will search a reelstrip for a particular symbol name and randomly select a specified number of stopping positions, chosen to land on a randomly selected board row.

Additionally the `Board` class handled symbol generation, displaying the current `.board` in the terminal, and retrieving symbol positions and properties as defined in `config.special_symbols`.

Cluster Pays

Cluster games award wins when there are sufficiently many neighboring like-symbols. Neighbours must share the same reel or row, where diagonal connections do not count towards the cluster size. A minimum of 5 like-symbols is typical, though this can be defined in `GameConfig` class. Since it is possible for up to `num_rows * num_columns` winning symbols to occur, it is common to define a particular payout range. For example `5 kind` pays `p1` `6-7 kind` to pay `p2`, `8-10 kind` to pay `p3` and `12+` symbols pay `p4` etc... Instead of manually including all possible pay combinations in `config.paytable` there is a `convert_range_table()` function in the Config class which takes in a symbol range, name and payout amount which is used to generate all `config.paytable` entries. This pay group should be of the format:

```
paygroup = {
    ((min_combination[int], max_combination[int]), name[str]) : payout[float],
    ...
}
```

Ranges defined in `min_combination` and `max_combination` are inclusive, so for example if the `5-kind` payout for symbol `H1` pays `10x`, this would be written as: `((5,5),H1): 10`.

Often (though not always) cluster pays games include a tumbling mechanic. Within the cluster sample game for example, while there are still winning combinations, the board is tumbled, wins are evaluated for the new board, the wallet manager is updated and relevant events are emitted:

```
while self.win_data["totalWin"] > 0 and not (self.wincap_triggered):
    self.tumble_game_board()
    self.win_data = self.get_cluster_data(record_wins=True)
```

```
    self.win_manager.update_spinwin(self.win_data["totalWin"])
    self.emit_tumble_win_events()
```

Clusters are found using a Breath First Search (BFS) algorithm. Wild attributes can be set (`wild` is the default value). Wild symbols can contribute to multiple clusters, including those formed by different symbols.

Line wins evaluation

The `LinesWins` object evaluates winning symbol combinations for the current `self.board` state. Generally 3 or more consecutive symbols result in a win, though these specific combination numbers and payouts can be defined in:

```
config.paytable = {(kind[int], symbol[string]): payout[float]}
```

In order to identify winning lines, line arrays must be defined in:

```
config.paylines = {
    0: [0,0,0,0,0],
    1: [0,1,0,1,0],
        ...
    }
```

in the `.paylines` dictionary, the key is the line-index and the value is an array dictating which rows result in a winning combination. Like symbols are matched and if the key `(kind, name)` exists in `self.paytable`, the corresponding win is evaluated.

Custom keys used to identify **wild** attributes and symbol names can be explicitly set and will default to `"wild"` and `"W"` unless otherwise specified. In the case of `(kind, "W")` existing in `self.paytable`, the base payout value is checked against the `(kind, sym)` where *sym* is the first non-wild. If for example the payline `[0,0,0,0,0]` has the symbol combination `[W,W,W,L4,L4]`, resulting in wins `(3,"W")` or `(5,"L4")`. We compare both outcomes and determine that the three-kind Wild combination has a larger payout. Therefore we only take the first three symbols as the winning combination. Note that the sample lines calculation provided will only take into account the base-game wins. If the game is more complex, such as

having multipliers on symbols, the final payout amount may need to be handled separately when deciding which winning combination to use. One common approach to dealing with this is to only define the Wild symbols to pay when there is a complete line (so only 5-kind Wilds would pay for a board of this size).

The `get_lines()` evaluation function returns all win information including the winning symbol name, winning positions, number of consecutive matches and win amounts. The `meta` information also includes symbol and global multiplier information, as well as the index of winning lines as defined in `config.paylines = {index: [line], ... }`.

Scatter Pays

Scatter-pays (pay-anywhere) games award wins based on the total number of like-symbols appearing on the game board. Symbols do not have to be arranged in any order. Typically a minimum of 8 like-symbols (or Wilds) are required to count as a win, though these values can be defined in the `GameConfig` class. Since it is possible for up to `num_rows * num_columns` winning symbols to occur, it is common to define a particular payout range. For example `8-kind` pays `p1` `9-kind` to pay `p2`, `10-12` kind to pay `p3` and `12+` symbols pay `p4` etc... Instead of manually including all possible pay combinations in `config.paytable` there is a `convert_range_table()` function in the Config class which takes in a symbol range, name and payout amount which is used to generate all `config.paytable` entries. This pay group should be of the format:

```
paygroup = {
    ((min_combination[int], max_combination[int]), name[str]) : payout[float],
    ...
}
```

Ranges defined in `min_combination` and `max_combination` are inclusive, so for example if the `8-kind` payout for symbol `H1` pays `10x`, this would be written as: `((8,8),H1): 10`.

Often (though not always) Scatter pays games are also cascading/tumbling. Within the scatter sample game for example, while there are still winning combinations, the board is tumbled, wins are evaluated for the new board, the wallet manager is updated and relevant events are emitted:

```
while self.win_data["totalWin"] > 0 and not (self.wincap_triggered):
    self.tumble_game_board()
    self.win_data = self.get_scatterpay_wins(record_wins=True)
```

```
    self.win_manager.update_spinwin(self.win_data["totalWin"])
    self.emit_tumble_win_events()
```

The Scatter pay evaluation function also checks for `multiplier` and `wild` attributes attached to symbols. Wild symbols can contribute to wins for any number of symbols.

Tumbling boards

The `Tumble` class inherits `Board` and handles removing winning symbols from `self.board` and filling vacant positions with symbols which appear directly above winning positions using the properties `reel_positions` and `reelstrip_id`. Examples of applications surrounding tumbling (cascading) events can be found in the `0_0_cluster` and `0_0_scatter` sample games.

The win evaluation functions for the cluster and scatter win-types assign the property `explode = True` to winning symbol objects. A new board is select by scanning the current `self.board` object reel-by-reel and counting the number of symbols which satisfy `sym.check_attribute("explode")`. This same number of symbols is then appended, counting backwards from the initial `self.reel_positions` values. If padding symbols are used, the symbol stored in `top_symbols` will be used to fill the first vacated position.

Ways wins evaluation

The `WaysWins` object evaluates winning symbol combinations for the current `self.board` state. Generally 3 or more consecutive symbols result in a win, though these specific combination numbers and payouts can be defined in:

```
config.paytable = {(kind[int], symbol[string]): payout[float]}
```

The ways calculation will search for like-symbols (or Wilds) on consecutive reels. The maximum number of ways is determined from the board size: `max_ways = (num_rows)^(num_columns)`. Note: the ways calculation does not account for Wild symbols appearing on the first reel.

The Ways evaluation takes also takes into account multiplier values attached to symbols containing the `multiplier` attribute. Unlike lines calculations where multiplier values are added together for symbols on consecutive reels, the total number of ways is instead multiplied by the multiplier value. Leading to the payout amount to grow substantially more quickly. So for example given the board:

```
L5 H1 L4 L4 L4
L1 H4 L3 H2 L4
H1 H1 H1 L3 H3
```

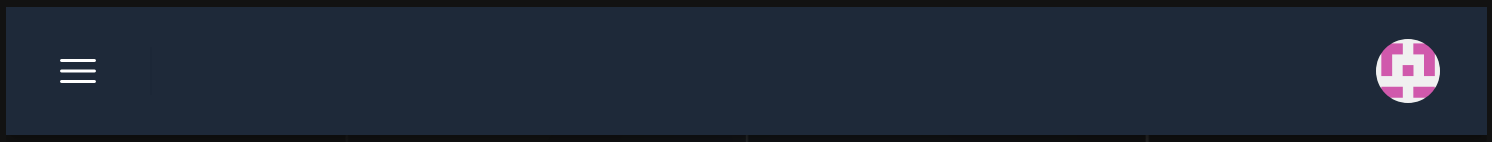
If there is a multiplier value of, say 3x on the `H1` symbol on reel 3, the total ways for symbol `H1` is `(3,H1)` pays:

```
(1) * (2) * (3) = 6 ways
```

The `return_data` will include all winning symbol names, number of consecutive like-symbols, winning positions and total win amounts for each unique symbol type. the `meta` tag will additionally include the total number of ways a symbol wins, which will range from `1` to `(num_rows)^(num_columns)` and and additional symbol and/or global multiplier contributions.

Config class object

The game-specific configuration `GameConfig` inherits the `Config` super class. This contains all game specifications, many of which will be set manually for each new game within `GameConfig`. `Config` allows for setting custom `win_levels`, which are returned during win-events and can indicate the type of animation which needs to be played. Additionally the class sets up several path destinations used for writing files and functions to read in and verify reelstrips stored in the `.csv` format.



Events Module Documentation

Overview

The `events.py` module defines reusable game events that modify the `gamestate` and log significant actions. These events ensure proper tracking of game states and facilitate structured client communication.

Functions

`json_ready_sym(symbol, special_attributes)`

Purpose: Converts a symbol object into a dictionary suitable for JSON serialization, including only specified attributes.

Parameters:

- `symbol (object)` : The symbol object to convert.
- `special_attributes (list)` : A list of attribute names to include if they are not `False` .

`reveal_event(gamestate)`

Purpose: Logs the initial board state, including padding symbols if enabled.

`fs_trigger_event(gamestate, include_padding_index, basegame_trigger, freegame_trigger)`

Purpose: Logs the triggering of free spins, whether from the base game or a retrigger event.

Assertions:

- Either `basegame_trigger` or `freegame_trigger` must be `True`, not both.
- `gamestate.tot_fs` must be greater than 0.

`set_win_event(gamestate, winlevel_key='standard')`

Purpose: Updates the cumulative win amount for a single outcome.

`set_total_event(gamestate)`

Purpose: Updates the total win amount for a betting round, including all free spins.

`set_tumble_event(gamestate)`

Purpose: Logs wins from consecutive tumbles.

`wincap_event(gamestate)`

Purpose: Emits an event when the maximum win amount is reached, stopping further spins.

`win_info_event(gamestate, include_padding_index=True)`

Purpose: Logs winning symbol positions and their win amounts, adjusting for padding if enabled.

`update_tumble_win_event(gamestate)`

Purpose: Updates the banner for tumble win amounts.

`update_freespin_event(gamestate)`

Purpose: Logs the current and total free spins remaining.

`freespin_end_event(gamestate, winlevel_key='endFeature')`

Purpose: Logs the end of a free spin feature and assigns the final win level.

`final_win_event(gamestate)`

Purpose: Logs the final payout multiplier at the end of a simulation.

`update_global_mult_event(gamestate)`

Purpose: Logs changes to the global multiplier.

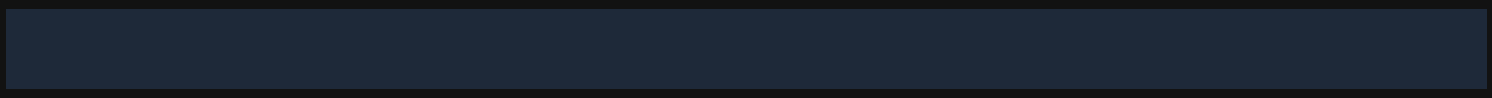
`tumble_board_event(gamestate)`

Purpose: Logs symbol positions removed during a tumble and their replacements.

Usage Notes

- Each function appends an event dictionary to `gamestate.book['events']`.
- Deep copies ensure that modifications do not affect past event states.
- Events provide structured output suitable for UI updates and analytics.

This module is essential for maintaining a transparent, trackable game state across different game mechanics.



Executables Class Documentation

Overview

The `Executables` class groups together common actions that are likely to be reused across multiple games. These functions can be overridden in `GameExecutables` or `GameCalculations` if game-specific alterations are required. Generally, `Executables` functions do not return values.

Function Descriptions

`draw_board(emit_event: bool = True) -> None`

Forces the initial reveal to have a specific number of scatters if bet mode criteria specify it. Otherwise, it generates a new board and ensures it does not contain more scatters than necessary.

`force_special_board(force_criteria: str, num_force_syms: int) -> None`

Forces a board to have a specified number of a particular symbol by modifying reel stops.

`get_syms_on_reel(reel_id: str, target_symbol: str) -> List[List]`

Returns reel stop positions for a specific symbol name.

`emit_wayswin_events() -> None`

Transmits win events associated with ways wins.

`emit_linewin_events() -> None`

Transmits win events associated with line wins.

`emit_tumble_win_events() -> None`

Transmits win and new board information upon a tumble event.

`tumble_game_board() -> None`

Removes winning symbols from the active board and replaces them, triggering a tumble board event.

`evaluate_wincap() -> None`

Checks if the running bet win has reached the wincap limit and stops further spin functions if necessary.

`count_special_symbols(special_sym_criteria: str) -> int`

Returns the number of active symbols of a specified special kind.

`check_fs_condition(scatter_key: str = "scatter") -> bool`

Checks if there are enough active scatters to trigger free spins.

`check_freespin_entry(scatter_key: str = "scatter") -> bool`

Ensures that the bet mode criteria are expecting a free spin trigger before proceeding.

`run_freespin_from_base(scatter_key: str = "scatter") -> None`

Triggers the free spin function and updates the total number of free spins available.

`update_freespin_amount(scatter_key: str = "scatter") -> None`

Sets the initial number of spins for a free game and transmits an event.

`update_fs_retrigger_amt(scatter_key: str = "scatter") -> None`

Updates the total number of free spins available when a retrigger occurs.

`update_freespin() -> None`

Called before a new reveal during free spins, resetting spin win data and other relevant attributes.

`end_freespin() -> None`

Transmits the total amount awarded during the free spin session.

`evaluate_finalwin() -> None`

Checks base and free spin sums, then sets the payout multiplier accordingly.

`update_global_mult() -> None`

Increments the multiplier value and emits the corresponding event.

Dependencies

This class relies on multiple external modules, including:

- `src.state.state_conditions.Conditions`
- `src.calculations.lines.LineWins`
- `src.calculations.cluster.ClusterWins`
- `src.calculations.scatter.ScatterWins`
- `src.calculations.ways.WaysWins`
- `src.calculations.tumble.Tumble`
- `src.calculations.statistics.get_random_outcome`
- `src.events.events` (Various event handling functions)

These modules provide necessary game logic, event management, and mathematical calculations for the execution of the class functions.

Usage

This class is designed as a base class and is expected to be extended by game-specific implementations where needed. It ensures core game mechanics, such as board generation, free spin handling, and win event management, are handled in a reusable manner.

Output files

All relevant output files are automatically generated within the `game/library/` directories. If the required sub-directories do not exist, the will be automatically generated.

Books

The primary data file output when simulations are run are the book files. These contain summary simulation information such as the final payout multiplier, basegame and freegame win contributions, the simulation criteria and simulation events. The contents of `book.events` is the information returned by the RGS `play/` API response.

The uncompressed `books/` files are used within the front-end testing framework and should be used to debug events. Only a small number of simulations should be run due to the file size. Compressed book files are what is uploaded to `AWS` and consumed by the RGS when games are being uploaded. Only data from compressed books will be returned from the `play/` API.

Force files

Each bet mode will output a file of the format `force_mode.json`. Every time the `.record()` function is called, the description keys used as input are appended to the file. If the key already exists, the `book-id` is appended to the array. This file is used to count instances of particular events. The optimization algorithm also makes use of these keys to identify max-win and freegame books. Once all bet mode simulations are finished, a `force.json` file is output which contains all the unique fields and keys.

Lookup tables

The final payout multiplier for each simulation is summarized in the `lookUpTable_mode.csv`. This is the file accessed by the optimization algorithm, which works by adjusting the weights, initially assigned to `1`. There is also a `IdToCriteria` file which indicates the win criteria required by a specific simulation number, and a `Segmented` file used to identify what gametype contributed to the final payout multiplier. Both these additional files are not typically uploaded to the ACP and are instead used for various analysis functions.

Config files

There are three config files generated after all simulations and optimizations are run. `config_math.json` is used by the optimization algorithm and contains all relevant bet mode details, RTP splits and optimization parameters. `config_fe.json` is used by the front-end frame work and contains symbol information, padding reels and bet mode details which need to be displayed to players. `config.json` contains bet mode information and file hash information and used used by the RGS to determine and verify changes to files being uploaded to the ACP.

File path construction

The `OutputFiles` class within `src/config/output_filenames` is used to construct filepaths and output filenames as well as setting up output folders if they do not yet exist.

GeneralGameState Class Overview

Class: `GeneralGameState`

Description:

The `GeneralGameState` class is an abstract base class (ABC) that defines the general structure for game states. Other game state classes inherit from it. It includes methods for initializing game configurations, resetting states, managing wins, and running simulations.

Constructor:

`__init__(self, config)`

- Initializes the game state with the provided configuration.
- Initializes variables like `library`, `recorded_events`, `special_symbol_functions`, `win_manager`, `criteria`, etc.
- Calls helper methods to reset seeds, create symbol mappings, reset book values, and assign special symbol functions.

Methods:

`create_symbol_map(self) -> None`

- Extracts all valid symbols from the configuration.

- Constructs a `SymbolStorage` object containing all the symbols from the paytable and special symbols.

`assign_special_sym_function(self)` (Abstract Method)

- This method must be overridden in derived classes to define custom symbol behavior.
- Issues a warning if no special symbol functions are defined.

`reset_book(self) -> None`

- Resets global game state variables such as `board`, `book_id`, `book`, and `win_data`.
- Initializes default values for win tracking and spin conditions.
- Resets `win_manager` state.

`reset_seed(self, sim: int = 0) -> None`

- Resets the random number generator seed based on the simulation number for reproducibility.

`reset_fs_spin(self) -> None`

- Resets the free spin game state when triggered.
- Updates `gametype` and resets spin wins in `win_manager`.

`get_betmode(self, mode_name) -> BetMode`

- Retrieves a bet mode configuration based on its name.
- Prints a warning if the bet mode is not found.

`get_current_betmode(self) -> object`

- Returns the current active bet mode.

`get_current_betmode_distributions(self) -> object`

- Retrieves the distribution information for the current bet mode based on the active criteria.
- Raises an error if criteria distribution is not found.

`get_current_distribution_conditions(self) -> dict`

- Returns the conditions required for the current criteria setup.
- Raises an error if bet mode conditions are missing.

`get_wincap_triggered(self) -> bool`

- Checks if a max-win cap has been reached, stopping further spin progress if triggered.

`in_criteria(self, *args) -> bool`

- Checks if the current win criteria match any of the given arguments.

`record(self, description: dict) -> None`

- Records specific game events to the `temp_wins` list for tracking distributions.

`check_force_keys(self, description) -> None`

- Verifies and adds unique force-key parameters to the bet mode configuration.

`combine(self, modes, betmode_name) -> None`

- Merges forced keys from multiple mode configurations into the target bet mode.

`imprint_wins(self) -> None`

- Records triggered events in the `library` and updates `win_manager`.

`update_final_win(self) -> None`

- Computes and verifies the final win amount across base and free games.
- Ensures that total wins do not exceed the win cap.
- Raises an assertion error if the sum of base and free game payouts mismatches the recorded final payout.

`check_repeat(self) -> None`

- Determines if a spin needs to be repeated based on criteria constraints.

`run_spin(self, sim)` (Abstract Method)

- Must be implemented in derived classes.
- Placeholder prints a message if not overridden.

`run_freespin(self)` (Abstract Method)

- Must be implemented in derived classes.
- Placeholder prints a message if not overridden.

`run_sims(self, betmode_copy_list, betmode, sim_to_criteria, total_threads, total_repeats, num_sims, thread_index, repeat_count, compress=True, write_event_list=True) -> None`

- Runs multiple simulations, setting up bet modes and criteria per simulation.
- Tracks and prints RTP calculations.
- Writes temporary JSON files for multi-threaded results.
- Generates lookup tables for criteria and payout distributions.

Summary

- `GeneralGameState` provides a foundation for defining and managing game states.
- It includes methods for configuring symbols, handling wins, recording events, and executing game simulations.
- Certain methods must be overridden in derived classes to customize behavior.

Wallet Manger

When a set of simulations are setup and executed through the `src/state/run_sims()` function, a new instance of the `WinManager` class is spawned. This class is responsible for tracking `basegame` and `freegame` wins for single simulation rounds (when running `run_spin()`), and also for cumulative win amounts for a given `BetMode`.

```
class WinManager:
    def __init__(self, base_game_mode, free_game_mode):
        self.base_game_mode = base_game_mode
        self.free_game_mode = free_game_mode

        self.total_cumulative_wins = 0
        self.cumulative_base_wins = 0
        self.cumulative_free_wins = 0

        self.running_bet_win = 0.0

        self.basegame_wins = 0.0
        self.freegame_wins = 0.0

        self.spin_win = 0.0
        self.tumble_win = 0.0
```

Cumulative wins

The cumulative win-amounts are useful in the terminal printouts to quickly check the RTP splits for a given multiprocessing thread. These cumulative values are updated each time a simulation is run and successfully passed, within `state.imprint_wins()` basegame and freegame win amounts are updated using `win_manager.update_end_round_wins()`.

`total_cumulative_wins` incorporate wins from all game-types on a single betmode level, while `cumulative_base_wins` and `cumulative_free_wins` track the cumulative win amounts for the

basegame and freegame respectively.

Spin-level wins

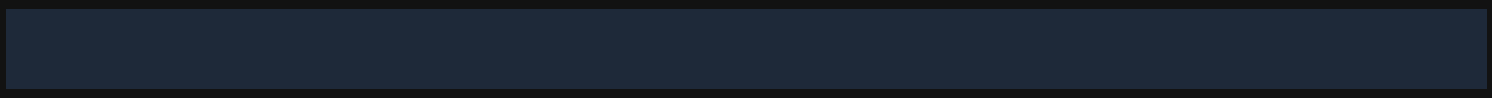
The `running_bet_win` tracks wins from the basegame and freegame modes and continuously increases during simulation steps. The final `running_bet_win` value will equal the payout multiplier `basegame_wins` and `freegame_wins` are single simulation level parameters which are reset when `run_spin()` is called. These values are subsequently used for the `lookUpTableSegmented` files, which helps to identify the contribution of different game-types to the final payout multiplier.

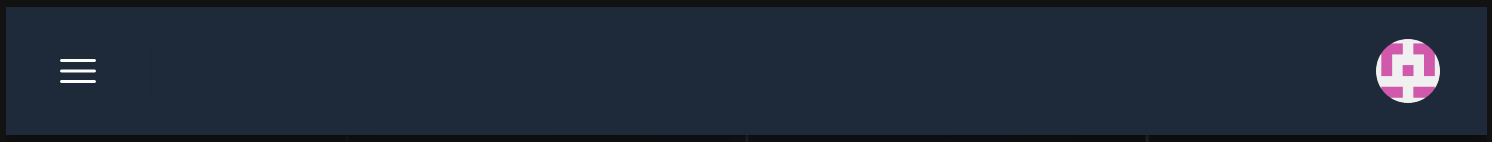
The `spin_win` property tracks the win for a given `reveal` event. So for example is reset for each spin within a `freegame`. Finally the `tumble_win` property is used for tracking wins where there are consecutive win events within a single reveal, most commonly seen within tumbling/cascading games. We may want to keep track of the cumulative win amount resulting from multiple tumble events to update win-banners or apply multipliers at the end of the sequence.

Update functions

There are several `WinManager` update functions used to update and reset the `spin_win` and gametype wins. The `running_bet_win` property does not need to be called explicitly, nor does the `cumulative_wins` (as this is called when the simulation is accepted and saved). The gametype should be updated explicitly though when the basegame actions have concluded, as well as at the end of each freegame spin (if applicable). This can be seen the sample `gamestate.run_spin()` game files:

```
self.win_manager.update_gametype_wins(self.gametype)
```





RGS Details

This specification outlines the API endpoints available to providers for communicating with the Stake Engine. These APIs enable key operations such as creating bets, completing bets, validating sessions, and retrieving player balances.

Introduction

This document defines how the provider's frontend communicates with the Stake Engine endpoints. It includes a detailed description of the core API functionality, along with the corresponding request and response structures.

The API facilitates communication between your game and the server. Each of the APIs will request the server to perform an action such as; authenticating a session, playing a round of a game and ending a round of a game. The APIs can be here.

Stake Engine NPM Client

Simplify communication to the RGS via the Stake Engine client. This package has helpers to streamline communication with the RGS.

Find information on the package and how to use it here.

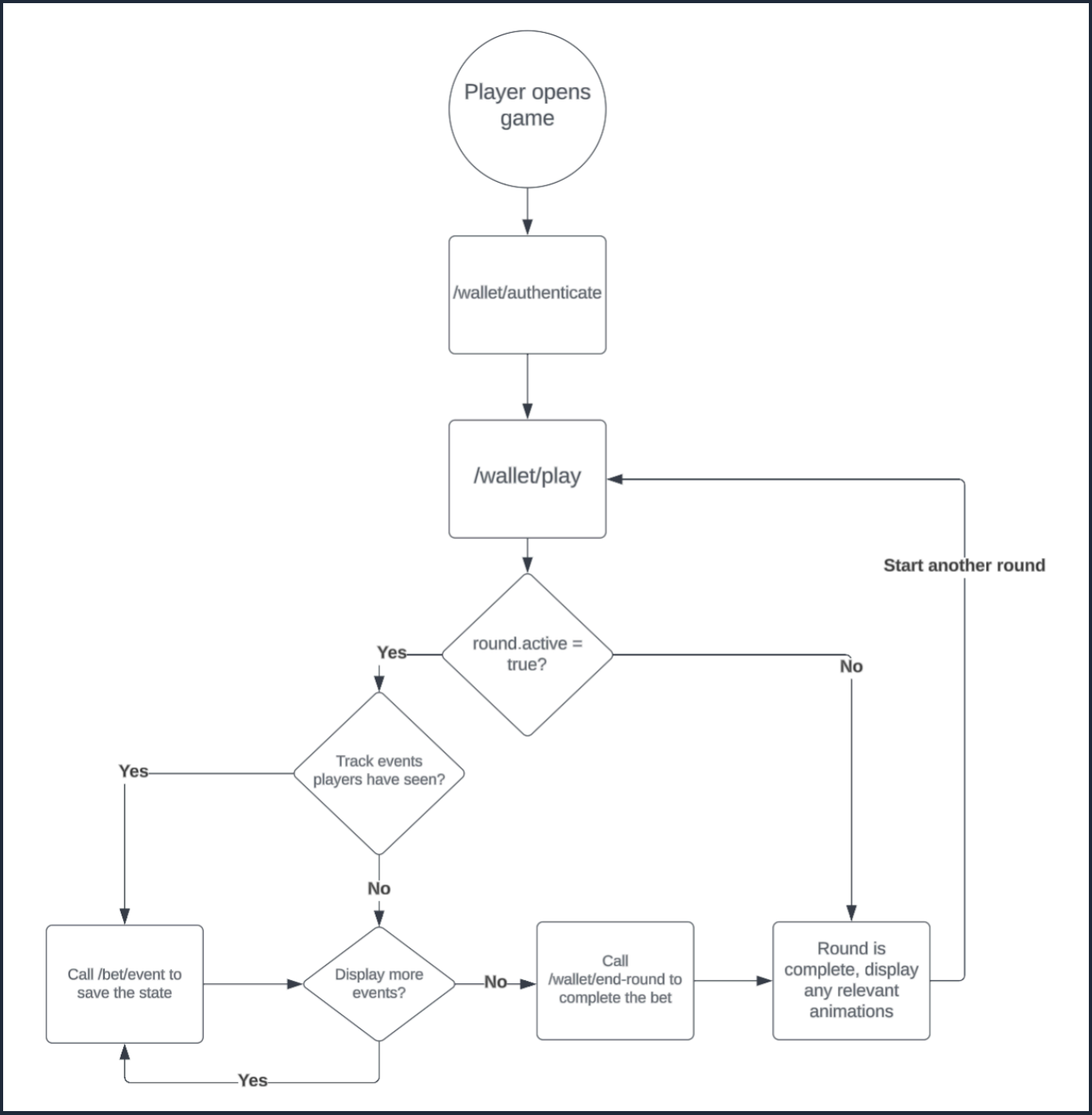
- https://github.com/StakeEngine/ts-client

API flows

All flows require the `/wallet/authenticate` API to be called when the game first loads. This authorizes the sessionID to be used by the `/wallet/play`, `/wallet/balance` and `/wallet/end-round` endpoints. If `/wallet/authenticate` endpoint has not been called with by the game, all subsequent APIs call will be returned with a 400 `ERR_IS` error as the session is invalid.

There the intended way to interact with the Stake Engine RGS API is described as a Basic Flow. This flow takes creates a round and will close the round after all animations have been complete. It accomplishes this by calling the `/wallet/play` API and then calling the `/wallet/end-round` API when the round is complete.

Basic flow



You will call `/wallet/play` and `/wallet/end-round` in a basic flow which is the simplest way to interact with the API. If you have a longer round that may include many steps (such as a bonus round in a slot game) you may want to save where the user is up to watching incase they disconnect. When they reload the game in the future, you can use the value found in `round.event` in the `/wallet/authenticate` API response to know where to display the animations for that round from.

URL Structure

Games are hosted under a predefined URL. Providers should use the parameters below to interact with the RGS on behalf of the user and correctly display game information.

```
https://{{.TeamName}}.cdn.stake-engine.com/{{.GameID}}/{{.GameVersion}}/index.html?sessionID={{.SessionI
```

Query Params in URL

Field	Description
sessionID	Unique session ID for the player. Required for all requests made by the game.
lang	Language in which the game will be displayed.
device	Specifies ‘mobile’ or ‘desktop’.
rgs_url	The URL used for authentication, placing bets, and completing rounds. This URL should not be hardcoded, as it may change dynamically.

Language

The `lang` parameter should be an ISO 639-1 language code.

Supported languages:

- `ar` (Arabic)
- `de` (German)
- `en` (English)

- `es` (Spanish)
- `fi` (Finnish)
- `fr` (French)
- `hi` (Hindi)
- `id` (Indonesian)
- `ja` (Japanese)
- `ko` (Korean)
- `pl` (Polish)
- `pt` (Portuguese)
- `ru` (Russian)
- `tr` (Turkish)
- `vi` (Vietnamese)
- `zh` (Chinese)

Understanding Money

Monetary values in the Stake Engine are integers with **six decimal places** of precision:

Value	Actual Amount
100,000	0.1
1,000,000	1
10,000,000	10
100,000,000	100

For example, to place a $1 bet, pass `"1000000"` as the amount.

Currency impacts **only** the display layer; it does not affect gameplay logic.

Supported Currencies

Currency	Abbreviation	Display	Example
United States Dollar	USD	$	$10.00
Canadian Dollar	CAD	CA$	CA$10.00
Japanese Yen	JPY	¥	¥10
Euro	EUR	€	€10.00
Russian Ruble	RUB	₽	₽10.00
Chinese Yuan	CNY	CN¥	CN¥10.00
Philippine Peso	PHP	₱	₱10.00
Indian Rupee	INR	₹	₹10.00
Indonesian Rupiah	IDR	Rp	Rp10
South Korean Won	KRW	₩	₩10
Brazilian Real	BRL	R$	R$10.00
Mexican Peso	MXN	MX$	MX$10.00
Danish Krone	DKK	KR	10.00 KR
Polish Złoty	PLN	zł	10.00 zł
Vietnamese Đồng	VND	₫	10 ₫
Turkish Lira	TRY	₺	₺10.00
Chilean Peso	CLP	CLP	10 CLP
Argentine Peso	ARS	ARS	10.00 ARS
Peruvian Sol	PEN	S/	S/10.00

Currency	Abbreviation	Display	Example
Nigerian Naira	NGN	₦	₦10.00
Saudi Arabia Riyal	SAR	SAR	10.00 SAR
Israel Shekel	ILS	ILS	10.00 ILS
United Arab Emirates Dirham	AED	AED	10.00 AED
Taiwan New Dollar	TWD	NT$	NT$10.00
Norway Krone	NOK	kr	kr10.00
Kuwaiti Dinar	KWD	KD	KD10.00
Jordanian Dinar	JOD	JD	JD10.00
Costa Rica Colon	CRC	₡	₡10.00
Tunisian Dinar	TND	TND	10.00 TND
Singapore Dollar	SGD	SG$	SG$10.00
Malaysia Ringgit	MYR	RM	RM10.00
Oman Rial	OMR	OMR	10.00 OMR
Qatar Riyal	QAR	QAR	10.00 QAR
Bahraini Dinar	BHD	BD	BD10.00
Stake Gold Coin	XGC	GC	10.00 GC
Stake Cash	XSC	SC	10.00 SC

Here are some functions that will help you achieve the display format for the currencies.

```
/**
 * Available currency codes for Stake Engine
 */
type Currency =
  | 'USD' // (United States Dollar)
  | 'CAD' // (Canadian Dollar)
```

```
  | 'JPY' // (Japanese Yen)
  | 'EUR' // (Euro)
  | 'RUB' // (Russian Ruble)
  | 'CNY' // (Chinese Yuan)
  | 'PHP' // (Philippine Peso)
  | 'INR' // (Indian Rupee)
  | 'IDR' // (Indonesian Rupiah)
  | 'KRW' // (South Korean Won)
  | 'BRL' // (Brazilian Real)
  | 'MXN' // (Mexican Peso)
  | 'DKK' // (Danish Krone)
  | 'PLN' // (Polish Złoty)
  | 'VND' // (Vietnamese Đồng)
  | 'TRY' // (Turkish Lira)
  | 'CLP' // (Chilean Peso)
  | 'ARS' // (Argentine Peso)
  | 'PEN' // (Peruvian Sol)
  | 'XGC' // Stake US Gold Coin
  | 'XSC'; // Stake US Stake Cash

/**
 * Currency metadata: symbol, default decimals, symbol placement
 *
 */
const CurrencyMeta: Record<
  Currency,
  { symbol: string; decimals: number; symbolAfter?: boolean }
> = {
  USD: { symbol: '$', decimals: 2 },
  CAD: { symbol: 'CA$', decimals: 2 },
  JPY: { symbol: '¥', decimals: 0 },
  EUR: { symbol: '€', decimals: 2 },
  RUB: { symbol: '₽', decimals: 2 },
  CNY: { symbol: 'CN¥', decimals: 2 },
  PHP: { symbol: '₱', decimals: 2 },
  INR: { symbol: '₹', decimals: 2 },
  IDR: { symbol: 'Rp', decimals: 0 },
  KRW: { symbol: '₩', decimals: 0 },
  BRL: { symbol: 'R$', decimals: 2 },
  MXN: { symbol: 'MX$', decimals: 2 },
  DKK: { symbol: 'KR', decimals: 2, symbolAfter: true },
  PLN: { symbol: 'zł', decimals: 2, symbolAfter: true },
  VND: { symbol: '₫', decimals: 0, symbolAfter: true },
  TRY: { symbol: '₺', decimals: 2 },
  CLP: { symbol: 'CLP', decimals: 0, symbolAfter: true },
  ARS: { symbol: 'ARS', decimals: 2, symbolAfter: true },
  PEN: { symbol: 'S/', decimals: 2, symbolAfter: true },
  XGC: { symbol: 'GC', decimals: 2 },
  XSC: { symbol: 'SC', decimals: 2 },
```

```
};

/**
 * Formats a number with its currency symbol, respecting default decimals and symbol placement.
 * The function is intended to be used for displaying balances.
 */
function DisplayBalance(balance: Balance): string {
  // Grabs the currency, if it doesn't exist in the list then it will display
  // the currency code behind the balance value.
  const meta = CurrencyMeta[balance.currency] ?? {
    symbol: balance.currency,
    decimals: 2,
    symbolAfter: true,
  };
  const formattedAmount = balance.amount.toFixed(meta.decimals);

  if (meta.symbolAfter) {
    return `${formattedAmount} ${meta.symbol}`;
  } else {
    return `${meta.symbol}${formattedAmount}`;
  }
}
```

Social Casino Currencies

- XGC (Gold)
- XSC (Stake Cash)

Bet Levels

Although bet levels are not mandatory, bets must satisfy these conditions:

1. The bet must fall between `minBet` and `maxBet` (returned from `/wallet/authenticate`).
2. The bet must be divisible by `stepBet`.

It is recommended to use the predefined `betLevels` to guide players.

Example:

```
{
  "minBet": 100000,
  "maxBet": 1000000000,
  "stepBet": 10000,
  "betLevels": [
    100000, // $0.10
    200000,
    400000,
    600000,
    ...
    1000000000 // $1000
  ]
}
```

Bet Modes / Cost Multipliers

Games may have multiple bet modes defined in the game configuration. Refer to the Math SDK Documentation.

When making a play request:

```
Player debit amount = Base bet amount × Bet mode cost multiplier
```

Response Codes

Stake Engine uses standard HTTP response codes (200, 400, 500) with specific error codes.

400 – Client Errors

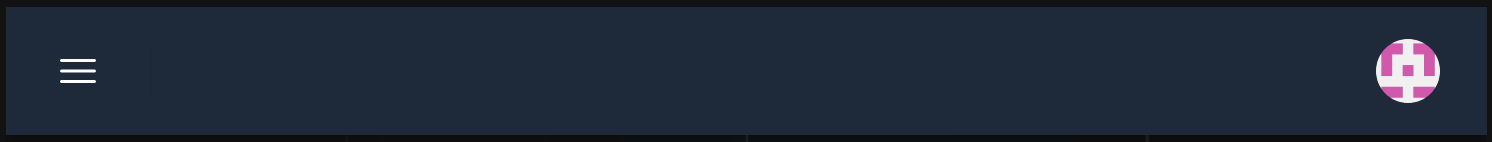
Status Code	Description
ERR_VAL	Invalid Request
ERR_IPB	Insufficient Player Balance
ERR_IS	Invalid Session Token / Session Timeout
ERR_ATE	Failed User Authentication / Token Expired
ERR_GLE	Gambling Limits Exceeded
ERR_LOC	Invalid Player Location

500 – Server Errors

Status Code	Description
ERR_GEN	General Server Error
ERR_MAINTENANCE	RGS Under Planned Maintenance

Math Publication File Formats

When publishing math results, ensure that the file-format is abided by. These are strict conditions for successful math file publication.



Getting Started with RGS Responses

This brief tutorial is intended to get you up and running with the RGS using a simple game called *fifty-fifty*.

Game Overview

The rules are straightforward:

- You request a response from the RGS's `/play` API.
- You have a 50/50 chance of either:
  - **2x** your bet back
  - Losing your **1x** bet.

Your **balance** is displayed alongside the outcome of the previously completed round. The JSON response for each round is shown on the right-hand side of the screen.

If your **win is greater than 0**, you'll need to manually call the `/end-round` API to finalize the bet—just like in a custom frontend implementation.

> *"For more information, see RGS Technical Details"*

Simple Math Results

Navigate to the `math-sdk/games/fifty_fifty/` directory and execute the `run.py` script. This will generate:

- A **Zstandard-compressed** set of simulation results
- A **lookup table** matching each result to its simulation
- The required `index.json` file

All necessary files to publish the game to the **Stake Engine** will be placed in `library/publish_files/`.

Simple Frontend Implementation

We'll use **Svelte 5** bundled with **Vite** to create a static frontend. We'll initialize the project using **Node Package Manager (NPM)** and optionally **Node Version Manager (NVM)**.

> *"**Note:** This guide assumes you are using NPM version `v22.16.0`."*

Setup Steps

1. **Create the Vite project**: `npm create vite@latest`
2. **Edit the `vite.config.ts` file**: Make sure the defineConfig function includes: `base: "./"` (under *plugins*),
3. **Replace styles and main component**:

- Copy the contents of `css.txt` into your generated `app.css`
- Replace the contents of `app_svelte.txt` into: `src/App.svelte`

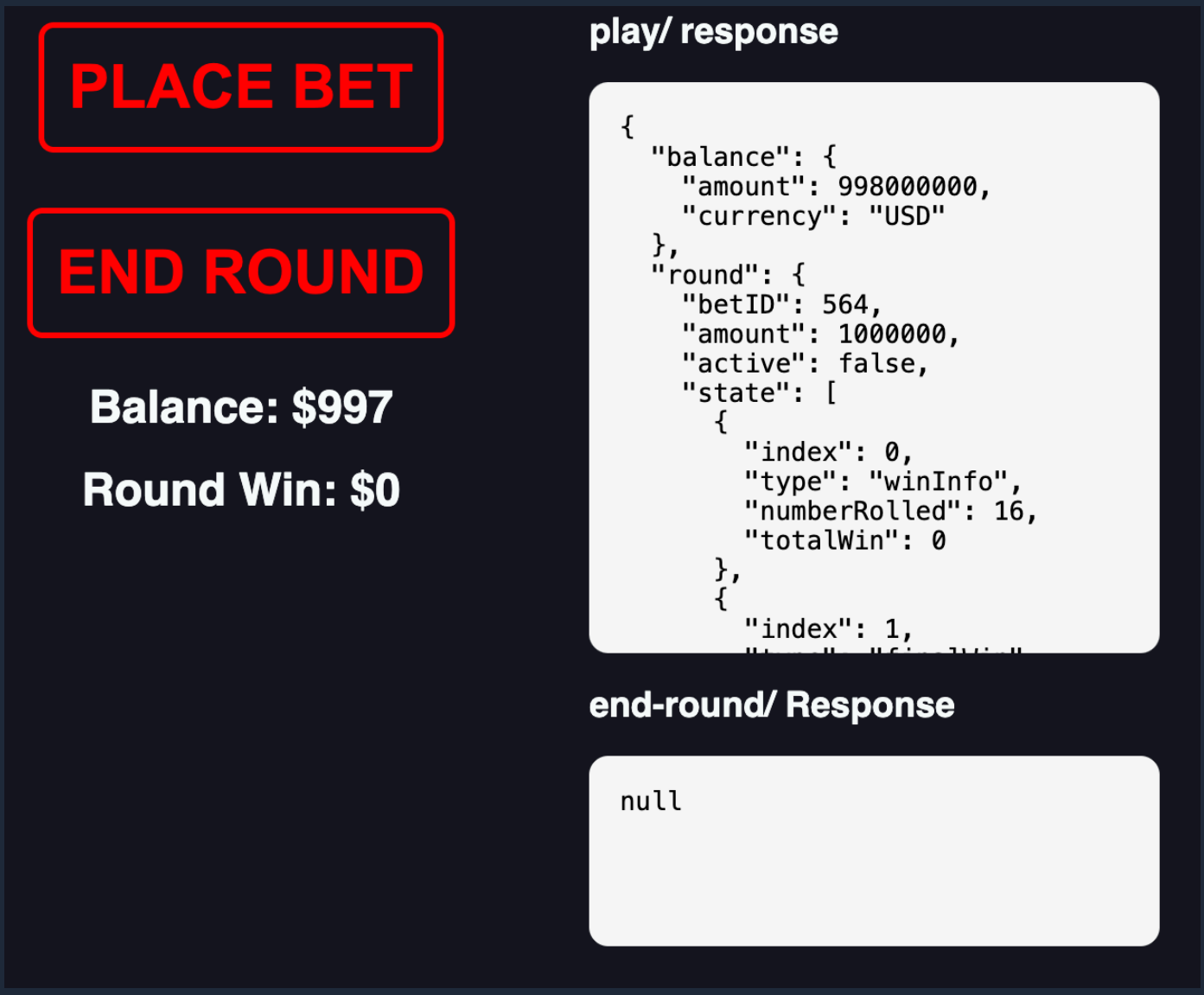
4. **Build the project**: `yarn build`
5. **Deploy**:
    - Upload the contents of the `dist/` folder to the Stake Engine under *frontend files*

What This Frontend Does

This simple Svelte app will:

- Authenticate your session with the RGS
- Request a response from the `/play` API
- (If applicable) Call the `/end-round` API to finalize a win

Once the math/frontend files have been uploaded to Stake Engine, launcing the game should result in the following:



Pressing ***Place BET*** will populate the *play/ response* field with the RGS game round structure. If the round-win is >0, press ***END ROUND*** to finalise the bet, which will subsequently update your balance and close the bet.

Wallet

The wallet endpoints enable interactions between the RGS and the Operator's Wallet API, managing the player's session and balance operations.

Authenticate Request

Validates a `sessionID` with the operator. This must be called before using other wallet endpoints. Otherwise, they will throw `ERR_IS` (invalid session).

Round

The `round` returned may represent a currently active or the last completed round. Frontends should continue the round if it remains active.

Request

```
POST /wallet/authenticate
```

```
{
  "sessionID": "xxxxxxx",
}
```

Response

```
{
  "balance": {
    "amount": 100000,
```

```
    "currency": "USD"
  },
  "config": {
    "minBet": 100000,
    "maxBet": 1000000000,
    "stepBet": 100000,
    "defaultBetLevel": 1000000,
    "betLevels": [...],
    "jurisdiction": {
      "socialCasino": false,
      "disabledFullscreen": false,
      "disabledTurbo": false,
      ...
    }
  },
  "round": { ... }
}
```

Balance Request

Retrieves the player's current balance. Useful for periodic balance updates.

Request

```
POST /wallet/balance
```

```
{
  "sessionID": "xxxxxx"
}
```

Response

```
{
  "balance": {
    "amount": 100000,
    "currency": "USD"
  }
}
```

Play Request

Initiates a game round and debits the bet amount from the player’s balance.

Request

```
{
  "amount": 100000,
  "sessionID": "xxxxxxx",
  "mode": "BASE"
}
```

Response

```
{
  "balance": {
    "amount": 100000,
    "currency": "USD"
  },
  "round": { ... }
}
```

End Round Request

Completes a round, triggering a payout and ending all activity for that round.

Request

```
POST /wallet/end-round
```

```
{
  "sessionID": "xxxxxx"
```

```
}
```

Response

```
{
  "balance": {
    "amount": 100000,
    "currency": "USD"
  }
}
```

Game Play

Event

Tracks in-progress player actions during a round. Useful for resuming gameplay if a player disconnects.

Request

```
POST /bet/event
```

```
{
  "sessionID": "xxxxxx",
  "event": "xxxxxx"
}
```

Response

```
{
  "event": "xxxxxx"
}
```

Response Codes

Stake Engine uses standard HTTP response codes (200, 400, 500) with specific error codes.

400 – Client Errors

Status Code	Description
ERR_VAL	Invalid Request
ERR_IPB	Insufficient Player Balance
ERR_IS	Invalid Session Token / Session Timeout
ERR_ATE	Failed User Authentication / Token Expired
ERR_GLE	Gambling Limits Exceeded
ERR_LOC	Invalid Player Location

500 – Server Errors

Status Code	Description
ERR_GEN	General Server Error
ERR_MAINTENANCE	RGS Under Planned Maintenance

Math Publication File Formats

When publishing math results, ensure that the file-format is abided by. These are strict conditions for successful math file publication.

Game Licence Terms and Conditions

Published Aug 6, 2025

This Game Licence Terms and Conditions ("**Agreement**") has been made and entered into on the Effective Date.

Parties:

A. Carrot Gaming Pty Ltd, a limited liability company, incorporated and existing under the laws of Australia, with corporate registration number 677 182 553 and having its registered address at 2/287-293 Collins Street, Melbourne, Australia ("**Carrot**"); and

B. The individual or incorporated entity who has completed Carrot's online registration and identity verification process, and whose details are identified and verified by Carrot through the identity documents and information submitted as part of the registration process and on or around the time acceptance of this Agreement ("**Developer**")

Individually referred to as a "**Party**" and collectively referred to as "**Parties**".

Key Definitions:

"Effective Date"	**The date on which the Developer indicates their acceptance of this Agreement by ticking the checkbox.**
"Term"	36 (thirty-six) months from the Effective Date ("**Initial Term**") and thereafter shall automatically renew for successive periods of 36 (thirty six) months ("**Renewal Term(s)**"), unless either Party provides the other with written notice of its intention not to renew at least three (3) months prior to the end of the Term or otherwise in accordance with this Agreement. Reference to "**Term**" in this Agreement means the Initial Term and any Renewal Term(s).
"Licence Fee"	Subject to the Negative Rollover mechanism in clause 8.1(e), the licence fee payable by Carrot and / or Nominee to the Developer in consideration for the licence of the Intellectual Property Rights in the Game to Carrot, being 10% of the Gross Gaming Revenue generated via the Game and activated on the Website, calculated on a monthly basis by Carrot and / or its Affiliates from the Commencement

"Effective Date"	**The date on which the Developer indicates their acceptance of this Agreement by ticking the checkbox.**
	Date.For the avoidance of doubt, where the Developer successfully secures activation of multiple Games, for the purpose of calculating the License Fee, the GGR from all such Games shall be aggregated and assessed on a combined basis.The Licence Fee is payable in US Dollars Tether (USDT) or any other currency nominated by Carrot from time to time.The Licence Fee is inclusive of Value Added Tax (VAT), sales taxes, and other duties, fees, excises or tariffs and withholding taxes, where applicable. Such VAT, sales taxes and possible other charges, duties, fees, excises or tariffs or withholding taxes incurred or arising as a result of this Agreement shall be the responsibility of the Developer.
"Game"	Each game or games created by the Developer, including all associated Intellectual Property Rights in each Game, including in the Game Name, its graphics, animation, story line, avatars and the software underlying the operation of the Game, which the Developer uploads to the ICT Infrastructure from time to time pursuant to this Agreement, including the New Versions of each Game.
"Website"	Any website or websites owned or operated by Carrot and / or Stake from time to time, whether known or unknown on the Effective Date. For the avoidance of doubt, reference to "Website" in this Agreement includes reference to all "Websites" in relation to which the Game is being assessed, integrated and / or activated pursuant to this Agreement.

1. Recitals

1.1. Carrot is a provider of online gaming solutions for the gaming industry.

1.2. Carrot's Affiliates, provide and operate the Websites.

1.3. The Developer designs and develops the Game(s).

1.4. The Developer wishes to license all Intellectual Property Rights in the Game to Carrot and / or its Nominee, in return for which the Developer will be entitled to use the Website to build and develop its Games and/or receive the Licence Fee from Carrot.

1.5. The Parties acknowledge that the Developer may have registered an account on the ICT Infrastructure, uploaded and / or obtained activation of Game(s) prior to the Effective Date, and it is the mutual understanding and agreement that, from the Effective Date, all such Developer accounts and all Games, including any Intellectual Property Rights therein, existing or activated prior to the Effective Date, shall be fully governed by and subject to the terms and conditions of this Agreement.

1.6. THEREFORE, in consideration of the mutual covenants, agreements, conditions and terms set forth herein, and for valuable consideration, the receipt and sufficiency of which are hereby acknowledged, the Parties agree as follows.

2. General Definitions

2.1. In this Agreement, unless the context otherwise requires, capitalised expressions shall have the meanings set out in this clause 2.1 or in Part B of the Front Sheet.

| **"Affiliate"** | means any legal entity that:i. directly or indirectly owns or controls the Party;ii. is under the same direct or indirect ownership or control as the Party; oriii. is directly or indirectly owned or controlled by the Party, for so long as such ownership or control lasts.Ownership or control shall exist through direct or indirect ownership of fifty percent (50%) or more of the nominal value of the issued equity share capital or of fifty percent (50%) or more of the shares entitling the holders to vote for the election of the members of the board of directors or persons performing similar functions. |

| **"Agreement"** | means this "Game Licence Agreement", together with the Front Sheet and all its Exhibits and any and all amendments, appendices and other documents thereto, which may be agreed upon from time to time between the Parties. |

| **"Applicable Law"** | means any domestic or foreign statute, law, common law, ordinance, binding policy, binding guidance, rule, administrative interpretation, regulation, order, writ, injunction, directive, judgement, decree, permit or other requirement that applies to any of the Parties. |

| **"Business Day"** | means any day other than a Saturday, Sunday, or public holiday in England. |

| **"Claim"** | has the meaning given to it in clause 10.1.b). |

| **"Commencement Date"** | means the date when the Game will be activated on the Website, to be nominated by Carrot (at its absolute discretion) in accordance with clause 4 of this Agreement and advised to the Developer. |

| **"Competitor"** | means another service provider or operator of a website that competes, whether directly or indirectly, with the Website (to be determined by Carrot, acting reasonably). |

| **"Confidential Information"** | means all commercial, financial, marketing, technical or other information of a secret or confidential nature (including trade secrets, Intellectual Property Rights, Developer Data, know-how) relating to a Party and disclosed (regardless of whether the said material or information is disclosed in writing, verbally or by any other means) by such Party to the other Party whether before, on, or after the Effective Date and includes, but is not limited to:a. all information relating to the Carrot and its Affiliates, including the processes, concepts, and ideas behind the their business;b. all information relating to the terms and conditions set out in this Agreement, including any commercial information;c. all information relating to the administrative, financial, or operational arrangements of a Disclosing Party which is of a secret or proprietary nature or is otherwise expressly stated by that Party to be confidential;d. all technical and non-technical information, data, drawings, experience, trade secrets, and know-how relating to the business affairs, products, services, customers, and strategies of a disclosing Party, which is directly or indirectly disclosed to a receiving Party before or after coming into force of this Agreement, whether in

writing, orally or electronically, including, without limitation, information or data relating to a disclosing Party's products, systems, ideas, software, design methodology, evaluation methodology and criteria, manufacturing processes and related equipment, suppliers, customers, business plans, strategies, and financial situation, and any notes, memoranda, summaries, analyses, compilations or any other writings relating thereto;e. any data or information of any description relating to Players or their activities on their accounts in connection with the use of the Game, including personal and private information of Players; andf. all analyses, compilations, studies and other documents prepared by or on behalf of a Disclosing Party and of its employees or advisors. |

| **"Data Protection Legislation"** | means all applicable data protection and privacy legislation in force from time to time in the UK including without limitation the UK GDPR and the Data Protection Act 2018 or any successor legislation. |

| **"Derivative Works"** | means (a) for copyrightable or copyrighted material: a work that is based upon one or more pre-existing works, such as a revision, modification, translation, abridgment, condensation, expansion, collection, compilation or any other form in which such a pre-existing work may be recast, transformed or adapted, and that, if prepared without authorization by the owner of the pre-existing work, would constitute copyright infringement, (b) for patentable or patented material: any adaptation, addition, improvement, or combination based on a pre-existing work, (c) for material subject to trade secret or protection or confidentiality obligations: any new material, information, or data relating to and derived from such existing trade secret material or Confidential Information, including new material which may be protected by copyright, patent, trade secret or other proprietary rights and/or (d) any work or material which is derived from or makes use of any other Intellectual Property Rights. |

| **"Developer Data"** | means all data and/or information developed, received or acquired by Carrot under this Agreement and which originates from the Developer and its operations, Players, employees, assets and programs in whatever form such data and/or information may exist, including, but not limited to, all data related to play of the Game by Players. |

| **"Developer Materials"** | means all materials, including, but not limited to, Developer Data that the Developer provides under this Agreement. |

| **"Disabling Devices"** | means undisclosed software viruses, time bombs, logic bombs, trojan horses, trap doors, back doors, or other computer instructions, intentional devices or techniques that are designed to threaten, infect, assault, vandalize, defraud, disrupt, damage, disable, maliciously encumber, hack into, incapacitate, infiltrate or slow or shut down a computer system or any component of such computer system, including any such device affecting system security or compromising or disclosing Player data in an unauthorized manner. |

| **"Disclosing Party"** | means the Party disclosing Confidential Information to the Receiving Party, including to any Affiliate of the Receiving Party. |

| **"Dispute"** | has the meaning given to it in clause 16.1. |

| **"Error"** | means an error or a defect that affects the proper functioning of the Game (including any bug). |

| **"Force Majeure Event"** | means an event, whether or not foreseen, thati. is beyond the reasonable control of one Party and is not due to the fault or negligence of such Party; andg. could not have been avoided by such Party's exercise of due diligence, that prevents that Party from complying with any of its obligations under this Agreement, including, but not limited to, an act of God (such as fire, explosion, earthquake, drought, tidal wave and floods), war, hostilities, invasion, civil war, acts or threats of terrorism, change of legislation, breakdown or failure of computer systems or the internet, hacking or attacks on computer systems or data, loss of or interruption of power supplies or communications facilities, strike or lock-out, epidemics and pandemics. |

| **"Game Documentation"** | has the meaning given to it in clause 4.13.a). |

| **"Game Name"** | means, in the event the Developer has named the Game:i. the name of the Game; andii. any trade mark or logo (whether containing the name of the Game or not) which exists and which is used by the Developer in relation to the Game as at the Effective Date and any applications or registrations of the same from time to time. |

| **"Game Rights"** | means the Intellectual Property Rights in or associated with the Game (including its underlying software and the object code and Source Code therein), Game Name, Developer Materials and/or Game Documentation. |

| **"GGR" or "Gross Gaming Revenue"** | means the aggregate value of the wagers placed on the Game by the Players, less the total aggregate winnings as a result of those wagers, calculated on a monthly basis. |

| **"Good Industry Practice"** | means that degree of professionalism, skill, diligence, prudence and foresight which would be expected from a skilled and experienced recognized service provider of similar stature as the Parties hereto under similar circumstances. |

| **"ICT Infrastructure"** | means the information and communication technology platform and / or website used by Carrot for the purposes of enabling the upload and activation of the Game pursuant to this Agreement. |

| **"Intellectual Property Rights"** | means patents, utility models, rights to inventions, trademarks, service marks, trade names, logos, domain names, business names, rights in get-up, goodwill and the right to sue for passing off or unfair competition, rights in designs (including registered designs and design rights), copyright and related rights (including rights in computer software such as source code and object code, and all other works or material recorded or embodied in the software, including the audio or visual content in any screen-displays in the user interface and moral rights), database rights, rights to preserve the confidentiality of information, rights in know-how, trade secrets and all other intellectual property rights, in each case whether registered or unregistered and including applications for grant of any of the foregoing and all rights or forms of protection having equivalent or similar effect to any of the foregoing which may subsist anywhere in the world now or in the future together with all (a) rights to the grant of and applications for the same, (b)

corresponding applications, re-issues, renewals, extensions, divisions and continuations of the aforesaid, and (c) all similar and analogous rights in any country or jurisdiction. |

| **"IPR Sale"** | has the meaning given to it in clause 13.1. |

| **"New Game"** | means any games developed by or on behalf of the Developer similar to or consistent with the Game. |

| **"New Version"** | means a new version or new release of all or any part of the Game developed by or on behalf of the Developer in which previously identified faults have been remedied or to which any modification, enhancement, revision or update has been made, or to which a further function or functions have been added. |

| **"Nominee"** | means an Affiliate which Carrot nominates to receive any rights arising from this Agreement. |

| **"Player"** | means a player who accesses the Games through the Website. |

| **"Prohibited Content"** | means advertising or content that: (a) promotes pornographic material or is lewd, profane, obscene, unlawful; (b) is defamatory, libellous, discriminatory or constitutes "hate speech"; (c) infringes the rights (including the Intellectual Property Rights) of third parties; and/or (d) incites or encourages racism. |

| **"Receiving Party"** | means the Party receiving Confidential Information from the Disclosing Party, including from any Affiliate of the Disclosing Party. |

| **"Regulatory Authority"** | means any governmental, semi-governmental, administrative, fiscal, judicial, or quasi-judicial body, department, commission, authority, tribunal or agency having authority over the services provided pursuant to this Agreement. |

| **"Relevant Game Rights"** | has the meaning given to it in clause 13.2. |

| **"Restrictive Open Source Code"** | has the meaning given to it in clause 6.b)(vi). |

| **"ROFO Notice"** | has the meaning given to it in clause 13.2. |

| **"ROFO Offer"** | has the meaning given to it in clause 13.3. |

| **"ROFO Offer Period"** | has the meaning given to it in clause 13.3. |

| **"ROFO Waiver"** | has the meaning given to it in clause 13.3. |

| **"Territory"** | means worldwide. |

| **"Source Code"** | in relation to any software, means such software in eye-readable form and in such form that it can be compiled or interpreted into equivalent object code, together with all technical information,

documentation, instructions and further information reasonably required to build such software and/or for the use, reproduction, modification and enhancement of such software. |

| **"Stake"** | refers to the Carrot Affiliate owning and / or operating the Websites. |

2.2. In this Agreement:

```
a. the headings in this Agreement do not affect its interpretation;

b. reference to a gender includes the other gender and the neuter;

c. references to a "person" include an individual, a body corporate, association or partnership and includes a reference

d. unless the context otherwise requires, a clause is a reference to a clause of the Agreement;
```

e. references to "writing" include typing, printing, lithography, photography, display on a screen, electronic and facsimile transmission and other modes of representing or reproducing words in a visible form, and expressions referring to writing shall be construed accordingly;

f. references to the singular include the plural and vice versa;

g. any words that follow 'include', 'includes' or 'including', 'in particular' or any similar words and expressions shall be construed as illustrative only and shall not limit the sense of any word, phrase, term, definition or description preceding those words; and

h. a reference to legislation is a reference to that legislation, as amended from time to time.

3. Upload of the Game

3.1. The Parties agree that the Game will be uploaded to the ICT Infrastructure for the purpose of testing and assessment by Carrot according to the criteria and standards determined by Carrot from time to time (at its absolute discretion).

3.2. Once the Game is uploaded to the ICT Infrastructure, Carrot will assess the Game and Carrot, at its absolute discretion and with no liability to the Developer whatsoever, will either:

a. reject the Game; or

b. accept the Game by notice in writing to the Developer, in which case the Game will be considered for integration and activation pursuant to clause 4 of this Agreement.

3.3. The Parties acknowledge and agree that:

a. a Game is only deemed accepted upon the provision of the written notice referred to above;

b. Carrot reserves the right (at absolute discretion) to reject the Game at any time after upload but prior to acceptance; and

c. if a Game is not accepted then the payment obligation in respect of any Licence Fees for that Game does not arise.

3.4. For the avoidance of doubt, the acceptance process referred to in this clause applies to each individual Game uploaded to the ICT Infrastructure by the Developer.

3.5. The Parties acknowledge that there are multiple Websites on which the Game could be activated once it satisfies all the requirements in this Agreement. At the point of upload to the ICT Infrastructure, the Developer may be notified regarding which Website the Game is being tested and assessed in relation to.

3.6. **Access to the development environment**

a. For the purpose of assessing the ongoing development, nature, and progress of the Game and any New Games, the Developer acknowledges that Carrot and/or its Nominee will have continuous, real-time access to the Developer's development platform or environment where the Game and any New Games are being developed.

b. This access shall allow Carrot and/or its Nominee to view the state of development, the underlying characteristics of the Game and any New Games, and ensure alignment with Carrot's quality standards and strategic objectives, at Carrot's sole discretion.

c.

4. Licence of the Game Rights

4.1. Once the Game has been accepted by Carrot pursuant to clause 3.2b), then this clause 4 applies.

4.2. Within three (3) days of the successful acceptance by Carrot pursuant to clause 3.2b), to the extent it has not already done so, the Developer shall deliver to Carrot one copy of the object code for the Game, as it exists as of that date. During the term of this Agreement, the Developer shall provide to Carrot any updated object code within ten (10) days after completion of any modifications or revisions to the Game or creation of any New Versions.

4.3. Within (3) days of any written request by Carrot, the Developer shall deliver to Carrot a copy of the Source Code for the Game, as it exists as of that date.

4.4. In consideration of the Licence Fee paid by Carrot or its Affiliates to the Developer, the Developer hereby grants to Carrot and / or Nominee an exclusive (subject to Clause 4.10), irrevocable, perpetual (subject to

Clause 14.2.d)(i)) licence under the Game Rights to do the following acts in the Territory:

a. use the Game (including without limitation the Game Name), Developer Materials and Game Documentation;

b. install, integrate, host and use the Game on the Website and/or permit Stake or a third party hosting provider to do so;

c. permit Players to access and play the Game on the Website;

d. advertise, market and promote the Game;

e. carry out customer support and care, fraud screening and payment management in relation to the Game;

f. to access, modify and use the Source Code or object code to the extent necessary to exercise Carrot's rights under this clause 4.4 and to develop, modify, alter, update and maintain the Game (either itself or through an Affiliate or third party) including as required under clauses 14.2.d)(ii) and 14.5; and

g. to use, reproduce, modify and exploit the Game Rights to the extent required for the purpose of exploiting Carrot's Intellectual Property Rights in any developments, modifications, updates and/or alterations to the Game (as set out in clause 4.12).

4.5. Notwithstanding the generality of clause 4.4, the Developer acknowledges that Carrot has an exclusive, irrevocable and perpetual right (but not an obligation) to use the Game Name in relation to the Game. At its sole discretion, Carrot may choose to change a Game's existing Game Name, or in the event a Game has not been named by the Developer, Carrot may choose a name for the Game.

4.6. The parties acknowledge and agree that Carrot shall own all Intellectual Property Rights in any name (and any associated branding) Carrot chooses for a Game in accordance with clause 4.5.

4.7. The Developer shall not use the Game Name (or any name adopted by Carrot in accordance with clause 4.5) or anything confusingly similar and shall not use or register anything (including a corporate name, trade mark, design or domain name) which incorporates the Game Name (or any name adopted by Carrot in accordance with clause 4.5) or anything confusingly similar.

4.8. For the purpose of this Agreement, "use" of the Game and/or Game Rights shall include without limitation the right to load, trial, test, accept, integrate, activate, store, transmit, display, launch and/or host the Game and the right to copy the Game for these purposes.

4.9. Carrot may sub-license its rights in the Game, Developer Materials and Game Documentation to its Affiliates and to Stake and/or any third party hosting provider.

4.10. To the extent applicable and subject to clause 7.1.a)(viii), the Developer reserves its rights to use in future games any generic proprietary tools and technologies which it has developed that can be used across a number of games and have not been specifically developed for the Game.

4.11. The Developer hereby covenants and undertakes that they have not done or omitted to do and will not do or omit to do any act, matter or thing whereby the Game Rights may be invalidated.

4.12. The Developer acknowledges and agrees that in the event Carrot carries out any development, modification, update and/or alteration to the Game (either itself or through an Affiliate or third party), including as required under clauses 14.2.d)(ii) and 14.5, Carrot shall own all Intellectual Property Rights in such developments, modifications, updates and/or alterations.

4.13. **Integration of the Game**

a. The Parties acknowledge that the integration between the ICT Infrastructure or Stake's (or relevant third-party provider's) infrastructure required for activation of the Game pursuant to clause 4.14, including but not limited to front-end systems, back-end systems, solutions and interfaces, and the Game will require efforts from both Parties. The Parties shall act in good faith to provide all knowhow and resources as required to make the integration process as smooth and quick as possible. The Developer will provide Carrot with all documentation relating to the integration, operation and ongoing management of the Game (including any other existing support such as IT maintenance and/or customer support) on the Effective Date ("**Game Documentation**").

b. The Developer acknowledges and agrees that Carrot may charge a service fee in connection with the use of Carrot's remote gaming server. The amount and terms of the service fee may be notified to the Developer in writing from time to time.

4.14. **Integration and Activation of the Game**

a. The Developer acknowledges and agrees that:

(i) the Developer will licence the Game Rights to Carrot and / or Nominee in accordance with clause 4.4; and

(ii) Carrot reserves the right to reject integration and / or activation of the Game on the Website at its absolute discretion, including but not limited to where the Game fails to meet the quality standards determined by Carrot from time to time or where any required formal certification or jurisdiction-specific approval has not been obtained.

a. The Developer acknowledges and agrees that the Game will exclusively be offered on the Websites and will not be licenced or sold to any third parties which are not Carrot Affiliates.

b. Subject to clause 4.14, only once acceptance of the Game under clause 3 has been completed, the Game will be licensed by Carrot to Stake (or relevant third-party provider) to activate on the Website.

c. Subject to the Developer satisfying the requirements set out in clause 7.1.a), Carrot may at its absolute discretion facilitate integration of the Game before all the obligations arising from clause 3 have been formally completed and / or documented. If this occurs the Developer grants to Carrot a royalty-free, exclusive, non-revocable and perpetual right to sub-licence the Game to Stake (or a third-party provider), to

facilitate integration on the Website. The acceptance process referred to in clause 4 applies to each individual Game considered by Carrot's for integration and / or activation.

4.15. If Carrot exercises its right to reject the Game pursuant to clause 4.9.a)(ii), any licence granted under clause 4.2 shall terminate with respect to the applicable Game.

4.16. Carrot and / or Stake (or relevant third-party provider) will only activate the Game on the Website and make it available to Players after the integration and acceptance testing has been completed and the Game has been enabled for launch.

4.17. Without prejudice to this clause 4 and clause 7.1.b)(i), Carrot and / or Stake (or any relevant third-party provider) reserves the right to remove or deactivate the Game at any time at its absolute discretion.

4.18. **Right to seek and maintain certification**

a. The Developer grants Carrot and/or its Nominee the full right and authority to undertake all acts and things necessary to obtain and maintain any required formal certification, regulatory authorisation, or jurisdictional approval for the Game in any regulated market where the Websites operate or may operate. This includes, without limitation, submitting the Game and any associated Game to Regulatory Authorities for review and testing against relevant criteria.

b. The Developer shall cooperate fully with Carrot and/or its Nominee in this regard, providing all necessary information, documentation, and technical assistance as reasonably requested by Carrot and/or its Nominee, at the Developer's cost where such requirements arise from the Game's design or Developer's obligations.

c. For the purpose of facilitating such certification and subsequent activation, the Developer explicitly grants Carrot the right to implement fairness mechanisms, conduct necessary testing, and to assign or sub-licence any and all Intellectual Property Rights in the Game (including Game Rights to the relevant operating entity of the respective Website (which may be Carrot, Stake, or another Affiliate to the extent required by such Regulatory Authority or as necessary to enable activation and operation of the Game in compliance with Applicable Law.

5. Developer Obligations

5.1. **Use of the Game**

a. The Developer shall not under any circumstances:

(i) access the Game or (subject to clause 4.10) use the Game Rights (including, without limitation the Game Documentation) in order to build or create a similar or competitive game, to be determined by Carrot at its absolute discretion;

(ii) distribute, license, exploit and/or permit any third-party to use any of the Game Rights;

(iii) develop, produce, make, distribute, license, and/or exploit any Derivative Work of the Game and/or permit any third-party to do so;

(iv) copy the Game by any means for any purpose whatsoever except if and to the extent that may be required for compliance with Applicable Law and, in such cases, subject to Carrot's prior written consent that shall not be unreasonably withheld, conditioned or delayed and subject to the Developer providing evidence to Carrot of such requirements; and

(v) knowingly include in the Game any material that could adversely affect Carrot or any of its Affiliate's name, image or reputation.

5.2. Except to the extent the Developer is prohibited under any arrangement with a third party, the Developer shall notify Carrot of a New Game under development (such notice to include sufficient details of the proposed specification and functionality of the New Game) and shall give Carrot the right of first refusal to take a licence of such New Game prior to making the New Game available or disclosing details of it to any other person. Carrot shall confirm in writing to the Developer within thirty (30) days of receipt of a notice pursuant to this clause 5.2 whether or not it wishes to proceed with a licence for the New Game. If Carrot confirms that it does wish to proceed with a licence to the New Game, this Agreement shall be varied to include the New Game as a "Game". If Carrot rejects its right of first refusal or fails to respond within thirty (30) days of receipt of notice pursuant to this clause 5.2, Carrot shall forfeit its right of first refusal to the applicable New Game. If Carrot's right of first refusal for the New Game is forfeited, Carrot shall still be entitled to request a licence of the New Game under this Agreement but the Developer shall not be obliged to grant such a licence and/or any terms as to exclusivity or first mover advantage in respect of that New Game to Carrot.

5.3. **Testing and updates to the Game**

i. The Developer is responsible for assisting with the integration and functional testing of the Game prior to the Commencement Date for the purpose of ensuring the proper functioning of the Game on the Website at no cost to Carrot or Stake (or any relevant third-party provider).

ii. In the event of any update to the Game production environment, the Developer will (at its cost) undertake the testing required to ensure that the Game is functioning properly after such update on the Website.

iii. In case the Game has been launched and is subsequently subject to updates (for whatever reason), Carrot and /or its Affiliates will test the functionality of the updated Game, and the Developer will, where reasonably requested by Carrot, undertake the testing required to ensure that the updated Game is functioning properly on the Website. For the sake of clarity, all testing referred to in this Agreement shall be performed on all applicable operating systems and browsers.

iv. The Parties acknowledge that whilst the initial version of the Game uploaded to the ICT Infrastructure and/or Stake's (or relevant third-party provider's) ICT infrastructure may be accepted by Carrot, Carrot does not guarantee that any updated version of the Game will be accepted. should the updated version be substantially different from the initial version. If an updated version of the Game is not accepted by Carrot upon upload (pursuant to clause 3) or as part of integration and / or activation assessment (pursuant to clause 4), then:

(i) clause 4.15 applies; and

(ii) Carrot may (at its sole discretion) continue the activation of the existing version of the Game on the Website and the Developer will continue to receive the Licence Fee from this activation.

5.4. **Errors affecting the Game**

i. The Developer shall immediately notify Carrot in writing of any Errors it has detected in the services.

ii. The Developer shall, in line with Good Industry Practice, correct any Errors discovered during acceptance testing. If this is not reasonably possible, the Developer shall work around the Error at its own expense. If the Error is so significant that, due to the Error, the purpose of the Agreement remains essentially unfulfilled, then clause 7.1 applies.

iii. The Developer shall, in line with Good Industry Practice, correct any Errors arising throughout the term of the Agreement.

5.5. **Other Developer Obligations**

a. The Developer shall:

i. comply with any codes of conduct, policies or other notices provided by Carrot in writing from time to time;

ii. ensure that the Game's underlying software facilitates all necessary processes to enable Stake (or relevant third-party provider) to determine the source of funds of Players and verify the identity of such individuals;

iii. ensure that the use and/or exploitation of the Game and any associated content or data, the Developer Materials and/or the Game Documentation in accordance with this Agreement does not infringe the Intellectual Property Rights of any third-party;

iv. provide all documents and do all acts and things requested by Carrot to verify the Developer's identity and the location of the Developer's nominated cryptocurrency wallet provided pursuant to clause 8.1.d)(v). Notwithstanding anything to the contrary in this Agreement and without derogating from the rights in clause 14, Carrot reserves the right (at its absolute discretion) to terminate this Agreement immediately if the Developer fails to provide the information requested pursuant to this sub-clause or, if on the basis of the information provided, Carrot determines that entering or continuing a contractual relationship with the Developer would breach Applicable Laws or have an adverse impact on Carrot or its Affiliates' reputation.

5.6. **Developer to facilitate activation on the Website**

i. For the purpose of enabling the Developer to assess the functionality of the Game, determine whether any updates are required and assess the Game for Errors the Developer will be given a test account on the Website. This test account will be able to play the Game, however, will not be able to withdraw any winnings from the Website.

ii. On and from the Commencement Date, Stake (or relevant third-party provider) will be responsible for all day-to-day operational activities in relation to the Game, including but not limited to, marketing, customer support and care, fraud screening and payment management.

5.7. **Data Access for Operational Purposes**

i. The Developer acknowledges and agrees that Carrot and/or its Nominee shall have the right to access and use all data generated by or flowing through the Developer's systems, interfaces, and platforms to the extent necessary for Carrot and/or its Nominee to perform their obligations and exercise their rights under this Agreement. This includes, without limitation, data required for Know Your Customer (KYC) verification purposes, financial administration, calculation and processing of Licence Fees, and general operational management of the Game and the Website.

ii. The Developer shall provide all necessary technical assistance, access, and documentation to facilitate such access and use by Carrot and/or its Nominee in a timely manner, in accordance with Good Industry Practice.

6. Representations & Warranties

6.1. Each Party hereby represents and warrants to the other Party that:

(i) each Party has the necessary power, authority and approval, corporate or otherwise, to enter into, execute and perform its obligations under this Agreement;

(ii) this Agreement, when executed and delivered by such Party, will constitute valid and legally binding obligations of such Party, enforceable in accordance with its respective terms;

(iii) the execution of this Agreement, and the fulfilment of the terms hereof, will not, according to the Parties' knowledge on the Effective Date, result in any breach of any judgement, decree or order of any court or governmental body, any Applicable Law (including, without limitation, any applicable anti-trust or competition regulations), or the articles of association of such Party or any contract binding on such Party;

(iv) it shall comply with Applicable Law;

(v) to the best of its knowledge and belief, no litigation or administration action is in process or being threatened which involves a Party and which is reasonably likely to affect in a material manner the ability of

such Party to perform its obligations under this Agreement;

(vi) it will not knowingly say or do anything or omit to do anything that may bring the reputation of the other Party into disrepute or knowingly harm the goodwill in such Party's Intellectual Property Rights; and

(vii) it is able to pay its debts when they fall due.

6.2. The Developer represents and warrants that:

i. It has the right to enter into this Agreement and it has title to and/or the authority to grant the rights to Carrot as set forth in this Agreement, including but not limited to such rights to use and license any Intellectual Property Rights which are licensed to Carrot under and/or which may be required to perform its obligations under this Agreement;

ii. it will comply with all the rules and obligations set forth in this Agreement, including but not limited to any Exhibits contained in this Agreement;

iii. all acts and things rendered under this Agreement are in compliance with Applicable Law and that the Developer is solely responsible for any obligations or duties applicable to it under Applicable Law. Carrot shall not be responsible or liable for any action, inaction, negligence, or non-compliance by the Developer and expressly disclaims any and all liabilities that may arose in relation thereto;

iv. it will carry out its obligations arising from this Agreement with all due skill and diligence and in a good and workmanlike manner, and in accordance with the Good Industry Practice;

v. it has obtained all applicable and necessary and required licences, consents and permits to perform its obligations arising from this Agreement.

vi. the Game will operate in accordance with the Game Documentation. In the event that it does not, and upon receiving written notice from Carrot, the Developer will modify and / or update the Game to make it perform in accordance with the Game Documentation;

vii. to the extent that the Game includes any third-party software, including software licensed under licenses known as "free" or "open source" (or equivalent), the Developer warrants that it fully complies with all terms of such third-party licenses; the Developer has not included or used any software licensed under the General Public Licence or any similar open source licence containing a "copyleft" requirement ("**Restrictive Open Source Code**") in, or in the development, or, the Game, nor does the Game operate in such a way that it is compiled with or linked to any Restrictive Open Source Code.

viii. The individual or entity completing the electronic acceptance process for this Agreement (e.g., "ticking" a box), hereby warrants that they are either the sole owner of the Developer entity/team or are duly authorised by the Developer to bind the Developer to the terms of this Agreement and to grant all rights and licences herein, including without limitation the Game Rights. The Developer acknowledges that Carrot relies on this warranty for the validity and enforceability of this Agreement.

7. Corrective Action by the Developer

7.1. **Game Standards**

a. The Developer warrants that at the Commencement Date, the Game:

i. is fully functional;

ii. is compliant with all technical specifications advised by Carrot and / or its Affiliates from time to time in writing;

iii. operates in conformance with the Game Documentation;

iv. is not affected by any Disabling Devices or Errors, and the Developer has, in accordance with Good Industry Practice, used all anti-virus and anti-malware software to screen the Game to ensure that it is not subject to any Disabling Devices or Errors;

v. does not incorporate, contain or refer to any Prohibited Content;

vi. does not contain any known vulnerabilities or latent vulnerabilities and that all reasonable efforts have been made to identify and remove any such issues in accordance with Good Industry Practice;

vii. visuals align and are synchronised with the Game math; and

viii. is not substantially similar to any game previously uploaded to the Website, including without limitation games that differ only in minor respects such as visual elements or cosmetic changes.

b. If there is a breach of clause 7.1a):

i. the Game may be removed or deactivated from the Website, at Carrot's absolute discretion; and

ii. Carrot will notify the Developer and the Developer will take immediate corrective action, in good faith and in cooperation with Carrot and Stake (or relevant third-party provider), to rectify that issue as soon as reasonably practicable at no cost to Carrot.

c. If the breach of clause 7.1a) is not remedied to Carrot's satisfaction within a reasonable period of time, then Carrot reserves the right (at its absolute discretion and without any liability to the Developer whatsoever) to deactivate the Game and terminate the licence granted under clause 4.2 in relation to the applicable Game. For the avoidance of any doubt, Carrot may elect only to deactivate the Game and not terminate the applicable licence pursuant to this clause.

d. In the event that Carrot exercises its right pursuant to clause 7.1c), then the Developer's right to receive the Licence Fee ceases on the date the Game is removed or deactivated from the Website.

8. Developer Entitlements

8.1. **Payment**

a. Unless otherwise agreed by Carrot in writing from time to time, if the Licence Fee calculated for any month is less than USD $1,000, that amount shall not be paid and will instead roll over and accumulate with any Licence Fee amounts calculated in subsequent months. Once the combined accrued Licence Fee equals or exceeds $1,000, the full accumulated amount shall become payable in the next monthly payment cycle. No interest shall accrue on any deferred amounts.

b. The Licence fee shall only become payable upon all the following criteria being satisfied:

i. activation of a Game; and

ii. the delivery of the applicable object code (and /or Source Code if requested by Carrot) in relation to each Game or New Version in accordance with clause 4.2; and

iii. the Developer submitting all applicable identification and verification documents requested by Carrot from time to time.

c. For the avoidance of doubt, the Parties acknowledge that the Licence Fee is payable for each Game or New Version which is active on the Website during that given month of the Term.

d. The Licence Fee shall:

i. be a fixed percentage during the Term of the Agreement payable in accordance with clause 8.1e) ;

ii. be payable in arrears on a monthly calendar basis from the Commencement Date unless expressly agreed otherwise in writing by Carrot;

iii. be calculated on Gross Gaming Revenue generated by the Game on the Website;

iv. be calculated by Stake (or relevant third-party provider) on the basis of the data compiled by Carrot and / or Stake (or relevant third-party provider). For the avoidance of doubt, the data collected by Carrot and / or Stake (or relevant third-party provider) shall be conclusive and final, and the Developer does not have any right to audit that data or underlying calculations; and

v. be paid to the wallet nominated by the Developer from time to time in writing. Where the Developer updates its wallet details, it is the Developer's sole responsibility to notify Carrot as soon as practicable, and in any event at least fourteen (14) days before the end of a given month. Where the Developer fails to notify Carrot of any such change by this deadline, Carrot does not take any responsibility for remitting payment to an incorrect account or any payment processing delays which may arise. In this regard, the Developer acknowledges and agrees that Carrot and / or its Nominee may process payment, and the Developer authorises Carrot to share its wallet details with Nominee for this purpose only.

e. The Licence Fee shall be payable as follows:

i. For the first month of the Term, the Licence Fee payable to the Developer will be calculated as a percentage of the GGR generated by all activated Games for that first month.

ii. From the second month of the Term onwards, the Licence Fee shall be calculated as a percentage of the GGR for that month, provided that if the GGR for any month is negative, the negative amount shall be carried forward and offset against future positive GGR (a “Negative Rollover”). No Licence Fee shall be payable to the Developer for any month until the cumulative GGR (taking into account any Negative Rollover) becomes positive.

iii. Where the cumulative GGR in a month becomes positive after offsetting a prior Negative Rollover, the Developer shall be entitled to receive the Licence Fee only on the amount by which the cumulative GGR exceeds zero.

iv. Where the Developer has made available multiple Games under this Agreement, the GGR from all such Games shall be aggregated and assessed on a combined basis for the purposes of calculating any Licence Fee payable to the Developer. The Developer shall not be entitled to receive a Licence Fee in respect of any month in which the net aggregated GGR (after deducting any Negative Rollover) is zero or negative.

v. Example:

- Month 1: aggregate GGR is $100,000 → Licence Fee is payable on $100,000
- Month 2: aggregate GGR is $50,000 → Licence Fee is payable on $50,000
- Month 3: aggregate GGR is –$50,000 → No Licence Fee payable; $50,000 carried as Negative Rollover
- Month 4: aggregate GGR is $30,000 → No Licence Fee payable; Negative Rollover reduced to $20,000
- Month 5: aggregate GGR is $30,000 → Licence Fee payable on $10,000 (after $20,000 Negative Rollover applied)
- Month 6: aggregate GGR is $30,000 → Licence Fee payable on full $30,000

8.2. **Game Position and Activation**

a. Carrot, on its own behalf and on behalf of Stake (or any relevant third-party provider), does not make any warranty about the functionality of the Website with respect to the activation of the Game. Notwithstanding the foregoing, Carrot guarantees that it and Stake (or relevant third-party provider) will use best endeavours to remedy any technical issues affecting the activation and accessibility of the Game on the Website as soon as becoming aware of the same.

b. For the avoidance of any doubt, Carrot does not guarantee:

i. any particular uptime with respect to the Game being accessible to Players on the Website;

ii. that the Website will be error-free or run without errors during the Term; or

iii. that the Game will appear in any specific category or with any particular prominence during the Term; or

iv. that Carrot will drive traffic to the Game.

9. Protection of the Intellectual Property Rights

9.1. Each Party shall immediately notify the other Party in writing giving full particulars if any of the following matters come to its attention:

i. any actual, suspected or threatened infringement of the Game Rights;

ii. any claim made or threatened that the use and/or exploitation of the Game and any associated content or data, the Developer Materials and/or the Game Documentation in accordance with this Agreement infringes the rights of any third party; or

iii. any other form of attack, charge or claim to which the Game Rights may be subject.

9.2. In respect of any of the matters listed in clause 9.1, subject to clause 9.3:

i. the Developer shall, at their absolute discretion, decide what action to take if any;

ii. the Developer shall have exclusive control over, and conduct of, all claims and proceedings;

iii. Carrot shall not make any admissions other than to the Developer and shall provide the Developer with all assistance that the Developer may reasonably require in the conduct of any claims or proceedings; and

iv. the Developer shall bear the cost of any proceedings and shall be entitled to retain all sums recovered in any action for the Developer's own account.

9.3. In the event that the Developer fails to take action to defend or enforce the Game Rights under clause 9.2 within fourteen (14) days of notice by Carrot to do so and/or if any third party infringement of the Game Rights in the Territory interferes materially with Carrot's business, Carrot may take such action as it sees fit to defend and/or enforce the Game Rights, including commencing proceedings and may require the Developer to lend its name to such proceedings and provide reasonable assistance, including an award of costs against it, directly resulting from the Developer's involvement in such proceedings. Where such proceedings are conducted by Carrot, Carrot may apply 100% of all Licence Fees due under clause 8.1 subsequent to the date of notification by Carrot to the Developer of the relevant infringement towards any costs directly incurred in relation to the proceedings. Any waiver of Licence Fees by the Developer shall only apply for as long as Carrot actively pursues and properly conducts the proceedings. Any damages recovered by Carrot in the proceedings shall first be applied to compensating the Developer for any Licence Fees waived under this clause 9.3 and Carrot shall be entitled to retain all other sums recovered in any action for Carrot's own account.

10. Indemnity

10.1. **Infringement of third-party rights**

a. The Developer warrants that the use, activation, promotion and/or exploitation of the Game and any associated content or data, the Developer Materials and/or the Game Documentation by Carrot and/or Stake (and/or any relevant third-party provider) in accordance with this Agreement does not infringe any Intellectual Property Rights or trade secrets of any third party.

b. The Developer shall, at its own expense, defend, indemnify and hold Carrot and its Affiliates harmless against any and all claims or actions alleging that the use, activation, promotion and/or exploitation of the Game and any associated content or data, the Developer Materials and/or the Game Documentation in accordance with the terms of this Agreement, infringes any Intellectual Property Rights of a third party (a "**Claim**") and shall fully indemnify and hold harmless Carrot and its Affiliates from and against any losses, damages, costs (including reasonable legal fees) and expenses incurred by or awarded against Carrot and/or its Affiliates as a result of, or in connect with, any such Claim.

c. The Developer shall disclose all third-party contracts or licences related to the Game prior to the Commencement Date. If any such contracts exist, the Developer shall obtain consents to sublicense them to Carrot (prior to the Commencement Date) or ensure that Carrot's continued use will not be affected.

d. In the event that it is established that the use, activation, promotion and/or exploitation of the Game and any associated content or data, the Developer Materials and/or the Game Documentation under this Agreement infringes any Intellectual Property Rights of a third party, the Developer shall at its own expense either:

i. obtain the right of continued use of the infringing Intellectual Property Rights in accordance with the provisions of this Agreement; or

ii. modify the Game, Developer Materials and/or Game Documentation (as applicable) in order to eliminate the infringement, provided, always that after such modification, the Game shall meet all the requirements set forth in this Agreement or specified by Carrot and shall comply with Applicable Law.

e. If the Developer is unable to accomplish (a) or (b) within a reasonable period of time from receiving the Claim (to be determined by Carrot at its absolute discretion), Carrot may deactivate the Game from the Website and exercise the right arising pursuant to clause 7.1.c).

10.2. **General Indemnity**

a. The Developer will at all times indemnify, defend, and hold harmless Carrot and its Affiliates and any of its employees, agents, successors, and assigns from and against any and all third-party claims, damages, liabilities, costs and expenses, including reasonable legal fees, arising out of:

i. the Developer's breach of any obligation, covenant, representation or warranty contained in this Agreement;

ii. any enforcement of this Agreement;

iii. the Developer's unauthorized use of the Game Rights;

iv. the Developer's inclusion, embedding or distribution by the Developer to Carrot of any viruses, malware, spyware, ransomware or other malicious code or harmful components in the Game or any related materials provided under this Agreement;

v. the Developer's negligence and/or misconduct; and

vi. the Developer's violation of any Applicable Law.

b. In the event of occurrence of clause 10.2a)(iv) Carrot shall also be entitled to a full refund of all Licence Fees and any other amounts paid to the Developer under this Agreement.

11. Limitation of Liability

11.1. Nothing in this Agreement limits any liability which cannot legally be limited, including but not limited to liability for:

a. death or personal injury caused by negligence;

b. fraud or fraudulent misrepresentation; and

c. any other liability which cannot be excluded or limited under Applicable Law.

11.2. To the maximum extent permitted by Applicable Law, in no event will Carrot be liable to the Developer for any special, exemplary, punitive, incidental, or consequential damages, including but not limited to lost profits, arising out of or in connection with this Agreement, regardless of whether such damages were foreseeable.

11.3. The Developer agrees, promises and covenants that neither they, nor any person, organisation or other entity acting on their behalf will file, charge, claim, sue or cause or permit to be filed, charged or claimed, any action for damages or other relief (including injunctive, declaratory, monetary relief or other) against Carrot before any regulatory, adjudicatory, governmental, judicial, quasi-judicial or similar authority, involving any matter occurring in the past up to the Effective Date of this Agreement, relating to or involving or consequential upon any breach of applicable law by the Developer, or any continuing effects of actions, inactions, or practices which arose prior to the Effective Date of this Agreement, or involving and based upon any claims, demands, causes of action, obligations, damages or liabilities which are the subject of this Agreement.

11.4. The Developer shall be solely liable for any costs, liabilities, or obligations incurred prior to the Effective Date, and shall indemnify Carrot for any losses arising from such liabilities.

11.5. References to liability in this clause 11 include every kind of liability arising under or in connection with this Agreement including but not limited to liability in contract, tort (including negligence), misrepresentation, restitution or otherwise.

12. Confidentiality

12.1. Information in respect of which a Receiving Party can prove any of the following shall not be deemed to be "Confidential Information" for the purposes of this Agreement:

a. it was in the public domain prior to the date of coming into force of this Agreement or entered the public domain after that date through no wrongful act or default of a Receiving Party;

b. it is already known to or in the possession of the Receiving Party free of any obligation to keep it confidential at the time of disclosure;

c. it is disclosed by a Receiving Party in accordance with the terms of a Disclosing Party's prior written approval;

d. it was received by a Receiving Party expressly without obligation of confidence from a third party who did not acquire it under an obligation of confidence from a Disclosing Party;

e. it was developed by a Receiving Party completely independently of the information disclosed by the Disclosing Party; or

f. it is information which a Receiving Party is obliged to produce pursuant to an order of a court or of a competent jurisdiction or administrative tribunal.

12.2. The Parties shall not disclose Confidential Information to any person or entity except its employees or consultants, or those of its Affiliates, who are required to have the Confidential Information in order to assist it in acting as contemplated by the Agreement, and only to the extent necessary. Prior to disclosing any Confidential Information to such employees or consultants, the Disclosing Party shall ensure that they are aware of the provisions of this Agreement and have signed non-disclosure agreements with terms substantially similar to those contained in this Agreement. The Disclosing Party shall bear full responsibility and liability for the actions of such employees and consultants concerning the Confidential Information, at all times, regardless of termination of any labour, employment or other relationship with any such employees and consultants.

12.3. In the event that the Receiving Party becomes legally compelled to disclose Confidential Information of the Disclosing Party due to any request from a Regulatory Authority or other governmental authority, in any

judicial proceeding, or based on Applicable Law, the Receiving Party shall limit such disclosure as far as possible only to such information which is specifically requested and shall use all reasonable efforts to give the Disclosing Party at least 10 (ten) days prior written notice (or such shorter period if a response or answer is due within fewer than 10 (ten) days of its intention to comply, so that Disclosing Party may seek a protective order or other appropriate remedy to obtain confidential treatment of such Confidential Information. Notwithstanding the above, each Party may disclose to any Regulatory Authority in accordance with the Applicable Law any and all Confidential Information requested by the Regulatory Authority.

12.4. The Receiving Party shall treat the Confidential Information as strictly confidential and with at least the degree of care that it treats similar materials of its own in order to prevent unauthorized disclosure of Confidential Information to others, or a higher standard of care reasonable under the circumstances.

12.5. Upon termination or expiration of this Agreement, or upon the written request of Carrot at any time, the Developer shall:

a. immediately cease to use Carrot's Confidential Information, for any purpose; and

b. within ten (10) days, at Carrot's option, either:

i. return to Carrot all of its Confidential Information (regardless of form), any copies made thereof, and any materials containing or reflecting any portion of any Confidential Information, to the extent that they remain in the possession of the Developer; or

ii. destroy all Confidential Information and materials regardless of form.

12.6. Upon Carrot's request, the Developer shall certify in writing that the Developer has complied with this clause.

12.7.

13. Right of First Offer

13.1. If: (a) the Developer or any of the Developer's Affiliates decides to sell or approves any intended sale of the Game Rights (in whole or in part); or (b) the Developer or the relevant Developer's Affiliate receives a written indication of interest or offer to buy any of the Game Rights from a third party, which it wishes or is minded to accept, (an "**IPR Sale**"), the provisions of this clause 13 shall apply.

13.2. Prior to initiating or entering into any discussions or negotiations with any third party in relation to the IPR Sale, the Developer shall provide written notice to Carrot of the proposed IPR Sale stating: (i) the Game Rights which are subject to the proposed IPR Sale (the "**Relevant Game Rights**"); (ii) a description of the proposed IPR Sale including an indication of the proposed price (to the extent known and subject to confidentiality restrictions); and (iii) that the Developer, by such notice, is providing Carrot an opportunity to

make an offer for the Relevant Game Rights (the "**ROFO Notice**"). Notwithstanding the foregoing an "**IPR Sale**" shall not include any sale or assignment of the Game Rights to an Affiliate of the Developer in connection with a restructuring of the Developer's business (and in accordance with clause 17.9.a).

13.3. Following delivery of the ROFO Notice to Carrot, Carrot shall have thirty (30) days from the receipt of the ROFO Notice (the "**ROFO Offer Period**") to either: (i) deliver to the Developer a written offer, together with a description of the proposed material terms and conditions that Carrot is prepared to offer in connection therewith, whereby Carrot intends to acquire the Relevant Game Rights pursuant to an IPR Sale (a "**ROFO Offer**"); or (ii) deliver to the Developer a written notice waiving Carrot's right pursuant to this clause 13 in respect of the IPR Sale (a "**ROFO Waiver**"). Failure of Carrot to deliver a ROFO Offer prior to the end of the ROFO Offer Period shall mean that the Developer shall be free to sell or otherwise assign the Relevant Game Rights to a third party, subject to clause 17.10.

13.4. If Carrot delivers to the Developer a ROFO Waiver, or otherwise fails to accept the right of first offer to acquire the Relevant Game Rights in accordance with clause 13.3, then the Developer shall be able to enter into discussions or negotiations with any third party in relation to the sale or purchase of the Relevant Game Rights.

13.5. If Carrot delivers a ROFO Offer, the Developer shall notify Carrot in writing whether it provisionally accepts or rejects the offer within seven (7) days of delivery of the ROFO Offer. If the Developer rejects the ROFO Offer, Carrot may submit a revised ROFO Offer no later than fourteen (14) days after such rejection.

13.6. If the Developer notifies Carrot that it accepts a ROFO Offer or revised ROFO Offer in accordance with clause 13.5, Carrot shall have a maximum period of sixty (60) days (or such extended period as may be agreed by the Parties in order to negotiate such terms) to negotiate exclusively with the Developer in good faith for the purchase of the Relevant Game Rights substantially on the terms set out in the ROFO Offer, including the purchase price.

13.7. The Developer agrees that it shall not and shall not directly or indirectly cause any other party including its shareholders to initiate or continue any discussions and not to make any commitments to third parties in relation to the sale or transfer of the Relevant Game Rights during the thirty (30) day period in clause 13.3 and/or if applicable, the sixty (60) day period in clause 13.6 (or such extended period as may be agreed by the Parties in order to negotiate such terms).

13.8. If the Developer rejects a ROFO Offer or if Carrot and the Developer have not agreed terms for the sale and assignment of the Relevant Game Rights to Carrot within a period of sixty (60) days of any acceptance of a ROFO Offer by the Developer, or such extended period as may be agreed by the Parties in order to negotiate such terms, the Developer shall be free to sell or otherwise assign the Relevant Game Rights to a third party subject to clause 17.10 provided that the Developer may not sell or otherwise assign the Relevant Game Rights to a third party on substantially less favourable terms, including pricing, than the last set of terms offered by Carrot pursuant to a ROFO Offer.

13.9. This right of first offer shall expire on termination of this Agreement.

13.10. In addition to Carrot's right of first offer set out in clauses 13.1 to 13.9, the Parties acknowledge and agree that Carrot may offer to buy any of the Game Rights from the Developer prior to the procedure set out in 13.1 to 13.8 arising or afterwards in the event a sale is not completed. In the event Carrot makes an offer to buy any of the Game Rights, the Parties shall negotiate in good faith to agree the material terms and conditions for the sale, including the purchase price for the relevant Game Rights and, at Carrot's discretion, the proposed purchase of the Game Rights may take place at the point the terms and purchase price are agreed or at a future point in time.

14. Term and Termination

14.1. The Term of the Agreement is set out in the Front Sheet.

14.2. **Termination**

a. Either Party shall have the right to terminate this Agreement for cause, in whole or in part, upon written notice to the other Party with immediate effect in the event of any of the following:

i. the other Party is declared bankrupt, is put into liquidation, or otherwise becomes insolvent; or

ii. the other Party commits a material breach of any of the terms and conditions of this Agreement and, if the breach is capable of being remedied, does not remedy such breach within seven (7) days of written notice in which notice the possibility of termination has been explicitly mentioned and the steps required to be carried out in order to rectify the breach are, to the extent possible, set out with reasonable clarity.

b. Carrot shall have the right to immediately terminate this Agreement in the event of any of the following:

i. the Developer fails to satisfy any of the obligations arising from this Agreement;

ii. the Developer refers to the Carrot or its Affiliates in any way that might impact or damage its reputation;

iii. the Developer acts in any way that is deemed by Carrot as inappropriate;

iv. Carrot or its Affiliates receive a formal notice from a government, official or Regulatory Authority in relation to the Agreement;

v. the Applicable Law changes during the Term in a manner that affects this Agreement;

vi. material is published or communicated which may indicate to Carrot or its Affiliates that the affiliation arising from this Agreement will become prohibited under laws or regulation;

vii. there is a change of control of the Developer (within the meaning of section 1124 of the Corporation Tax Act 2010;

viii. or

ix. The Developer subcontracts any of its obligations under this Agreement without having first obtained Carrot's prior written consent.

c. Without prejudice to the rest of this clause 14.2, Carrot may terminate this Agreement at any time for convenience by giving thirty (30) days written notice to the Developer.

d. Where Carrot terminates this Agreement in accordance with clause 14.2.b)(i) because the Developer breaches clauses 4.4 (by licencing the Game to any third party which is not Carrot or a Nominee) or clause 13 (by selling the Game to any third party which is not Carrot or a Nominee without adhering to the procedural steps in this Agreement), Carrot may choose at its discretion to either:

i. end the licence granted under clause 4 at the same time; or

ii. retain the licence granted under clause 4 on an irrevocable and perpetual basis and keep any Game activated on the Website. In this case, the Parties agree that Carrot would have the express right to maintain, support or update the software themselves (including via third parties).

e. If this Agreement is terminated pursuant to clause 14.2.d), the Developer's entitlement to the Licence Fee ceases on the date that termination becomes effective, notwithstanding that the Game may continue to be activated on the Website.

f. Upon termination of this Agreement on any basis aside from the circumstances addressed in clause 14.2.d), Carrot may choose at its discretion to either:

i. end the licence granted under clause 4 at the same time; or

ii. continue paying the License Fee whilst the Game is activated on the Website. If Carrot elects to continue paying the Licence Fee after termination of this Agreement however, Carrot will retain the licence granted under clause 4 on an irrevocable basis as long as it keeps the Game activated on the Website (at Carrot's absolute discretion). In this case, the Parties agree that Carrot would have the express right to maintain, support or update the software themselves (including via third parties).

g. This Agreement will automatically expire if the Developer fails to secure activation of any Game during the Initial Term or any Renewal Term(s). For the avoidance of doubt, the expiry of the Agreement will be the last day of the Initial Term or the applicable Renewal Term.

14.3. **Suspension**

a. Carrot may (without liability to the Developer) suspend its payment obligations in respect of the Licence Fee with immediate effect by delivering written notice thereof to the Developer if the Developer fails to:

i. adhere to any warranty or undertaking set out in the Agreement;

ii. fails to remedy a notified breach within fourteen (14) days; or

iii. fails to rectify an Error in accordance with clause 5.4.

b. Notwithstanding the above, if Carrot suspends payment of the Licence Fee due to breach of this Agreement by the Developer (to be determined by Carrot at its absolute discretion), the Game may continue to be activated on the Website, however the GGR generated during the suspension will not contribute to the calculation of the Licence Fee.

14.4. **Removal of Game from Website**

a. The Developer acknowledges and agrees that neither Stake (or any relevant third-party provider) or Carrot will be obliged to give any notice to the Developer and neither will Stake (or any relevant third-party provider) or Carrot will have any liability to the Developer where the Game is deactivated from the Website at any given time due to:

i. a third-party claim in relation to the Game;

ii. the advice of its legal advisors; or

iii. reasonable regulatory concerns.

14.5. **Step-in Rights**

a. If the Developer fails to perform any of its obligations under this Agreement (including, without limitation, failure to rectify any Error or meet support obligations), Carrot shall be entitled, without prejudice to its other rights and remedies, to take such steps as it considers reasonably necessary to perform or procure the performance of those obligations. The reasonable and properly incurred costs of such steps may, at Carrot's discretion, be recovered from the Developer or set off against any sums due to the Developer under this Agreement.

15. Data Protection

15.1. The Parties acknowledge that for the purposes of applicable Data Protection Legislation, Carrot and/or its Affiliates shall act as the independent data controller with respect to any Players' personal data collected, accessed, or otherwise processed in connection with its installation, integration, hosting and use of the Game on the Website, according to the terms and definitions of this Agreement.

15.2. The Developer shall not process any personal data of Players of the Game and shall have no access to such data in the course of this Agreement, except where such access or processing is strictly necessary for the purpose of performing maintenance, updates, or corrections to the Game. In such limited cases, the Developer shall act as a data processor on behalf of Carrot and/or its Affiliates and shall comply with its obligations under Data Protection Legislation applicable to processors.

15.3. Carrot and/or its Affiliates are solely responsible for determining the means and purposes of processing personal data obtained in connection with its use of the Game and shall ensure that such

processing is conducted in accordance with the Data Protection Legislation.

15.4. Carrot and/or its Affiliates shall implement appropriate technical and organizational measures to safeguard the personal data of its end users and to prevent unauthorized or unlawful processing, accidental loss, destruction, or damage.

15.5. Carrot and/or its Affiliates shall be solely responsible for responding to and fulfilling any data subject requests under the Data Protection Legislation, including requests for access, rectification, erasure, restriction, portability, or objection.

15.6. Carrot and/or its Affiliates shall ensure it provides adequate information to Players regarding the collection and use of their personal data, including through a compliant privacy notice.

15.7. To the extent the Developer receives any communication, complaint, or request directly related to personal data processed by Carrot and/or its Affiliates, it shall promptly notify Carrot and/or its Affiliates and shall not respond directly unless legally required to do so.

15.8. The Developer shall, where reasonably necessary and upon request, provide limited assistance to Carrot and/or its Affiliates in relation to compliance with Articles 32 to 36 of the UK GDPR, solely to the extent such assistance does not require the Developer to access or process personal data, except as permitted in Clause 15.2.

15.9. Carrot and/or its Affiliates shall ensure that any international transfer of personal data it carries out complies with the applicable requirements of the Data Protection Legislation.

15.10. Each party shall comply with its respective obligations under the Data Protection Legislation in relation to any personal data processed in connection with this Agreement and shall be individually liable for its own acts and omissions in the course of such data processing activities.

15.11. Nothing in this clause shall limit or exclude any liability for breach of the Data Protection Legislation, to the extent that such limitation or exclusion is not permitted under law.

16. Governing Law and Jurisdiction

16.1. This Agreement and any dispute arising out of, or connected to it ("**Dispute**"), shall be governed by English law.

16.2. The Parties shall each use all reasonable endeavours to negotiate in good faith and settle amicably any Dispute. If, within five (5) Business Days of it arising, the Dispute cannot be settled amicably through ordinary negotiations of the Parties' respective representatives, each Party shall refer the Dispute to one their respective internal senior stakeholders who shall meet to attempt to resolve the Dispute. If the Dispute

cannot be resolved by such internal senior stakeholders within fourteen (14) days of it being referred to them, then the Parties agree that the courts of England and Wales will have the exclusive jurisdiction to:

(a) determine the Dispute;

(b) grant interim remedies, and

(c) grant any other provisional or protective relief.

17. Miscellaneous

17.1. **Notices**

17.2. Any notice required to be given under this Agreement will be deemed effective and validly given if sent by email to the Party to whom the notice is addressed at the email address as is set forth below.

17.3. Notice shall be considered effective on the next business day after the email is sent. In furtherance thereof, the Parties hereby acknowledge and agree that such email communications constitute electronic communications. Furthermore, the Parties agree that all notices, disclosures, and other communications provided by either Party under this Agreement via electronic communication satisfy any legal requirements that such notices, disclosures, or communications be in writing.

Carrot: support@stake-engine.com

Developer: The email address provided by the Developer during online registration.

17.4. **Anti Bribery**

a. The Parties shall:

i. comply with all Applicable Laws, statutes and regulations relating to anti-bribery and anti-corruption, including but not limited to the Bribery Act 2010;

ii. not engage in any activity, practice or conduct that would constitute an offence under sections 1, 2 or 6 of the Bribery Act 2010 if such activity, practice or conduct had been carried out in the UK;

iii. comply with the policies relating to ethics, anti-bribery and anti-corruption as Carrot may provide to the Developer and update from time to time; and

iv. in the case of the Developer, promptly report to Carrot any request or demand for any undue financial or other advantage of any kind received by the Developer in connection with the performance of this Agreement.

b. A breach of this clause shall be deemed a material breach of this Agreement.

17.5. **Severability**

a. If one or more provisions or part-provisions of this Agreement are held to be unenforceable under applicable law, then:

i. such provision or part-provision may be excluded from this Agreement;

ii. the remainder of the Agreement will be interpreted as if such provision or part-provision were so excluded; and

iii. the remainder of the Agreement will be enforceable in accordance with its terms.

b. If any provision or part-provision of this Agreement is deemed deleted under clause 17.5a), the Parties shall negotiate in good faith to agree a replacement provision that, to the greatest extent possible, achieves the intended commercial result of the original provision.

17.6. **Force Majeure**

a. Neither Party shall be liable for any costs or damages due to delay or non-performance under this Agreement arising out of a Force Majeure Event.

b. Following the occurrence of a Force Majeure Event affecting the functioning of the Game for more than sixty (60) days, either Party, at its sole discretion, may elect to terminate this Agreement and the performance of any obligations hereunder.

17.7. **Waiver**

a. No waiver of any default or breach of this Agreement by either Party shall be deemed a continuing waiver or a waiver of any other breach or default, no matter how similar. Similarly, the specification of any particular remedy in this Agreement shall not serve as a waiver of any other remedy available to either Party at law or in equity.

17.8. **Relationship**

a. Nothing in this Agreement shall be deemed or construed to constitute a partnership or joint venture between the Parties, nor to constitute either Party as the agent or the legal representative of the other Party for any reasons whatsoever.

b. Neither Party is granted any right or authority to act for, or to incur, assume or create any obligations, responsibility or liability, express or implied, on behalf of the other Party or to bind the other Party in any manner whatsoever.

17.9. **Assignment**

a. This Agreement is personal to the Developer and the Developer shall not assign, transfer, mortgage, charge, subcontract, declare a trust of or deal in any other manner with any or all of its rights and obligations

under this Agreement without Carrot's consent in writing, whose consent shall not be unreasonably delayed or withheld. However, the Developer may freely assign the right to payment under this Agreement in its sole discretion.

b. Carrot may at any time assign, transfer, mortgage, charge or deal in any other manner with any or all of its rights and obligations under this Agreement.

17.10. The Developer shall procure that any acquirer or purchaser of its business, any of the Game Rights and/or assets relating to the Game agrees to be bound by the terms of this Agreement as if it was the Developer.

17.11. **Survival**

a. Clauses 2, 4, 9, 10, 11, 12, 16, 17 and any other provisions which expressly or by implication from their nature are intended to survive termination or expiry, survive termination or expiry of this Agreement.

17.12. **Entire Agreement**

a. This Agreement, including all associated attachments and exhibits thereto, represents the entire agreement between the Parties and supersedes any prior agreements, negotiations, letters of intent, deal memos, correspondence, or communication of any kind between the Parties concerning the subject matter herein.

17.13. **Amendment**

a. Carrot reserves the right to amend this Agreement at its sole discretion by providing written notice of the amendment to the Developer. Any such amendment shall become effective upon receipt of the notice by the Developer.

17.14. **Counterparts and Electronic Execution**

a. This Agreement may be executed in counterparts with the same effect as if both Parties hereto had signed the same document. Each counterpart shall be as valid and binding as each other's counterpart and all counterparts shall be construed together and shall constitute one agreement.

b. The Parties' consent to all acts pursuant to this Agreement being done by electronic means, including but not limited to the execution of this Agreement. Exchange of electronic counterparts shall be taken to have the same effect as signatures on the same counterpart and on a single copy of this Agreement.

17.15. **Third Party Rights**

a. This Agreement does not give rise to any rights under the Contracts (Rights of Third Parties) Act 1999 to enforce any term of this Agreement.

17.16. **Insurance**

a. During this Agreement and for a period of one (1) year afterwards, the Developer shall maintain in force insurance policies with reputable insurance companies, against all risks that would normally be insured against by a prudent businessperson in connection with the risks associated with this Agreement, and produce to Carrot on demand full particulars of that insurance and the receipt for the then current premium.

By ticking this box, you confirm that you have read, understood, and agree to be bound by the terms and conditions of this Agreement.