Mothership Dice Roller Integration with Randsum

Overview

This guide outlines how to replace the custom dice logic in the **Mothership RPG Companion** with the external library **Randsum** for more robust and maintainable dice rolling. Randsum is a modern, type-safe dice roller that supports standard and complex dice notations and is published on npm [1]. The existing project uses a custom 0-based dice parser, so the integration must preserve this behaviour.

1. Install Randsum

1. Use npm (or your preferred package manager) to add Randsum to your project:

   ```
   npm install randsum
   ```

   Randsum's maintained package is `randsum`; the old `@randsum/dice` versions on UNPKG are obsolete. Avoid integrating via the `https://unpkg.com/@randsum/dice@0.2.9` CDN because it uses outdated builds, requires asynchronous `import()` calls and lacks TypeScript types.

2. Adapt `utils/dice.ts`

Create an **adapter** in `utils/dice.ts` that wraps Randsum's API and preserves your existing interface:

- **Parse the formula:** Split the dice notation (e.g., `2d100`) from any arithmetic operator (`+`, `-`, `*`, `/`).
- **Roll the dice:** Call Randsum's `roll()` function on the core notation. Randsum returns a `RollResult` object with an array of roll records and a **1-based** total [2].
- **Convert to 0-based values:** Subtract `1` from every die value to convert 1..s into 0..(s–1) and subtract the number of dice from the total. This keeps percentile rolls aligned with Mothership tables.
- **Apply arithmetic manually:** If the original string contains `+`, `-`, `*` or `/`, perform these operations on the 0-based total after rolling. Randsum's modifiers cover plus/minus but not multiplication/division [3]; therefore manual post-processing is required.
- **Return the same structure:** The function should return an object `{ total, rolls, modifier, formula }` so your React components (e.g., the floating dice roller) work unchanged.

Here is a minimal adapter skeleton:

```
import { roll as randsumRoll } from 'randsum';
import type { RollResult as RandsumResult } from 'randsum';
import type { RollResult } from '../types';

export function parseAndRoll(formula: string): RollResult {
  // Remove whitespace and normalise casing
  const cleaned = formula.replace(/\s/g, '').toLowerCase();

  // Find operator after the first 'd'
  const dIndex = cleaned.indexOf('d');
  let opIndex = -1;
  for (const op of ['+', '-', '*', '/']) {
    const i = cleaned.indexOf(op, dIndex + 1);
    if (i !== -1) {
      opIndex = i;
      break;
    }
  }

  const core = opIndex === -1 ? cleaned : cleaned.slice(0, opIndex);
  const modifierOp = opIndex === -1 ? null : (cleaned[opIndex] as '+' | '-'
| '*' | '/');
  const modifierVal = opIndex === -1 ? 0 : parseInt(cleaned.slice(opIndex +
1), 10);

  // Parse number of dice to adjust the total later
  const quantity = parseInt(core.slice(0, core.indexOf('d')), 10);

  // Roll using Randsum; result.sum is 1-based
  const result: RandsumResult = randsumRoll(core);
  // Randsum can return an array (for simple rolls) or an object
  const oneBasedTotal = Array.isArray(result)
    ? result.reduce((sum, v) => sum + v, 0)
    : result.total;
  const oneBasedValues = Array.isArray(result) ? result : result.result;

  // Convert to 0-based values
  const zeroBasedRolls = oneBasedValues.map((v: number) => v - 1);
  let total = oneBasedTotal - quantity;

  // Apply arithmetic
  let modifier = 0;
  if (modifierOp && !isNaN(modifierVal)) {
    switch (modifierOp) {
      case '+':
        total += modifierVal;
        modifier = modifierVal;
        break;
      case '-':
        total -= modifierVal;
```

```
        modifier = -modifierVal;
        break;
      case '*':
        total *= modifierVal;
        break;
      case '/':
        total = Math.floor(total / modifierVal);
        break;
    }
  }

  return {
    total,
    rolls: zeroBasedRolls,
    modifier,
    formula: cleaned,
  };
}

export const rollDice = (formula: string): number =>
parseAndRoll(formula).total;
```

This adapter keeps the synchronous API your components expect and isolates the Randsum dependency within one file.

3. Keep the API synchronous

Avoid dynamic imports from a CDN. Loading `https://unpkg.com/randsum/dist/index.mjs` with `await import()` would make `parseAndRoll()` asynchronous, forcing you to refactor the dice roller and other functions. Installing Randsum via npm keeps your functions synchronous.

4. UI components remain unchanged

Components such as `FloatingDiceRoller.tsx` call `parseAndRoll()` directly and rely on 0-based results for critical logic. Because the adapter preserves the return type, you can continue using advantage/disadvantage toggles, panic checks and roll history without any changes.

5. Notes and pitfalls

- **0-based vs 1-based:** Randsum is 1-based by design; forgetting to subtract `1` from each die will break percentile checks, panic rules and critical thresholds. Always normalise results to 0-based.
- **Mathematical modifiers:** Randsum supports only plus/minus modifiers natively; multiplication or division must be performed after the roll.
- **Do not use outdated packages:** The package `@randsum/dice@0.2.9` available via UNPKG is deprecated. Use the current `randsum` package from npm [1].

Conclusion

To integrate Randsum into the Mothership RPG Companion, install the library via npm and rewrite `utils/dice.ts` to adapt the new dice roller. The rest of the application—including UI components and business logic—remains untouched. This change replaces your custom parser with a maintained, type-safe library while preserving the 0-based dice semantics required for Mothership.

---

1 raw.githubusercontent.com
https://raw.githubusercontent.com/RANDSUM/randsum/main/packages/roller/README.md

2 raw.githubusercontent.com
https://raw.githubusercontent.com/RANDSUM/randsum/main/packages/roller/src/roll/index.ts

3 raw.githubusercontent.com
https://raw.githubusercontent.com/RANDSUM/randsum/main/packages/roller/src/types.ts