



Musketeers - Production Lifecycle Manager (Developer Guide)

Developers: Walker Eacho, Chad Coviell, Harry Guo

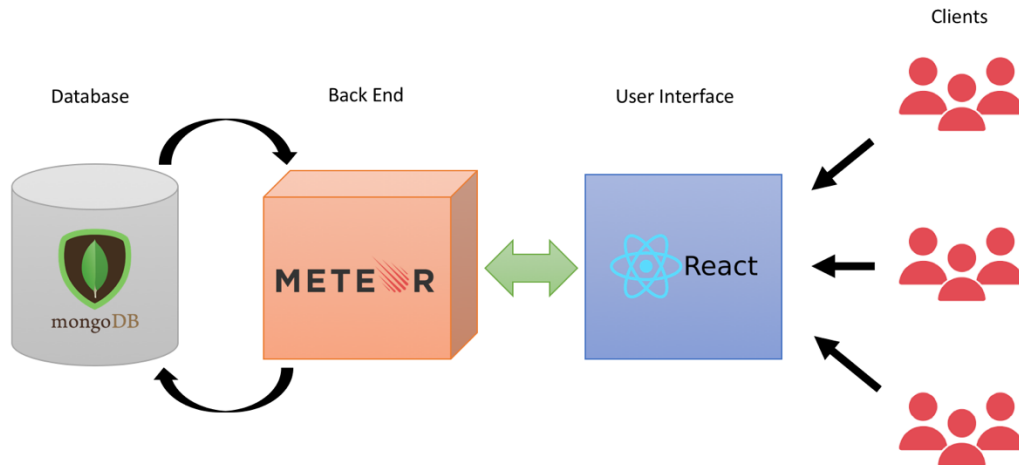
OVERVIEW.....	2
HIGH LEVEL DESIGN.....	2
TECHNOLOGIES IN USE	3
MONGODB.....	3
METEOR	3
REACT	3
CONFIGURING A DEVELOPMENT/BUILD ENVIRONMENT.....	4
INSTALLATION	4
ENVIRONMENT SETUP	4
DATABASE SCHEMA	5
USERS	6
ROLES	6
INGREDIENTS	7
VENDORS.....	7
STORAGE CAPACITIES	8
FINAL PRODUCTS	8
INTERMEDIATES	8
SPENDING REPORT	9
PRODUCTION REPORT	9
PRODUCTION HISTORY	10
FRESHNESS REPORT	10
RECALL REPORT	10
CARTS	11
LOG.....	11

Overview

Production Lifecycle Manager is a tool that provides clients a way to manage food production, including inventory, formula production, logistics, sales, etc. It allows for complete management of the ingredients and allows for production of food products. The system tracks all the ingredients the client owns and provides additional information such as lot numbers, cost information, storage information, etc. The application also tracks the vendors which supply the stock. There are three degrees of authorization to the program: (1) administrator (2) manager (3) unprivileged user. The administrators can perform day-to-day tasks while also given the ability to override system rules direction to correct data and perform other core configuration operations. The manager can order ingredients and perform production runs. The unprivileged user is given read-only access to our system. It is a cutting-edge piece of technology designed to put the old days of spreadsheets and macros to waste and gives the clients total control over their inventory.

High Level Design

At a high level, our design is separated into three parts: (1) database (2) back-end and (3) user interface. We use NoSQL MongoDB as our database, Meteor as our back-end web framework, and React as our front end. The diagram can be seen below:



At the core of our design, clients will interact our software through a simple and user-friendly interface that is easy to understand and navigate. Through connections between our database and server, the user will instantly see the changes they implement on the application.

The connections made between the different key sectors are through event handling and listeners. When the user makes interacts with the user interface, the “events” that occur trigger methods in our Meteor layer. Our Meteor layer interacts with our database layer through a

[publish-subscribe pattern](#). Finally, through listeners, the changes made within our database is reflected in our user interface.

Technologies In Use

This section talks about the main technologies and the features we utilize to bring our application to life.

MongoDB

Our database of choice is [MongoDB](#). MongoDB is an open-source, cross-platform, document-oriented database program that runs NoSQL. Within each database, the documents are JSON-like with schemas. Meteor and Mongo run together fairly well and there is plenty of documentation to get these two stacks to function together. Through the publish-subscribe pattern, databases are “published” and certain classes can “subscribe” to these databases to access them. Synchronization code does not need to be written as the changes propagate immediately. Through this pattern, we can also ensure that any feature of the client does NOT have direct access to the database, and that all database handling is on the server side.

Meteor

Meteor, or otherwise known as Meteor JS is a free, open-source, isomorphic, JavaScript web framework written using Node.js. It integrates with MongoDB well, and also with the React library. Meteor allows for rapid prototyping and produces cross platform code (Android, iOS, Web). Functionality on this back-end layer is done through Meteor methods. Meteor methods are Meteor’s remote procedure call system to write to the database. Essentially, a method is an API endpoint for the server and is tightly integrated with the publish-subscribe pattern and data loading. This also integrates with UI framework React (see next).

React

React, or ReactJS, is a JavaScript library for building user interfaces. It is open-source and maintained by Facebook. React works in a very modular way, in which the way we view our application is no longer *whole*, but rather a sum of components. That is the basis for React. You build components with different functionality and then you put it all together, including components within components. Components can be used repeatedly, and each component can change without affecting the other components on the application. This allows developers to create web-applications that use data and change without ever reloading the page. It has a lot of support from the open source community, as well as plenty of libraries to streamline development.

At the core of React is the **render()** method. The “rendering” of a component is written in HTML, and is what you see on the screen. The rest of the class is written in JavaScript and provides functionality to the component. Each component also has default [React methods](#) that are “lifecycle methods” that you can override to run code at particular times in the process.

Configuring a Development/Build Environment

Installation

To install Meteor, click the link [here](#) and run the command to install the latest version of Meteor for your operating system.

Environment Setup

A great tutorial of Meteor with React can be found [here](#).

But in the meantime, feel free to follow the steps below to set up your environment.

1) Create base application

Go to your terminal and type:

```
meteor create my-app
```

This creates a new folder called `my-app` that contains all of the files that a Meteor app needs.

```
client/main.js      # a JavaScript entry point loaded on the client
client/main.html    # an HTML file that defines view templates
client/main.css     # a CSS file to define your app's styles
server/main.js      # a JavaScript entry point loaded on the server
package.json        # a control file for installing NPM packages
package-lock.json   # Describes the NPM dependency tree
.meteor             # internal Meteor files
.gitignore          # a control file for git
```

To run your app, cd into your folder `my-app` and type `meteor`.

This will launch your application on `http://localhost:3000`.

You can mess around with this starter app in the meantime and see how the boilerplate code works between client and server. When you save files that you edit in your code editor, the page in the browser will automatically update with the new content. This is Meteor’s “hot code

push” and makes changes available to see immediately without having to relaunch your application.

2) Install React

Go to your terminal and type:

```
meteor npm install --save react react-dom
```

Congrats! You have now installed React for your Meteor application and are ready to render components. Once again, would highly recommend going through the [Meteor with React Tutorial](#) to see how everything works. It outlines how to create a collection (database), event handling, user accounts, security, Meteor methods, the publish-subscribe pattern, and testing.

Another good resource to look at to look at [Pup](#), a Meteor with React web application boilerplate. It is a full web application with basic features and plenty of documentation to help you navigate the basics of creating a full-scale application.

Database Schema

The most important feature that our entire revolves around are the databases. Anything that is done on the client side corresponds to a change in our databases. This section will describe how each of the databases are configured and how they are utilized.

Databases:

- Users
- Roles
- Ingredients
- Vendors
- Storage Capacities
- Final Products
- Intermediates
- Carts
- Lots
- Spending Report
- Production Report
- Production History
- Freshness Report
- Recall Report
- Log

Users

Users contain the following:

- `_id`
- `createdAt`
- `services`
 - `password`
 - `resume`
- `username`
- `email`
 - `email`
 - `verified`
- `profile`
 - `name`
 - `first`
 - `last`
 - `username`
- `[roles]`

This is the built in Users database that is created using the Accounts library from Meteor. It contains your standard profile information (first name, last name, user name, email) as well as providing salting and encryption for passwords and verification of email. The resume contains the login tokens when using the application. There is a roles array that is attached to the user that contains all the roles and permission a user has.

By default, there is an “admin” created upon launch with administrator privilege to create new users and edit user permissions. There is also a company single sign on through Duke NetID. By default, NetID users will be granted the “unprivileged user” permission. They will need to contact an admin to request changes to their permission.

Roles

Roles contain the following:

- `_id`
- `Name`

This is the built in Roles database that is created using the Roles library from Meteor. Very standard database collection of roles that makes checking roles easier using the Roles API.

Ingredients

Ingredients contain the following:

- `_id`
- `name`
- `temperatureState`
- `storageUsed`
- `vendorInfo`
 - `[vendor]`
 - `vendorID`
 - `price`
- `packageInfo`
 - `packageType`
 - `numPackages`
- `nativeInfo`
 - `nativeUnit`
 - `numNativeUnitsPerPackage`
 - `totalQuantity`
- `formulaInfo`
 - `[formulaID]`

Ingredients collection contain all the necessary information that will be displayed on the inventory. Crucial information includes name, temperature state, and storage used. There are four sub-arrays of information as well. `VendorInfo` is an array of `<VendorID, price>` tuples. `PackageInfo` contains the package type as well as number of packages of the ingredient we have. `NativeInfo` contains the native unit of the ingredient, the number of native units per package, and the total quantity of the current ingredient we have. `Formula Info` is a string array that contains the formula ID's of all formulas that use the current ingredient.

Vendors

Vendors contain the following:

- `_id`
- `vendor (name)`
- `contact`
- `FCC (freight carrier code)`

Vendors contain the name, the contact information, and the freight carrier code within a single entry. Each vendor's name and freight carrier code is unique .

Storage Capacities

Storage Capacities contain the following:

- `_id`
- `name`
- `type`
- `capacity`
- `used`

Storage capacities contain the name, type of storage, total capacity, and used capacity per entry. Default we have frozen, refrigerated, and room temperature storage.

Final Products

Final Products contain the following:

- `_id`
- `name`
- `description`
- `productUnits`
- `[ingredientsList]`
 - `ingredientID`
 - `quantity`
- `quantity`

Final Products contain the name, description, and product Units that the formula entails and also keeps track of quantity as the formula is produced. The ingredients List is an array that keeps track of which ingredients and intermediates are used for the formula. Within each entry of the array, there is an ingredient ID and quantity used for the formula.

Intermediates

Intermediates contain the following:

- `_id`
- `name`
- `description`
- `productUnits`
- `[ingredientsList]`
 - `ingredientID`
 - `quantity`

- temperatureState
- storageUsed
- packageInfo
 - packageType
 - numPackages
- nativeInfo
 - nativeUnit
 - numNativeUnitsPerPackage
 - totalQuantity
- formulaInfo
 - [formulaID]

Intermediates are the crossover between final products and ingredients. They are produced from raw ingredients and / or other intermediates. They contain the same information as final products as well as most of the information from an ingredient item (excluding the vendor information, as these products are not associated with a vendor). They are stored in inventory and take up storage until they are consumed.

Spending Report

Spending Report contain the following:

- id
- productionTotal
- overallTotal

These database entries are the ingredients and intermediates mapped by their id. Within the report, we track of the production total (used in consumption to produce formulas) and overallTotal (when purchasing from vendors).

Production Report

Production Report contain the following:

- _id
- formula
- totalProduced
- ingredientsUsed
- totalSpent

The production report calculates the total spending based on the total production of formulas. This applies to both final products and intermediates.

Production History

Production History contain the following:

- `_id`
- `name`
- `lotNumber`
- `unitsProduced`
- `[lotsData]`
 - `Ingredient`
 - `[lots]`
 - `lotNumber`
 - `time`

Production History allows us to keep track, for each production run, the name of the formula produced, the lot number of the production, the units produced, as well as the lots of all the constituents used to produce the formula.

Freshness Report

Freshness Report contain the following:

- `_id`
- `inventoryID`
- `totalTime`
- `avgTime`
- `worstCase`
- `totalQtyConsumed`

The freshness report allows us to keep track of the average case and worst case an ingredient or intermediate sits in our inventory.

Recall Report

Recall Report contain the following:

- `_id`
- `inventoryId`
- `[lots]`
 - `lotNumber`
 - `quantity`
 - `[usedFor]`
 - `formulaID`

- lot Number
- time

The recall report allows us to keep track of the what was produced from each lot.

Carts

Storage Capacities contain the following:

- `_id`
- User
- [Ingredients]
 - Ingredient
 - Amount
 - Vendor

For carts, each user will have a cart created for them upon their first sign on (1-to-1 mapping). This simply contains an ingredients array that reflects the ingredient, amount in cart, and vendor that we are purchasing from.

Log

Logs contain the following:

- `_id`
- type
- name
- change
- from
- to
- user
- time

This is a global log that all users contribute to. It logs what type of change it was, displays the name of the change and the before and after values (if applicable). The logs also contain a time stamp and associated with a user so any change can be tracked by the system.