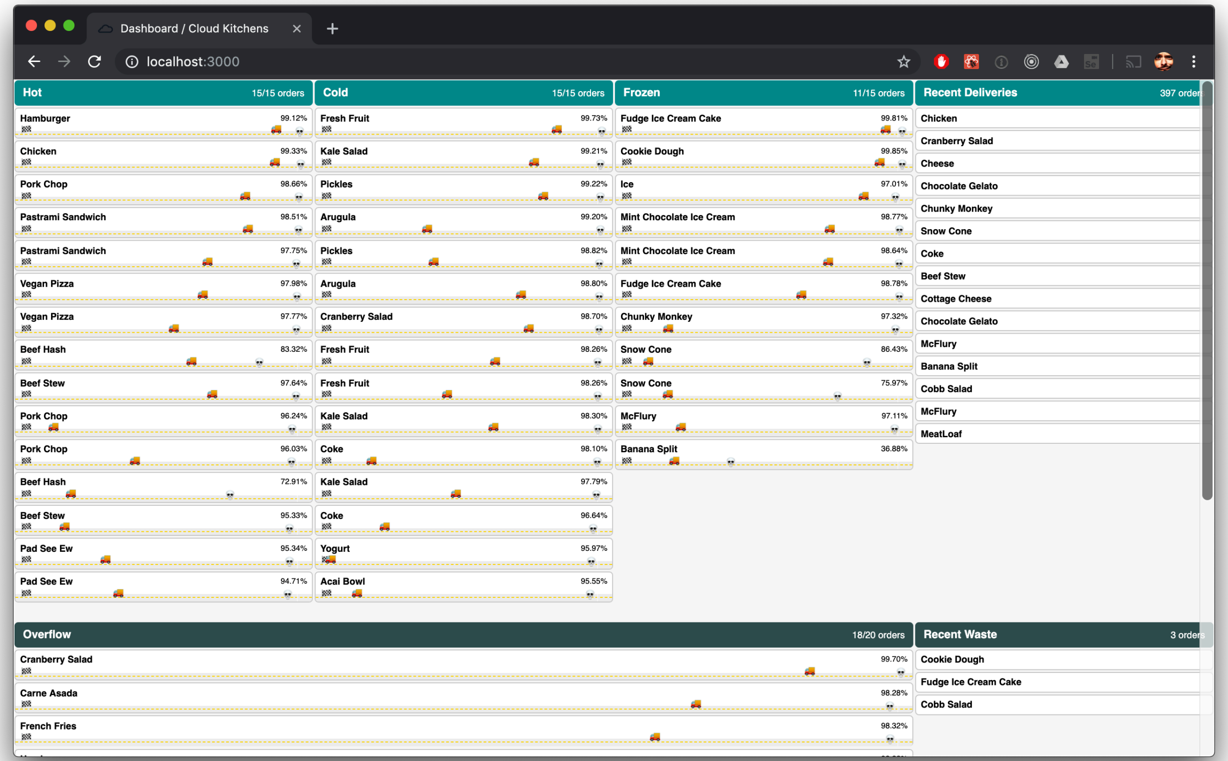
Cloud Kitchens

Prepared by David Furfero.


Dashboard / Cloud Kitchens

localhost:3000

Hot 15/15 orders

Hamburger 99.12%
Chicken 99.33%
Pork Chop 98.66%
Pastrami Sandwich 98.51%
Pastrami Sandwich 97.75%
Vegan Pizza 97.98%
Vegan Pizza 97.77%
Beef Hash 83.32%
Beef Stew 97.64%
Pork Chop 96.24%
Pork Chop 96.03%
Beef Hash 72.91%
Beef Stew 95.33%
Pad See Ew 95.34%
Pad See Ew 94.71%

Cold 15/15 orders

Fresh Fruit 99.73%
Kale Salad 99.21%
Pickles 99.22%
Arugula 99.20%
Pickles 98.82%
Arugula 98.80%
Cranberry Salad 98.70%
Fresh Fruit 98.26%
Fresh Fruit 98.26%
Kale Salad 98.30%
Coke 98.10%
Kale Salad 97.79%
Coke 96.64%
Yogurt 95.97%
Acai Bowl 95.55%

Frozen 11/15 orders

Fudge Ice Cream Cake 99.81%
Cookie Dough 99.85%
Ice 97.01%
Mint Chocolate Ice Cream 98.77%
Mint Chocolate Ice Cream 98.64%
Fudge Ice Cream Cake 98.78%
Chunky Monkey 97.32%
Snow Cone 86.43%
Snow Cone 75.97%
McFlury 97.11%
Banana Split 36.88%

Recent Deliveries 397 orders

Chicken
Cranberry Salad
Cheese
Chocolate Gelato
Chunky Monkey
Snow Cone
Coke
Beef Stew
Cottage Cheese
Chocolate Gelato
McFlury
Banana Split
Cobb Salad
McFlury
MeatLoaf

Overflow 18/20 orders

Cranberry Salad 99.70%
Carne Asada 98.28%
French Fries 98.32%

Recent Waste 3 orders

Cookie Dough
Fudge Ice Cream Cake
Cobb Salad

Cloud Kitchens Dashboard for monitoring order simulation.

Getting Started

Install the project dependencies.

```
$ cd cloud-kitchens
$ npm install
```

Running the Demo

1. Start the API server.

   ```
   $ npm run api
   ```

2. In a separate terminal session, start the web server.

   ```
   $ npm run www
   ```

3. Open a web browser and navigate to the Dashboard at http://localhost:3000.
4. To begin the ordering, open another browser and navigate to http://localhost:3000/orders. The orders will begin streaming immediately.

Order More

To order more, reload the orders web page or open an additional window.

Reset the Data

To clear the completed queue and all executing orders, stop and restart the API server.

Running the Tests

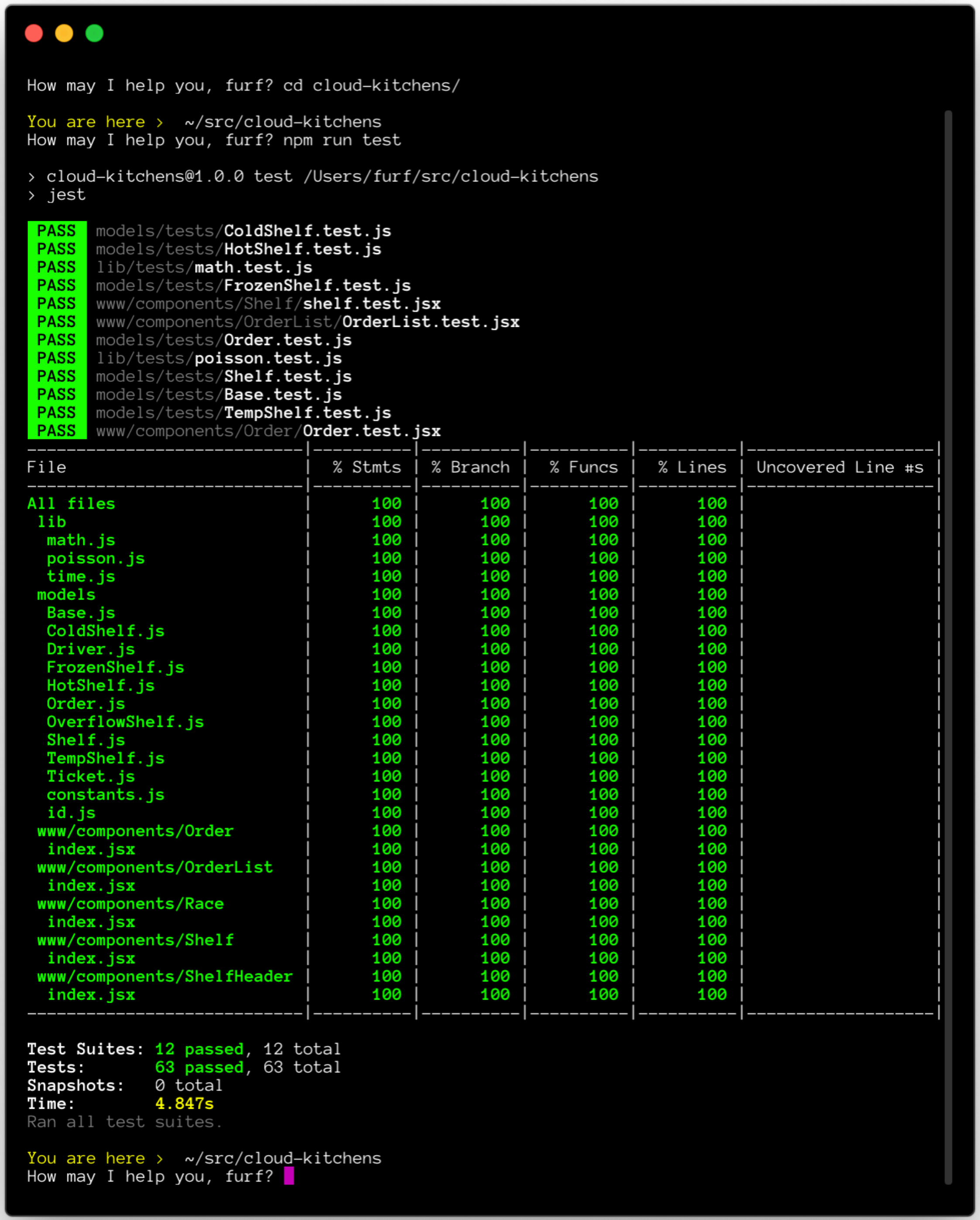
All tests, including linting (ESLint and Prettier) will run before every commit using husky for assigning npm scripts to Git hooks.

If you want to run the tests manually, you can do so via npm script.

```
$ npm run test
```

A coverage report is generated with each test.

Test coverage is incomplete, but we're off to a good start.



```
How may I help you, furf? cd cloud-kitchens/

You are here ›  ~/src/cloud-kitchens
How may I help you, furf? npm run test

> cloud-kitchens@1.0.0 test /Users/furf/src/cloud-kitchens
> jest

 PASS  models/tests/ColdShelf.test.js
 PASS  models/tests/HotShelf.test.js
 PASS  lib/tests/math.test.js
 PASS  models/tests/FrozenShelf.test.js
 PASS  www/components/Shelf/shelf.test.jsx
 PASS  www/components/OrderList/OrderList.test.jsx
 PASS  models/tests/Order.test.js
 PASS  lib/tests/poisson.test.js
 PASS  models/tests/Shelf.test.js
 PASS  models/tests/Base.test.js
 PASS  models/tests/TempShelf.test.js
 PASS  www/components/Order/Order.test.jsx
----------------------------|----------|----------|----------|----------|-------------------|
File                        |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------------------|----------|----------|----------|----------|-------------------|
All files                   |      100 |      100 |      100 |      100 |                   |
 lib                        |      100 |      100 |      100 |      100 |                   |
  math.js                   |      100 |      100 |      100 |      100 |                   |
  poisson.js                |      100 |      100 |      100 |      100 |                   |
  time.js                   |      100 |      100 |      100 |      100 |                   |
 models                     |      100 |      100 |      100 |      100 |                   |
  Base.js                   |      100 |      100 |      100 |      100 |                   |
  ColdShelf.js              |      100 |      100 |      100 |      100 |                   |
  Driver.js                 |      100 |      100 |      100 |      100 |                   |
  FrozenShelf.js            |      100 |      100 |      100 |      100 |                   |
  HotShelf.js               |      100 |      100 |      100 |      100 |                   |
  Order.js                  |      100 |      100 |      100 |      100 |                   |
  OverflowShelf.js          |      100 |      100 |      100 |      100 |                   |
  Shelf.js                  |      100 |      100 |      100 |      100 |                   |
  TempShelf.js              |      100 |      100 |      100 |      100 |                   |
  Ticket.js                 |      100 |      100 |      100 |      100 |                   |
  constants.js              |      100 |      100 |      100 |      100 |                   |
  id.js                     |      100 |      100 |      100 |      100 |                   |
 www/components/Order       |      100 |      100 |      100 |      100 |                   |
  index.jsx                 |      100 |      100 |      100 |      100 |                   |
 www/components/OrderList   |      100 |      100 |      100 |      100 |                   |
  index.jsx                 |      100 |      100 |      100 |      100 |                   |
 www/components/Race        |      100 |      100 |      100 |      100 |                   |
  index.jsx                 |      100 |      100 |      100 |      100 |                   |
 www/components/Shelf       |      100 |      100 |      100 |      100 |                   |
  index.jsx                 |      100 |      100 |      100 |      100 |                   |
 www/components/ShelfHeader |      100 |      100 |      100 |      100 |                   |
  index.jsx                 |      100 |      100 |      100 |      100 |                   |
----------------------------|----------|----------|----------|----------|-------------------|

Test Suites: 12 passed, 12 total
Tests:       63 passed, 63 total
Snapshots:   0 total
Time:        4.847s
Ran all test suites.

You are here ›  ~/src/cloud-kitchens
How may I help you, furf?
```

Jest test coverage report

Testing During Development

During development, you can have tests run whenever tests or covered code are updated using the `test:watch` command. (Only the necessary tests will run on update.)

```
$ npm run test:watch
```

Installing Watchman

To run `test:watch`, you will need to have Watchman installed. (This will also require brew.)

```
$ brew update
$ brew install watchman
```

Ensuring Code Quality and Consistency

We also use ESLint to ensure code quality and Prettier to ensure consistent formatting. These tools are configured to run cooperatively and avoid overlapping checks. ESLint is configured to follow Airbnb's popular JavaScript style guide.

You can lint the code base manually using the `lint` command.

```
$ npm run lint
```

You can quickly fix simple lint errors using the `lint:fix` command.

```
$ npm run lint:fix
```

Generating Documentation

The project is documented using JSDoc. Documentation is generated using the pre-commit Git hook, and it can be generated manually using the `docs` command.

```
$ npm run docs
```

You can also preview the documentation using the `docs:read` command. This command will generate the documents, start an HTTP server, and open your web browser to http://localhost:3001.

```
$ npm run docs:read
```

About the Code

Servers

The project consists of two servers: an Apollo GraphQL API server and a Next.js web server.

The API code is written in Node.js. The web server is written in React, React Apollo (data) and Less (CSS).

The Dashboard updates in real-time via GraphQL subscriptions to the server. The Orders page posts orders to the API via GraphQL mutations.

The server operates an in-memory cache of all orders. (If I have time in the next few hours, I will finish the persistent store (MySQL) capability. The data hydration on server restart will just have to wait for another time.)

The entire project, including the persistent store, is capable of being run in a serverless environment (AWS, Now, etc.).

Shelf Life Derivation

Every time an order is moved to a shelf, a ticket is created containing a reference to the shelf and a time stamp. This allows us to accurately track decay as an order moves between shelves of differing decay factors.

```
get shelfLife() {
  // Compute the order's current shelf life by iterating through its history
  // and reducing the accumulated shelf life by the decay incurred on each
  // shelf.
  const shelfLife = this.tickets_.reduce((shelfLife, ticket, index, tickets) => {
    // Calculate ticket age by comparing its timestamp to the next ticket.
    const next = tickets[index + 1];
    const endedAt = next ? next.createdAt : now();
    const age = endedAt - ticket.createdAt;

    // Decay rate is the product of the order's decay rate and the shelf's
    // decay factor.
    const decayRate = this.decayRate * ticket.shelf.decayFactor;

    // Decay and remaining shelf life are derived using the following
    // provided formula:
    // value = (shelfLife - orderAge) - (decayRate * orderAge)
    // Which can be simplified to:
    // value = shelfLife - ((decayRate + 1) * orderAge)
    const decay = (decayRate + 1) * age;
    return shelfLife - decay;
  }, this.shelfLife_);
  return Math.max(0, shelfLife);
}
```

Order Prioritization

With each expedite call, all healthy orders are prioritized for shelf placement by the likelihood that they will expire before their driver arrives. At-risk orders are given higher priority and placement on the Hot, Cold, and Frozen shelves.

The following algorithm favors orders nearing expiration and/or having slower drivers:

/models/Order.js

```
get priority() {
  const timeToLive = this.shelfLife / (this.decayRate + 1);
  const driverMargin = timeToLive - this.driver.timeToArrival;
  const priority = timeToLive * driverMargin;
  return priority;
}
```

Orders with negative `driverMargin` (and therefore negative `priority`) are discarded, even if they are healthy (shelf life > 0), because they will expire before their driver arrives. (It does not make sense to occupy shelf space with orders that will not be successfully delivered.)

/models/Expeditor.js

```
// Map objects to cache the computed `priority` property before sorting in
// order to minimize the O-complexity to O(n).
const priorities = this.orders.map(order => {
  return {
    priority: order.priority,
    order,
  };
}).sort((a, b) => a.priority - b.priority);
```

Once prioritized, each order is placed on the most favorable shelf. Shelves validate orders and return false if an order is invalid. They are also aware of their available capacity and will return false if at capacity.

```
get availableCapacity() {
  return this.capacity - this.orders.length;
}

get hasAvailability() {
  return this.availableCapacity > 0;
}

isValidOrder({ priority }) {
  return priority > 0;
}
```

```
add(order) {
  if (!this.hasAvailability || !this.isValidOrder(order)) {
    return null;
  }
  order.shelf = this;
  this.orders.push(order);
  return order;
}
```

> Note: I'm aware of the potential memory leak due to the two-way binding above. It will be remedied when the MySQL support is added and expired orders are purged from the in-memory cache.

The "Delivery Window" and "Waste Bin" are treated as shelves, allowing the following cascade to handle all order placement:

/models/Expeditor.js

```
const orders = priorities.map(({ order }) =>
  this.shelves[DELIVERY].add(order) ||
  this.shelves[order.temp].add(order) ||
  this.shelves[OVERFLOW].add(order) ||
  this.shelves[WASTE].add(order)
);
```

Poisson Randomizer

The orders are placed at 1-second intervals using a custom random number generator which returns a number of orders based on its probability within a Poisson distribution of a given lambda.

The randomizer is built by generating an array of Poisson probabilities whose indexes are the respective number of events. These probabilities, which sum to 1, are mapped to an array of cumulative sums.

```
getCumulativeSums([0.1, 0.3, 0.4, 0.2]) // [0.1, 0.4, 0.8, 1.0]
```

This gives us proportional distribution within a range of 0–1 which can be iterated and compared to a random number, also 0–1, to determine with accurate probability the number of events (the array index) to trigger each second.

Model Class Hierarchy

- `Base`
  - `Shelf`
    - `TempShelf`

            - `ColdShelf`
            - `FrozenShelf`
            - `HotShelf`
            - `OverflowShelf`
        - `WasteBin`
        - `Driver`
    - `Expeditor`
    - `Order`
    - `Ticket`

If I Had More Time

- More documentation! More tests! I have tried to give a decent indication of how I document and test my work, however time limitations prevent me from providing complete coverage.
- Rewrite the entire project in TypeScript. Given my time constraints, I wrote in JS where I'm more nimble. Otherwise, I'm beginning to write all of my side projects in TS and it's been a game-changer.
- Complete the persistent data store that I started writing using Serverless MySQL. This provides monitoring and management of a large number of concurrent database connections.
- Abstract out the `Driver` class further to support a finite driver pool.

PS—The animations on the Dashboard only took about an hour and a half. They're not to blame!