

# The Manticore

It's Care and Feeding



This is a guide to understanding the *Manticore* application for developers. It provides historical context for decisions made in the codebase, some discussion around the design of the user interface, and descriptions of program structure and algorithmic behaviour.

Additionally, I have included some notes about things I would consider changing going forward.

This document is accurate as of May 2024.

## Contents

<b><i>An overview and history of Manticore.</i></b>	<b>4</b>
<b><i>Development Environment</i></b>	<b>5</b>
<b><i>The User Interface</i></b>	<b>6</b>
Scrolling	6
Edition toggle	6
Party	6
Filters	7
“Attributes” Filters	7
Specific Monsters	7
Encounters	8
When Generation Occurs	8
Encounter Presentation	8
Responsive Layout	9
<b><i>Application Data</i></b>	<b>10</b>
Schema	10
An example monster record:	11
<b><i>Navigating the Project</i></b>	<b>12</b>
Source code organisation	12
Tests	13
<b><i>Data Processing and Representations</i></b>	<b>14</b>
A note on types and classes	14
Costs	15
MonsterScale	15
Starting an allocation	15
<b><i>Exhaustive Allocation</i></b>	<b>16</b>
allocateMonsters	16
ForkingBufferCursor	17
allocate inner function	17
<b><i>Random Allocator</i></b>	<b>18</b>
allocate	18
ShrinkableRandomAccess Collection	18
<b><i>Build</i></b>	<b>20</b>
SvelteKit	20
<b><i>Deploy</i></b>	<b>21</b>
Changing paths	21

## An Overview and History of Manticore.

*Manticore* is an purely client-side browser based application. There is no server side. A Node.JS environment is required to run the build tools but is not required for deployment.

It has been developed over the course of a decade, finally landing in a [Typescript](#) and [Svelte](#) implementation. This long history has involved a number of platform shifts that have each had some impact on the nature of the codebase.

The original core of the application was a logic program written in Clojure. Logic programs are particularly well suited to complex searches, but was rewritten in Typescript to make it a purely client side app, while porting the generation algorithm. As a result, the core algorithm is still a backtracking non-deterministic searcher.

Non-deterministic search is a key feature of *Manticore* when compared to other encounter genera-

tors: instead of generating a small set of random encounters, *Manticore* attempts to generate all possible encounters that match the search criteria. This has recently been expanded to allow the user to choose between exhaustive and random.

Exhaustive generation impacts the design of the UI and the codebase. Internally, exhaustive generation is referred to as deterministic generation. See page 16 for more.

Some design decisions in the architecture of the program date back to a time when any application delivered over the web needed to support *Internet Explorer* in some capacity. While specific support has been removed, some vestiges of that origin may remain.

Finally, the first edition rules had a test suite built around property jsverify and mocha. As the second edition update was never completed, tests have not yet been implemented for the new rules.

## Development Environment

You will need a [Node.JS](#) environment and git version control tools to work on this project.

The source code repository can be found at: <https://github.com/bre-haut/manticore.git>

npm will install all the other tooling you need:

```
$ npm install
```

The Svelte tooling has an integrated watcher/server that is launched with:

```
$ npm run dev
```

You can build a distributable artefact for production with:

```
$ npm run build
```

And run the unit tests with:

```
$ npm run test:unit
```

For more information see the project README or the Svelte and Sveltekit documentation.

## The User Interface

We'll start by looking at the user interface. The way the interface operates naturally dictates much of the way the application functions internally.

### Scrolling

One thing you might immediately note about the design of the UI is that it is one long page broken into sections, that you scroll down. You might be used to other encounter generators that have collapsable sections or disclosures to contain and hide away controls. However, scrolling is a natural idiom of the web. The design of manticore intends the user to start at the top, setting the most general details of their search, and scroll down adjusting more fine grained details as they go, and eventually landing on the results.

If at any time the user feels they need to change their criteria (maybe their search is too coarse and they are getting to many results to process, or they have fil-

tered out too many options, resulting in limited or uninteresting encounters), they can scroll up higher.

### Edition toggle

There is a toggle at the top right of the UI "*Use 13th Age 2e Alpha rules*" that switches both the generation engine, as well as some small changes to the UI. This currently defaults to 1e rules. Changes to this toggle are saved in [local storage](#).

### Party

This section sets party size and—in 2e mode—number of encounters per day. These values are stored in local storage. This section is used to calculate the budget for fitting monster allocations into.

*Party level* is also used for the most coarse grained filter in the system. Change this and you should see the counts beside tags in the following section change.

## Filters

The filtering system is the core of the UI. Filters are broken down into five categories:

- Book
- Size
- Kind
- Level
- Attributes

All these filters interact. A filter with its toggle to the left (deactivated) is turned off. Interacting with any checkbox in the filter category will cause the toggle to engage automatically.

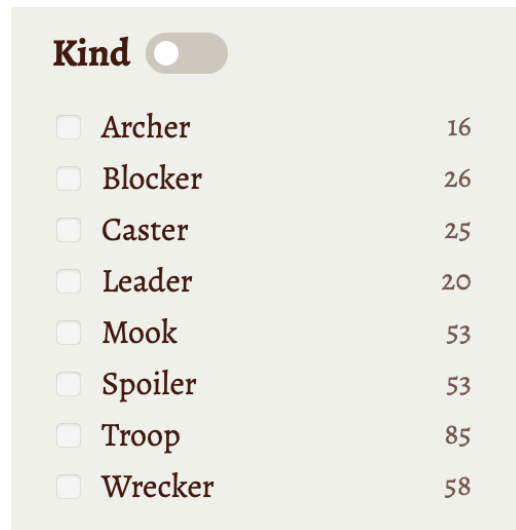
All filter checkboxes are *anded* together. For example, checking *Archer* and *Blocker* will select any monster that fits either criteria.

Each checkbox label has a small number to the right. This is the count of monsters that fit that criteria. In the example to the right, there are 16 monsters that count as Archers.

### “Attributes” Filters

While the first four filters are self explanatory and correspond to specific 13<sup>th</sup> Age concepts, *Attributes* is a *Manticore* specific term.

Attributes are just textual tags against monsters to categorise them in different ways that seemed



appropriate when I was entering the data. Some of these are fine grained, some are coarse. I've used sections of the various bestiaries as guides, and invented my own as well.

*This is one area where a more editorial eye from the 13th Age team would benefit the application.*

### Specific Monsters

This section is similar to the *Filters* section above in UI. Instead of operating on categories, the user can specify individual monsters they wish to use.

Like with the individual *Filters* this section can be skipped.

Unlike the *Filters*, this section dynamically updates with all the available monsters. As a result this section is the clearest signal to the user about how big the pool of monsters used for generation is.

## Encounters

Finally the *Encounters* section is displayed the generated encounters.

This section includes two radioboxes to switch between generating encounters *Exhaustively* or *Randomly*.

- **Exhaustively:** this is the default, and conceptually what *Manticore* was designed to do. Given the same input it always generates the same outputs.

I designed the app for this because I wanted to be able to explore the space of encounters in a predictable way, including refining and tuning the results.

- **Randomly:** this was added due to popular request. Some folks just don't like being overwhelmed with a lot of options.

## When Generation Occurs

Whenever the user makes a change to the filters the system fires off a new [WebWorker](#) to generate in the background. Prior generations are terminated if they have not com-

pleted. Once the results are available they are displayed in the *Encounters* section. Using the worker ensures we can do computationally intensive work without impacting the user experience.

Random generation necessarily requires the user to indicate that they want a reroll, as they may not find what they are after with the initial generation.

## Encounter Presentation

Encounters appear in rows. Each antagonist type is summarised with the name, the number of them in the encounter, and metadata including book and page number, and a link to the SRD if available. See below for an example.

In some instances, the generator can produce the same antagonists in different combinations. When this occurs they will be grouped together, with a single entry pulled out at the top, and a disclosure with “*N variations*” appearing immediately below. Toggling this disclosure will reveal the alternate breakdowns.

troop lvl 1 Orc warrior <a href="#">SRD ↗</a>	13th age pg. 242 ×2	troop lvl 3 Orc tusker <a href="#">SRD ↗</a>	Bestiary pg. 157 ×4
▼ <a href="#">3 variations</a>			
Orc warrior <a href="#">SRD ↗</a>	×4	Orc tusker <a href="#">SRD ↗</a>	×3
Orc warrior <a href="#">SRD ↗</a>	×6	Orc tusker <a href="#">SRD ↗</a>	×2
Orc warrior <a href="#">SRD ↗</a>	×8	Orc tusker <a href="#">SRD ↗</a>	×1



Additionally, individual antagonists that have a higher chance of out-matching the specified party will be colour coded either yellow or orange, matching the key.

## **Responsive Layout**

The user interface does have a responsive layout. This is a fairly simple implementation performed entirely with CSS rules.

*The mobile UI is a little awkward given the amount of material displayed. This would be an area worth investigating to improve usability.*

```
static > data > {} bestiary.json > ...
1  {
2    "books": {
3      "13th age": [
4        ["Giant ant",0,"normal","troop",["animal/critter","beast"], 206],
5
6        ["Decrepit skeleton",1,"normal","mook",["skeleton","undead"], 246],
7        ["Dire rat",1,"normal","mook",["animal/critter","beast","dire"], 206],
8        ["Giant scorpion",1,"normal","wrecker",["animal/critter","beast"], 206],
9        ["Goblin grunt",1,"normal","troop",["goblin","humanoid"], 229],
10       ["Goblin scum",1,"normal","mook",["goblin","humanoid"], 229],
11       ["Human thief",1,"normal","troop",["human","humanoid"], 235].
```

## Application Data

The data for the application resides in a single JSON file that is loaded when the application starts.

You can find it in the project in `static/data/bestiary.json`

The schema for this data is designed first and foremost to make reduce the amount of work for data entry. As a result it is quite terse.

*When the application was written there were no other easily accessible datasets of monster data available, so I created my own for the project.*

*It may benefit the project to reconsider how this data is handled, to see if sharing with other projects is viable, and to reduce the overall workload of people creating tools for 13<sup>th</sup> Age.*

When loaded the data is transformed into a more amenable internal format. We will discuss that later in this document.

## Schema

The top level object in the JSON file are two keys:

- **Books:** Contains an object where the keys are the names of books containing monsters, and the values are arrays of monster records. See below.
- **srdReferences:** Contains an object where the keys are monster names. The values are URLs to SRD material about the monster in question. The names need to be an exact textual match to the monster name defined in **Books**.

The individual monster records are arrays, purely to reduce the repetitive typing that using objects would require. There are over 700 monsters in the current dataset, and it doesn't include books like *13<sup>th</sup> Age in Glorantha*.

Monster records have the following values in order:

- 1. Name:** a string. This is what is displayed to the user.
- 2. Level:** a number.
- 3. Size:** a string. This needs to be one of the following:
  - "normal"
  - "large"
  - "huge"
  - "weakling"
  - "elite"
  - "double strength"
  - "triple strength"
  - "large elite"
- 4. Kind:** a string. There is no strict definition for Kinds.
- 5. Attributes:** an array of strings. Each string in the array is a tag for filtering this monster. See *Attributes*, page 7
- 6. Page number:** a number.

### An example monster record:

```
["Gnoll war leader", 4, "normal","leader",["gnoll","humanoid"], 229],
```

Name	Level	Size	Kind	Attributes	Page
------	-------	------	------	------------	------

*To facilitate fast data entry, [investigate a project snippet in VS Code](#) so that you can template out the common markup. This would also make it viable to switch from the array based syntax here to a JSON object syntax without burdening anyone creating new records.*

## Navigating the Project

Overall the project source files are organised very simply. At the top level, aside from all the NPM and Svelte Kit bumpf, there are three folders of note:

- **src:** This contains all the Typescript and Svelte files. This is covered in the next section
- **static:** All the static assets for the project are found here, *most significantly this includes the data files*, but also the fonts, images, and icons.
- **Build:** This folder wont exist on first checkout but is created when you run a build. This will contain all the assets ready to deploy, including compiled versions of all the Typescript and Svelte code, and all the static data from the static folder.

### Source code organisation

Essentially all the code for the application lives inside the lib folder.

The routes folder contains the svelte entry-point (`+page.svelte`) and the basic markup for setting up the application in `+layout.svelte`

Notable files at the top level are the global stylesheet, and the remaining tests that perform an integrity check on the bestiary data.

### The Lib Folder

Within lib are three key folders, and some toplevel code:

- **allocator:** This is the core of the generation logic. Both the exhaustive and random allocators are found here. These will be covered in more detail later.
- **costs:** This code produces a cost for a monster based on the rules provided in the

13<sup>th</sup> Age core rules. This code also compartmentalises the differences between editions of the game. Outside of this, nothing else in the model layer of the application is aware of the distinction.

- ui: All the svelte code is contained within this folder. Unsurprisingly, it's all the code for the user interface.

Also within the lib folder are some core files for handling data, representing a bestiary, and supporting utility code such as functions for operating on iterators (`iter.ts`, `clusterItems.ts`).

These will be covered in the next section, Data Processing and Representation.

## Tests

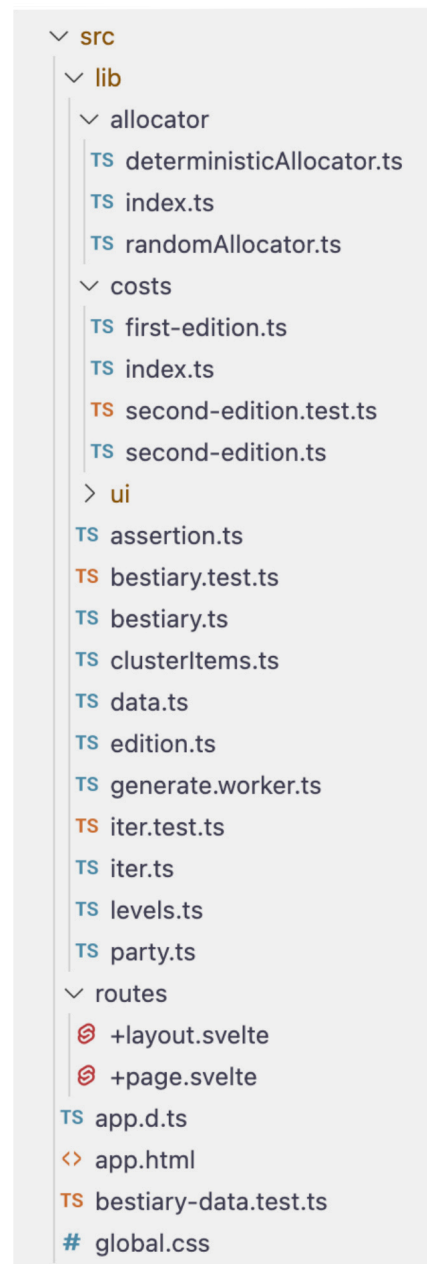
The majority of the test suite exists here in `*.test.ts` files. These are largely tests from the V1 codebase, and don't provide great coverage for second edition specific code.

Please note that the tests in `bestiary.test.ts` use a style of testing called [Property Based Testing](#). The short version is that the tests produce random known-good inputs, and then programatically check the output of the code under tests against those inputs, rather than testing specific examples.

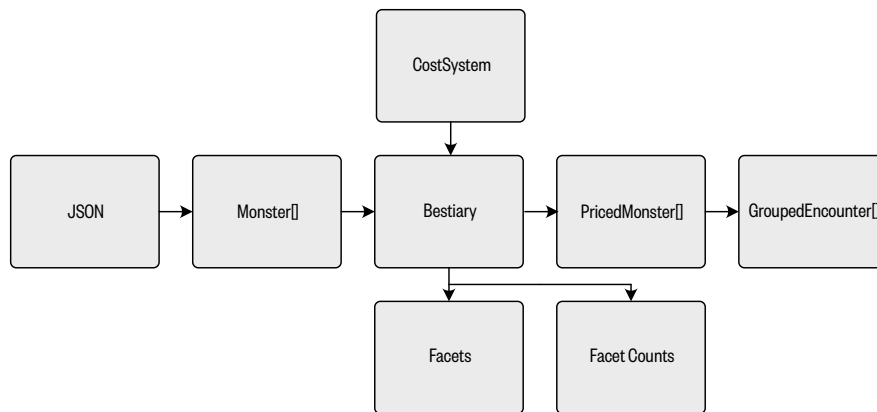
This lets us have greater confidence that the tests cover more surface area of the code under test than exemplar based testing could

achieve. As a result there are less tests that you might expect to see.

The [jsverify](#) library is used to support property based testing.



```
src
├── lib
│   ├── allocator
│   │   ├── deterministicAllocator.ts
│   │   ├── index.ts
│   │   └── randomAllocator.ts
│   ├── costs
│   │   ├── first-edition.ts
│   │   ├── index.ts
│   │   ├── second-edition.test.ts
│   │   └── second-edition.ts
│   └── ui
│       ├── assertion.ts
│       ├── bestiary.test.ts
│       ├── bestiary.ts
│       ├── clusterItems.ts
│       ├── data.ts
│       ├── edition.ts
│       ├── generate.worker.ts
│       ├── iter.test.ts
│       ├── iter.ts
│       ├── levels.ts
│       └── party.ts
├── routes
│   ├── +layout.svelte
│   └── +page.svelte
├── app.d.ts
├── app.html
├── bestiary-data.test.ts
└── global.css
```



## Data Processing and Representations

Loading and processing the bestiary data file begins in `src/ui/App.svelte` in the `onMount` handler.

The JSON file is fetched and the blob is passed into the `createBestiary` function, along with the edition. The edition is used to determine the current `CostSystem`.

`createBestiary` assumes that the JSON blob is a `DataSet` type (defined in `data.ts`). This assumption is

checked using the `bestiary_data.tests.ts` file so should be reliable.

The major piece of work prior to instantiating a new `Bestiary` object is transforming the monster records (see page 11), into objects with the `Monster` type. This process includes normalising *scale*, which takes into account *size* and *threat*, and is used as part of the cost calculation.

A `Bestiary` holds both the collection of `Monsters`, and also a `CostSystem`. It's

---

### A note on types and classes

Much of the manticore codebase uses Typescript *types* and plain javascript objects rather than classes. This is generally the case for all the immutable data carry records such as a `Monster`. Classes, such as `Bestiary`, are used for structural and behavioural objects.

In almost all cases constructors are never exported from modules, and wrapper functions are provided instead.

Where possible all types are immutable, with code written in a functional style transforming one immutable record into another.

role is to provide the UI with meta-data about the available monsters, and calculate counts for various facets as filters change, and secondly to produce a collection of costed monsters ready for the allocator algorithm to use.

The allocator is given the collection of priced monsters, and the party details to determine a total budget. The details of the allocators is described later.

Finally, the allocator returns a collection of `GroupedEncounters`. A `GroupedEncounter` is a collection of `Encounters`, and each `Encounter` is an array of `Allocations`, and a record of any percentage of the budget left unspent. An `Allocation` is a *Monster*, the number of that monster in this encounter, and a total cost for the `Allocation`. Total cost is the monster's cost × the number in this allocation. A lot of wrapping types here but all very simple.

The allocations are returned to the UI to display to the user.

## Costs

The specific details of costing differs between first and second edition. The first edition code is a little more opaque just because the first edition didn't have as much clarity around the building battles rules.

In both editions the values for costs are multiplied up from the values in the books to ensure that all calculations use integer values. While JavaScript number values are tech-

nically always floats the engines do ensure that when a number looks like an integer, it acts like one.

Cost systems have a method to determine what threshold of unspent points are allowed. Currently this is always 0: the entire budget must be spent to be considered viable.

## MonsterScale

Internally *Manticore* uses *Scale* to compute costs. *Scale* is a tuple of the normalised *Size* (Double Sized is equivalent to Large for the purposes of costing), and normalised *Threat* (mook, weakling, normal, elite).

## Starting an allocation

`Results.svelte` is responsible for starting generation and displaying the results. The function `generate` is called whenever the selected monsters change. This function manages a `WebWorker`. If a change to the selection occurs before the a generation has completed the prior `WebWorker` is terminated. `generate` also has some heuristics to minimize a flash of content for deterministic generation.

## GenerationWorker

`Svelte` wraps up workers using a special module loading path format:

```
import GenerateWorker from "$lib/generate.worker.ts?worker";
```

The imported class can then be instantiated as normal, and can be treated as a `Worker`.

## Exhaustive Allocation

The exhaustive allocator is launched from `src/generate.worker.ts`.

This allocator is referred to as the deterministic allocator within the codebase, because it always produces the same results for a given input set. This is in contrast to the random allocator which obviously does not.

You can find the deterministic allocator in `src/allocator/deterministicAllocator.ts`.

The two key pieces of code to understand this allocator are the function `allocateMonsters` and the class `ForkingBufferCursor`.

### **allocateMonsters**

This function is a [backtracking depth first search](#). Backtracking is implemented using the `ForkingBufferCursor`, and a recursive function, `allocate`, within `allocateMonsters`.

Note that `allocate` is a [generator function](#), meaning it yields out results one by one rather than building up a list to return. This allows the function to be more memory efficient by not building a lot of short lived structures.

Aside from the inner function, the body of `allocateMonsters` just sets everything up for the first call into `allocate`, and limited the amount of items generated if the search space turns out to be massive (which is trivially easy to do).

### **A functional programming digression.**

The core of this algorithm is written in a functional style. For the purposes of this discussion that means that it never mutates any values, instead creating new values for the next step. This is important for the backtracking as it means we can leverage the call stack for state management and know that when we return from a recursive call, we



are backtracking to a previous state of the program.

The recursive function tracks a variable called `acc`, the accumulator variable, which is a JavaScript Array. `acc` is cloned with the [slice\(\)](#) method to get new copies of the accumulator to pass to child calls.

## ForkingBufferCursor

This class is a simple cursor object that wraps up an Array, and allows its consumer to:

- Examine the current item with `value()`
- Advance one item at a time with `next()`
- Determine if the cursor has walked the entire buffer with `done()`
- Or fork off another `ForkingBufferCursor` from the current location with either `fork()` or `forkAndNext()`

## allocate inner function

The real work of generation occurs here.

The first step of this function is to determine if this branch of the search has reached its maximum depth, and if so to return. Remember that this function is called recursively.

There are three criteria for reaching the end of a search branch:

- The budget has been spent: Yield results.
- The maximum number of monster types has been reached: The system is capped at 7 (possibly too high a number), and this branch needs to be terminated because it would result in an unwieldy encounter.
- We've run out of monsters to allocate and this branch needs to be terminated.

Secondly we need to repeat the current monster to fill out an allocation. The generator function `repeatMonster` produces a stream of allocations from one up to the maximum possible with the remaining budget. Note that `repeatMonster` has a special case for managing mooks in second edition.

Before we yield the results of `repeatMonster`, the function recurses. In doing so, it searches the branch that does not include any allocations for this monster. The results of the recursive call to `allocate` are yielded into the current scopes output.

Finally we look through each allocation generated by `repeatMonster`, and recurse again with each variation of the allocation, yielding the results of those recursive calls into the current scopes output.

## Random Allocator

The random allocator is significantly simpler than the deterministic allocator.

The core of this algorithm is the `ShrinkableRandomAccessCollection`, a class that wraps up an array and allows random selection up to a maximum “weight”, and an `allocate` function that produces a number of allocations, in this case randomly.

### **allocate**

Like the deterministic allocator, the algorithm is implemented in an `allocate` function. Rather than needing recursive inner functions we simply have two nested while loops:

The outer loop runs until a circuit breaker is hit due too excessive of failures is reached, or the maximum allocations occurs. The circuit breaker value is currently 100. This value is relatively arbitrary; it seems to work, but theres not a lot of science behind it.

The inner loop is builds a single allocation, looping until it spent its budget, or it runs out of monsters to pull from. Each allocation updates the shrinkable collection's maximum weight.

Before an allocation is yielded, there is a call to the `collate` function that groups like monsters together. This step could be rolled into the allocation (see the TODO), but it seemed simpler to do it as a separate step later.

### **ShrinkableRandomAccess Collection**

As mentioned previously, this is the core of this algorithm.

When instantiated (via a static method; the constructor itself is private), it takes a collection of objects, and a `getWeight` function. For the purposes of this application the weight is always the cost of the monster in question.

The weight is used to sort the collection of monsters most costly to least costly. A field called the front tracks where access is allowed to start from

arithmetic on `aWeight` and `bWeight`. This is true, but the resulting code is harder to reason about, so the fully spelt out version is used instead.

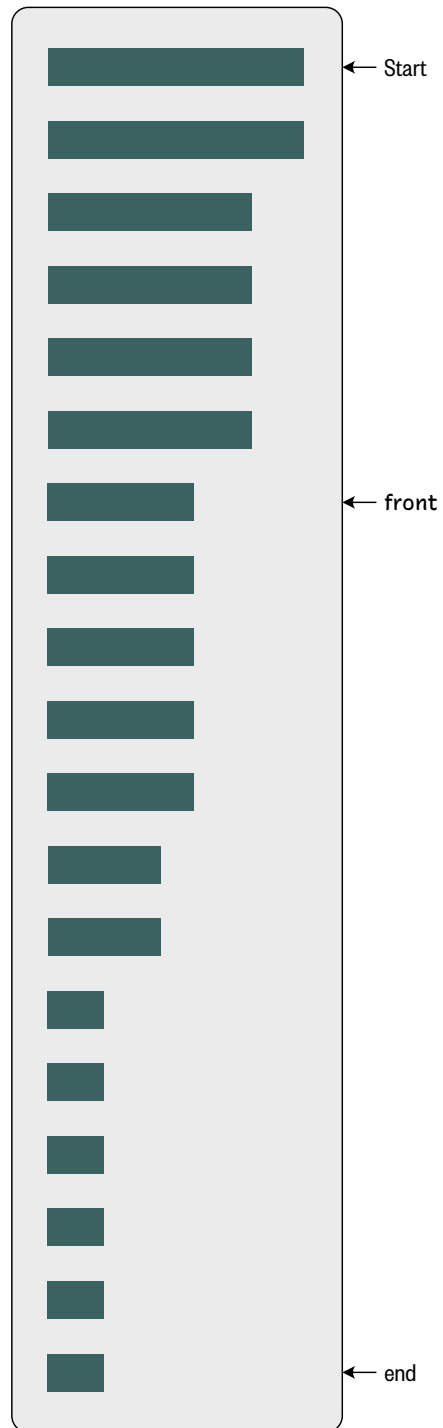
This allows the consumer to:

- `getRandom()` to randomly choose an index between that `front` and the end of the collection, returning the item at that index.
- `setMaxWeight(newMax)` to produce a new `ShrinkableRandomAccessCollection` that has the new front value calculated by walking the current front forward until it finds an item that has a weight that does not exceed the `newMax`.
- `fork()` the collection to get a new `ShrinkableRandomAccessCollection` that has the current front and `max`.
- `done()` to determine if the front has walked beyond the end of the collection.

This means that we never fetch an item that has a larger weight than the max, due to the ordering of the internal collection.

**A note about the `getWeightComparator` comparator**

Experienced developers may see the implementation of this comparator as clumsy and inefficient as it could be implemented with simple



## Build

Building *Manticore* is straight forward:

```
$ npm run build
```

This will build all the typescript, all the svelte, and manage all the static assets.

The resulting hive of files will be created in the `build` folder in the root of the project.

## SvelteKit

The build for this project is managed by [Svelte Kit](#).

The [static adaptor](#) is used to make a static site single page application.

## Deploy

Currently Manticore expects to be in the top level of the domain its hosted in. Which is to say that (for example) `build/index.html` needs to be in the root folder of the domain (such as the former canonical domain of `manticore.brehaut.net`).

With this caveat out of the way, deployment of *Manticore* is trivial: The hive of files in `build` needs to be placed in a web server.

As an example, I have a script that does:

```
#!/bin/sh
rsync -av build/* [user]@[host]:[web-server-path]
```

Done.

## Changing paths

If you don't wish to have *Manticore* hosted in the root of the domain it's hosted on, you will need to consult the Svelte and Svelte Kit documentation.