Atech Computers

Development Reference Manual

Table of Contents

[Table of Contents 2](#_Toc450638408)

[***Manual Version Control*** 3](#_Toc450638409)

[***1.*** ***Introduction*** 4](#_Toc450638410)

[***2.*** ***Project Structure*** 5](#_Toc450638411)

[***2.1*** ***meteor*** 5](#_Toc450638412)

[***2.2*** ***client*** 5](#_Toc450638413)

[***2.3*** ***packages*** 5](#_Toc450638414)

[***2.4*** ***private*** 5](#_Toc450638415)

[***2.5*** ***public*** 6](#_Toc450638416)

[***2.6*** ***server*** 6](#_Toc450638417)

[***2.7*** ***test*** 6](#_Toc450638418)

[***2.8*** ***bin*** 6](#_Toc450638419)

[***2.9*** ***docker*** 6](#_Toc450638420)

[***3.*** ***Routing*** 7](#_Toc450638421)

[***3.1*** ***Carefully designed client side router for Meteor*** 7](#_Toc450638422)

[***3.2*** ***How Reaction implements routing*** 7](#_Toc450638423)

[***3.3*** ***How Reaction deals with rendering*** 8](#_Toc450638424)

[***3.3*** ***Extending router*** 9](#_Toc450638425)

[***4.*** ***Templates*** 10](#_Toc450638426)

[***4.1*** ***Template replacement*** 10](#_Toc450638427)

[***4.2*** ***How does Reaction deal with template rendering*** 11](#_Toc450638428)

[***5.*** ***Functionality Development*** 13](#_Toc450638429)

[***5.1*** ***CSV File Import*** 13](#_Toc450638430)

[***5.1.1*** ***File Reading*** 14](#_Toc450638431)

[***5.1.2*** ***Reactive Design*** 15](#_Toc450638432)

[***5.1.3*** ***Supplier Product List Analysis*** 16](#_Toc450638433)

[***5.1.4*** ***Reaction Commerce Product Collection Analysis*** 18](#_Toc450638434)

[***5.1.5*** ***Reaction Commerce Product Collection Extension*** 19](#_Toc450638435)

[References 20](#_Toc450638436)

Manual Version Control

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Brief** | **Author** | **Date** |
| 1.0.0 | Initial version | Xiaochen Li | 17/4/2016 |
| 1.0.1 | Add 4.2 how does Reaction do with rendering | Xiaochen Li | 28/4/2016 |
| 1.0.2 | 5.1: CSV file import and reactive design | Xiaochen Li | 4/5/2016 |
| 1.0.3 | 5.1.3-5.1.5 Supplier product list analysis, database design based on the original Reaction Product collection. | Xiaochen Li | 10/5/2016 |

1. Introduction

Atech Computers website is built on ReactionCommerce, which is an open source platform based on Meteor framework. The platform provides web developers the functionalities that are necessary for all kinds of e-commerce business applications, such as account, product and order management, routing helper tools, and so on.

ReactionCommerce is a Meteor project, and it inherits all the features of Meteor. Meteor is a full-stack JavaScript platform for developing modern web and mobile applications. Meteor includes a key set of technologies for building connected-client reactive applications, a build tool, and a curated set of packages from the Node.js and general JavaScript community.

1. Project Structure

ReactionCommerce is built based on Meteor framework, so it follows its way of structuring its project. This section will introduce some of the important folders that are needed to be cared about.

***2.1 meteor***

Meteor facility folder in every Meteor project, including the information of Meteor and the project, such as Meteor version, all the packages and their versions supported by Meteor, and the collection of packages defined by the current project.

In addition, once we have built a package via build command, cache files for the product package will also be stored under .meteor/local folder.

***2.2 client***

Any directory named client/ is not loaded on the server. Similar to wrapping your code in if (Meteor.isClient) { ...}. All files loaded on the client are automatically concatenated and minified when in production mode. In development mode, JavaScript and CSS files are not minified, to make debugging easier. CSS files are still combined into a single file for consistency between production and development, because changing the CSS file’s URL affects how URLs in it are processed.

***2.3 packages***

It collects project-defined Atmosphere packages. Reaction functionalities are all defined as Atmosphere packages, their names starting with “reaction-”. Most Reaction packages can be obtained via its GitHub repository https://github.com/reactioncommerce/reaction.git. But payment-related packages have not been moved to this repository and they must be pulled from their own repository using “Reaction pull” command.

Packages are managed and built with the help of package.js file, which collects all the package information, and manages dependencies and versions. All files in the same package are combined into one JavaScript source file during the building process, in the appropriate order which all the dependencies are handled correctly.

***2.4 private***

All files inside a top-level directory called private/ are only accessible from server code and can be loaded via the [Assets](http://docs.meteor.com/#/full/assets_getText) API. This can be used for private data files and any files that are in your project directory that you don’t want to be accessible from the outside.

***2.5 public***

All files inside a top-level directory called public/ are served as-is to the client. When referencing these assets, do not include public/ in the URL, write the URL as if they were all in the top level. For example, reference public/bg.png as <img src='/bg.png' /> (be careful of the first ‘/’ meaning starting with root path). This is the best place for favicon.ico, robots.txt, and similar files.

***2.6 server***

Any directory named server/ is not loaded on the client. Similar to wrapping your code in if (Meteor.isServer) { ... }, except the client never even receives the code. Any sensitive code that you don’t want served to the client, such as code containing passwords or authentication mechanisms, should be kept in the server/ directory.

Meteor gathers all your JavaScript files, excluding anything under the client, public, and private subdirectories, and loads them into a Node.js server instance. In Meteor, your server code runs in a single thread per request, not in the asynchronous callback style typical of Node.

***2.7 test***

Any directory named tests/ is not loaded anywhere. Use this for any test code you want to run using a test runner outside of [Meteor’s built-in test tools](http://guide.meteor.com/testing.html).

***2.8 bin***

This folder includes Reaction scripts, such as run, install, reset and pull, all of which are called when running the “Reaction-like” commands.

***2.9 docker***

It includes scripts for manipulating Docker images. Since the project may be deployed to Docker engine hosts, so the tools are every useful to help us build Docker images.

1. Routing

Reaction implements the routing functionality provided by [kadira:flow-router](https://github.com/kadirahq/flow-router) for routing in the reaction-router package. It is a carefully designed client side router for Meteor applications.

Once imported as the dependency, flow router becomes available via the global variable called “FlowRouter”. The Reaction router, exported as “ReactionRouter”, references the “FlowRouter” variable.

***3.1 Carefully designed client side router for Meteor***

FlowRouter is a very simple router for Meteor. It does routing for client-side apps, but it does not handle rendering itself.

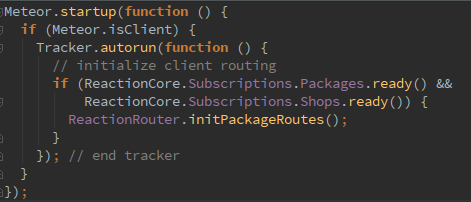
It exposes a great API for changing the URL and reactively getting data from the URL. However, inside the router, it's not reactive. Most importantly, FlowRouter is designed with performance in mind and it focuses on what it does best: routing.

***3.2 How Reaction implements routing***

Reaction-router package is responsible for initialization of ReactionRouter utility class, which is exposed as a global variable of the same name “ReactionRouter” that can be accessible throughout the application. But it should be noticed that most the helper functions, such as go and redirect, are only built to client side. Any route defined on server side will not function in the way it is supposed to.

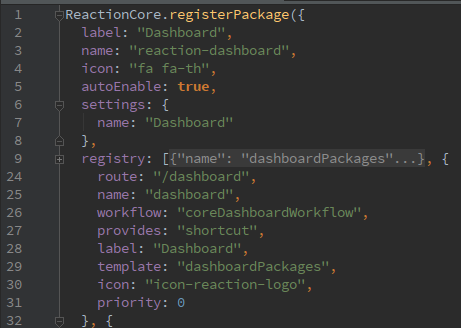
This manual will not involve much of its usage (please refer to the official website for further details <https://github.com/kadirahq/flow-router>). Instead, it will explain at code level how Reaction defines routes in “reaction-router” package and how it can be modified and extended, as well as some points which developers find interesting and worth attentions during their work with the project.

The router initialization starts with client router calling ReactionRouter to initialize package routes at Meteor startup (common/router.js).

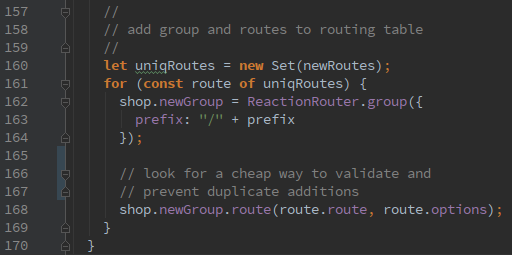


Note that Reaction only initializes Reaction-related package routes on client side. All Reaction routes are defined in Reaction Registry in “path-to-package/server/registry.js”.

Let’s take dashboard as an example. Dashboard defines its route as “/dashboard”, and then router initialization will search all the Reaction packages, extract the route from the registry and add the new route to the router.

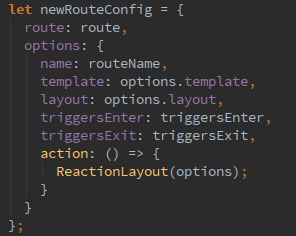


In addition, Reaction supports multi shops, which is reflected in router that all routes are organized into route groups (supported by FlowRouter) and use their corresponding shop name as path prefixes. However, we only manage one shop for our project, and this prefix seems to be redundant. It can be removed by setting the route group prefix to empty.



***3.3 How Reaction deals with rendering***

As mentioned above, FlowRouter does not handle rendering. Rather, it provides a callback for other rendering mechanisms, which gives developers flexibilities to choose the one they like. In render options, an action function allows us to call any rendering function.

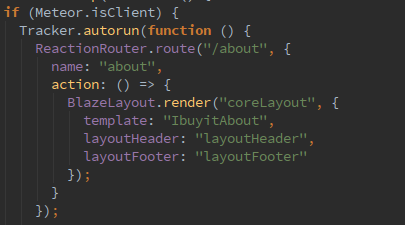


Reaction, by default, uses Blaze-Layout package (released with FlowRouter, in the snapshot above, it is called in ReactionLayout function). It facilitates page rendering by only passing several template names and avoiding developers involving too much in the Blaze rendering, because Blaze render method requires DOM and template instances as arguments.

Meteor applications are single-page web applications, which will also be explained in perspectives in the proceeding sections of this manual. Each route only renders one template per page (templates will be explained in next section), and there is not necessary to register sub templates which will be automatically dealt with by Blaze (one template rendering mechanism in meteor).

***3.3 Extending router***

Except for the routes defined by Reaction packages, other routes can also be declared in any files that are loaded on the client side. But in this framework, code for extending router should only written in /common/router.js. Sample code is:



When dealing with extended routes, be sure to put all the definitions before the package routes are loaded. Because at the end of the package routes loading, it calls initialize() function to finish the loading process, or else it is highly possible that the router cannot recognize the new route and returns a not found page.

1. Templates

Blaze is a powerful built in template library of Meteor. It provides the same live-updating capability as Angular and React, but it is much easier to use.

Templates are defined in html files within <template name=”my\_template\_name”> tag, with its name being exclusive in the entire application scope. They can also import sub templates.

At the start-up of the web application, templates as html files are loaded first. All templates are loaded and compiled into JavaScript objects, and from then on they are no longer html pages. This means that, unlike PHP, .Net or J2EE, among which platform that see html pages as separate files to be interpreted at runtime, Meteor must compile all the changes, build and deploy a new server package bundle, and then those changes can take effect. Although it will not take too much time for a small change to be built, it is annoying to wait even if IDE like WebStorm has a convenient mechanism to detect changes and trigger a rebuild automatically. Furthermore, managing html files separately allows files to be changed via administrator console without stopping the server. When it comes to Meteor, the server cannot receive any request during the build process.

HTML files in a Meteor application are treated quite a bit differently from a server-side framework. Meteor scans all the HTML files in your directory for three top-level elements: <head>, <body>, and <template>. The head and body sections in all the html files are separately concatenated into a single head and body, which are transmitted to the client on initial page load. In this aspect, defining head and body sections in more than one html files is not necessary, and potentially, it leads to confusions.

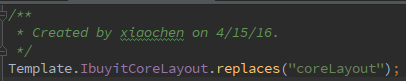
Helpers are variables and functions that belongs to a certain template. They are usually defined in JavaScript files that have the same names to their html files, and helpers are loaded after all the templates being loaded.

Details of how to use Blaze libraries and space bars can be found at the official website https://www.meteor.com/blaze.

* 1. ***Template replacement***

In order to keep the original functionalities, and in the meantime, customize the page design in a way that will not affect the Reaction updating process (Reaction is officially updated once a week), it is best to replace the built-in templates in /client directory.

Reaction includes the “[meteor-template-extension](https://github.com/aldeed/meteor-template-extension)” package which provides functionality for extending Meteor templates. With the help of this package, replacing an existing template is very simple via just one line of JavaScript code. For example, add a template extension map to client/templates.js:



As the code shows, a customized template called “IbuyitCoreLayout” replaces the Reaction built-in template called “coreLayout”. However, in this case, helpers, events, etc., that are defined for “IbuyitCoreLayout” will not be available. Reaction still uses those defined for “coreLayout” template. The original helpers and events are still allowed to be extended, and also they can be called in the new template directly without extra coding. But when being extended, helpers and events, etc., ignore the variables and functions which have been defined before with the same name.

In the Meteor framework, all html templates are translated to JavaScript objects during the build process. What the “replaces()” function really does is to replace the “coreLayout” object with the “IbuyitCoreLayout” object, and nothing else changes.

It is suggested that developers mirror the existing structure. For example, if you are extending the template located in

packages/reaction-core/client/templates/products/products.html

Then you would mirror that by creating your new template at

reaction/client/templates/products/myTemplate.html.

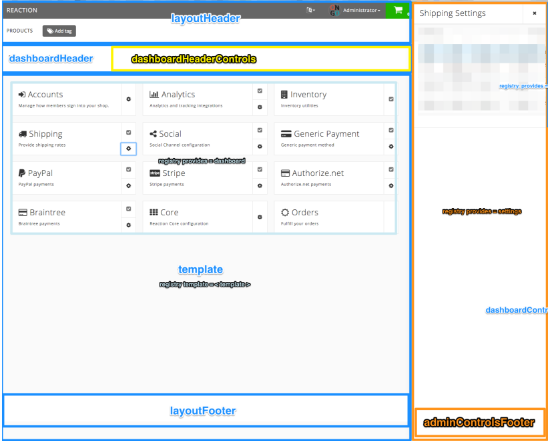
* 1. ***How does Reaction deal with template rendering***

We already know that FlowRouter provides developers an “action” option in which they can do whatever they like to render pages. The choices may be Angular, React or Blaze. The last one is what Reaction is using by default, and other meteor packages may be employed to implement the first two when rendering customized templates.

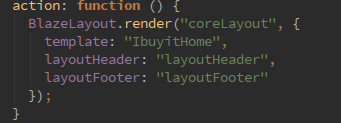
Reaction splits the entire page into several parts: header, footer, template, dashboard header and dashboard control. Each of them is a separate template: Header and footer are shared by all pages; dashboard header and setting control templates are only available for administrators. The “template” region in the picture shows the main part determining the page content. All those parts consist of the whole web application page whose template is called “coreLayout” defined by “reaction-layout” package.

Usually, developers have no necessity to change the basic structure of core layout. What they are supposed to do is customizing those separate templates to what they need. And then core layout will import them into its structure. Therefore, whatever the pages are, the only template to render is “coreLayout”.

When rendering Blaze templates, Reaction uses “BlazeLayout” instead of the built-in Meteor rendering method. “BlazeLayout” is a Meteor package released with FlowRouter and is more convenient to use. It takes the responsibility to manage template and DOM objects and only receives the template names as parameters.



However, only passing core layout and rendering it is not enough. BlazeLayout also requires the names for those partial templates as an option object passed as the second argument. The example comes from the home router:



Where the specific templates are needed to be declared, or else they will not be visible.

1. Functionality Development

This chapter will be in charge of explaining how each functionality is developed, especially in terms of the technical details on the basis of the previous chapters. The functionalities are addressed by the client, and they are the features that the client’s website and most of the current platforms fail to provide.

* 1. ***CSV File Import***

This is the feature that our client has been asking for. He has multiple product providers, all of which sell several categories of products and usually update their stock database and share out spreadsheets to their customers regularly. It has been a trouble for our client to update all the information. What he has kept doing is going to the providers’ websites and check if some product is still available in stock. Now he is looking for a system that can be customized with this function.

There are several technical points to be considered before development. First, how to read files on the client side. Meteor is a client side framework; and furthermore, it is reasonable to parse the files on the client side, just with one call of remote method to insert the product data. But if the files are processed on the server, there will be several times of client-server interactions to complete the procedure: transmits the files, the server returns the results and then the items needed are inserted into the database.

Second, and possibly the trickiest part of the file import functionality, how to incorporate several formats of csv files into one database. They have different names for the same column, they include different fields, and they also own the fields that never have values but are still required to keep. Reaction implements the product collection in its way, and it makes things more complicated as we must also be careful about not affecting the current logic.

Third, the parsed results must be listed on the page and thus the users are given a chance to check all the product information that is about to be inserted into the database. And the users are allowed to delete the products that are possibly not necessary any more. Furthermore, as illustrated by the sample csv files, more than one thousand products in a single file are expected. So the acceptable time for parsing a file is our concern. In terms of displaying the information, listing all the products in paging can be an appropriate way.

The final step of the whole process is calling the remote methods and inserting all the products into mongo. We have not got involved in this document in the database details at this moment. There are expectation of rules of updating the information and we will refer to the client when we have questions.

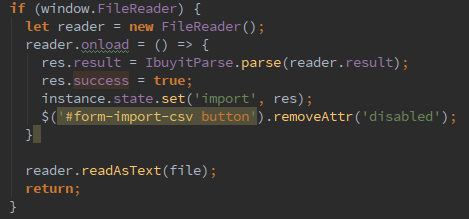
* + 1. ***File Reading***

One easier starting point of file scanning is to find a library for reading csv files rather than developing one of ourselves. One of the libraries called “PapaParse” is a powerful csv file reader for the browser written in JavaScript. It is very small with only 15kb in size. Generally speaking, it only has one method but it is sufficiently powerful to read both csv files and txt files.

PapaParse also has another stream called BabyParse which is maintained for Nodejs apps. BabyParse is simpler but is quite sufficient for our csv reading. The main difference between two libraries is that BabyParse uses Nodejs file system package ‘fs’ while PapaParse has more functionalities and does not need a file system because it is browser-based.

BabyParse cannot parse files (in terms of the file system) because ‘fs’ package does not exist on the client side (browser), but reading a JSON string is quite possible. So our way out is to find a reader to read files submitted from the inputs into a JSON string. FileReader object in the browsers is a tool to asynchronously read a file or a blob. Statistics of how good the browsers support FileReader can be found in <http://caniuse.com/#feat=filereader>, where we can see that FileReader is available in almost all recent desktop and mobile browsers.

One example of usage of BabyParse is:



FileReader receives several callback functions to implement the asynchronous reading, with one of the callbacks “onload” that occurs when a read operation by the FileReader object successfully completes [4]. Reading operation is called after all the callback functions are defined. As we call to read the file as text here, the result is returned as a JSON string by reader.result representing the csv file content.

BabyParse translates the JSON string into JavaScript objects. The object is an array, the title line being the first element. With the help of this title line, we can determine which provider the csv file belongs to.

* + 1. ***Reactive Design***

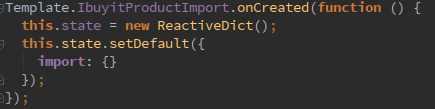
Reactive programming is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) oriented around [data flows](https://en.wikipedia.org/wiki/Dataflow_programming) and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow [5].

As for our csv file importing functionality, reactive design refers that the product list can be automatically changed after another file is imported without the need for the developers to manage page updates. Meteor has built in reactive mechanism with the help of spacebar and Blaze. This manual will not concern the implementation of Meteor reactive design. It only gives a brief introduction of how our project is employing the design.

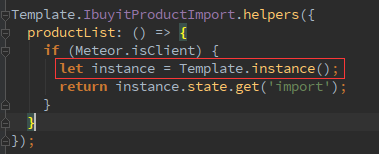
The ReactiveDict package is a general-purpose reactive datatype for use with Tracker, providing reactive get, set and equal functions. It traces all the values defined in it and triggers a corresponding functions to run once it detects value changes.

Working with templates, ReactiveDict does not definitely need Tracker, because the Blaze rendering mechanism takes the responsibility. Template interpreter looks for any ReactiveDict instance in helpers and traces the value changes. If any change is detected, the page will be automatically re-rendered.

A template can define a ReactiveDict instance of its own in onCreated callback functions like:

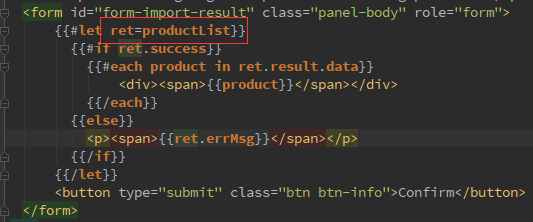


Thus, “state” can be referenced by “IbuyitProductImport” template instance. The template instance can be obtained by Template.instance() in helpers and as the second parameter of event functions:





The helper function “productList” is called in the template and then the page will be updated after we submit a new csv file.



* + 1. ***Supplier Product List Analysis***

Four suppliers are of our concern for this moment: IM, Anyware, PB and Synnex. They have entirely different list formats, which can be a big problem for our database design. But fortunately, the client requires that only some of the fields needed and a couple of fields that are much less important and are specific only to some supplier ignored. He also helped ask the suppliers to format their product list into csv or txt files only. Thus, a simplified database design can be achieved in a simpler way.

According to the client, product name, stock, prices (retail price, and possibly shop price or discounted price), stock (show back order if available) are the major ones included. Also, category (also called product group) is useful when searching a particular kind of product. But IM does not have a corresponding field meaning category, and it is not a supplier of highest priority, too. That is why its list is out of consideration right now.

The table next page illustrates the fields from all four suppliers. Those that are labeled gray are fields to be ignored in our database design; those labeled orange are supplier-specific fields. It is clear that the fields that make the database design complicated are ignored, only no more than 3 fields identifying different suppliers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **IM.csv (ignored)** | **Anyware.csv** | **PB.csv** | **Synnex.txt** |
| Product Details | No. | Item Number | PB Part Number | Part No. |
|  |  |  | Stock Code |
| Name | Item Name | Product Name | Description |
| Brand | Brand | Brand Name | Vendor |
|  |  | Manufacturer's Code |  |
| brief |  |  |  |
| details |  |  |  |
| Categories |  | Category | Product Group | Product Group |
| Stock | Entity | QuantityOnHand | Bulk Stock | stock |
| Stock Quantity |  |  |  |
| Stock warning Quantity |  |  |  |
| Price | Market Price | Selling Price | Your Price | RRP - retail price |
| Shop Price |  |  |  |
|  |  |  | Block1\_Price (1-4) |
|  |  |  | Block2\_Price (5-9) |
|  |  |  | Block3\_Price (10+) |
| Point limis for Buying |  |  |  |
| Others | Original Picture |  |  |  |
| Picture |  |  | ETA  Estimated Time of Arrival |
| Thumbnail |  |  | Gcode - supplier internal code |
| Keywords |  |  |  |
| weight |  |  |  |
| Best |  |  |  |
| New |  |  |  |
| Hot |  |  |  |
| OnSale |  |  |  |
| Can be a common  product sale? |  |  |  |

* + 1. ***Reaction Commerce Product Collection Analysis***

Reaction manages two kinds of Mongo collections for products: Product and ProductVariant. The latter one is mainly for those that have multiple prices, for different colors as an example. According to the client and our research, the suppliers do not have such kind of products, so it is not used in the design.

MongoDB is one of the schema-less database engines, but Meteor employs a package named simple-schema to achieve schema for simple, reactive schema validation purposes. Product schema is defined in packages/reaction-schema/common/schemas/prpduct.js. All fields defined can be found in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Fields** | **Data Type** | **Optional** |
| 1 | \_id | String |  |
| 2 | ancestors | String[] |  |
| 3 | shopId | String |  |
| 4 | title | String |  |
| 5 | pageTitle | String | true |
| 6 | description | String | true |
| 7 | type | String |  |
| 8 | vendor | String | true |
| 9 | metafields | Metafield | true |
| 10 | positions | ProductPosition | true |
| 11 | price | PriceRange |  |
| 12 | isLowQuantity | Boolean | true |
| 13 | isSoldOut | Boolean | true |
| 14 | isBackorder | Boolean | true |
| 15 | requiresShipping | Boolean | true |
| 16 | parcel | ShipingParcel | true |
| 17 | hashtags | String | true |
| 18 | twitterMsg | String | true |
| 19 | facebookMsg | String | true |
| 20 | googleplusMsg | String | true |
| 21 | pinterestMsg | String | true |
| 22 | metaDescription | String | true |
| 23 | handle | String | true |
| 24 | isVisible | Boolean |  |
| 25 | templateSuffix | String | true |
| 26 | createdAt | Date |  |
| 27 | updatedAt | Date | true |
| 28 | publishedAt | Date | true |
| 29 | publishedScope | String | true |
| 30 | workflow | Workflow | true |

**Note** Fields labeled blue can be employed in our design, and the schema is to be extended for all the four suppliers. Some fields from the original Product collection, such as “pageTitle”, “vendor”, “positions” and “hashtags”, are no longer needed, while most of them are useful for some of the functions.

* + 1. ***Reaction Commerce Product Collection Extension***

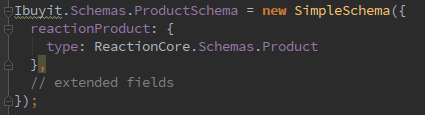
In order to incorporate suppliers’ fields into current database design, the original Product collection needs to be extended to accommodate information that specific to each product list. Fields with the same idea from different supplier formats are translated into one name. Synnex must own another schema to include block prices inheriting the universal product schema. Provider product fields and their corresponding schema is are listed below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Collection Fields** | **Anyware.csv** | **PB.csv** | **Synnex.txt** |
| no | Item Number | PB Part Number | Part No. |
| title | Item Name | Product Name | Description |
| brand | Brand | Brand Name | Vendor |
| category | Category | Product Group | Product Group |
| stock | QuantityOnHand | Bulk Stock | stock |
| retailPrice | Selling Price | Your Price | RRP - retail price |
| block1Price |  |  | Block1\_Price (1-4) |
| block2Price |  |  | Block2\_Price(5-9) |
| block3Price |  |  | Block3\_Price(10+) |

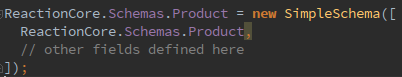
It is noticed that the collection field line starting with “title” is colored in light blue, meaning that the field is introduced from Reaction Product collection rather than an extended one, while the others are all new fields.

The package “simple-schema” provides two simple ways for extending schemas: sub-schema and additional fields.

**Sub-schema** is defining in the new schema a field as the type of the original schema. For example, if we want to extend Product collection, we may write the code like:



**Additional fields** refers to passing multiple schemas to SimpleSchema constructor to combine fields where no further document is created except for the original one extended, like:



Build & Deployment

Test

Appendix

Useful JavaScript features (Including ECMAScript6)

# References

[1] ReactionCommerce doc, <https://docs.reactioncommerce.com/>

[2] FlowRouter, <https://github.com/kadirahq/flow-router>

[3] Blaze, <https://www.meteor.com/blaze>

[4] FileReader object, <https://msdn.microsoft.com/en-us/library/hh772310%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396>

[5] Reactive Programming, <https://en.wikipedia.org/wiki/Reactive_programming>