Getting Started with Meteor

Jhonatan S. Oliveira oliveira@cs.uregina.ca
Department of Computer Science
University of Regina
Regina, Canada

Editor: CS807 - Advanced Architecture - Dr. Gerhard

Abstract

Meteor is considered a full-stack javascript framework. That is, it handles from the server and data modeling to the client and user interface. Besides this all, Meteor uses only javascript as its programming language, meaning that a whole application may be build using only javascript. Moreover, the framework incorporate reactive programming features. Here, the flow of the data is more focused. In Meteor, this is tracked by using events. A general application using the framework will keep track of events and modify database and user interface accordingly. In this paper, we also show how Meteor can be used in resource constrains computing applications to distribute computing tasks in servers and letting the client with lightweight tasks.

Keywords: Meteor, javascript, resource constrains

1. Introduction

In 2001, after the the bursting of the dot-com com bubble, the Web 2.0 phenomenon appeared and a survival way to the recent collapse (O'Reilly, 2005). New applications aimed to involve the user on the content creation process. One way of engaging the user is to offer a familiar environment, a similar to the one used in a desktop computer. Web platforms and frameworks made use of *reactive programming* (Bonér et al., 2014) to achieve a desktop-like experience on the web. A consequence of turning the content creation responsibility to the user is the massive amount of data created by them, which originated another phenomenon called *big data* (Sharma et al., 2014). In order to manage all these data, new paradigmas emerged. *Non-relational* or *No-SQL* (Strauch, 2012) is a database paradigm that manage the data storage and retrieval without using a relational table. Novel web platforms and frameworks had to incorporate these new aspects in order to model and maintain the complexity requirement of the new applications.

Meteor.js (Meteor, 2016), or Meteor for short, is an open-source web platform to create applications using the Web 2.0 paradigmas. The platform provide a reactive approach by focusing on the data flow. This is done by creating and managing events in the application. Also, data management is done with a non-SQL database called *MongoDB* (Mongo, 2016). Meteor simplify the application development by providing an unique programming language, javascript, throughout the whole stack process. Moreover, the platform makes available a set of common tools for business logic and data management. Finally, Meteor deploys the application in desktop and mobile without needing to change the source code.

In this paper, we seek to give an overview of Meteor aiming to get reader started on the platform. We start by showing the template system which draws pages and receives event from the user. Next, the database management, called *collections* in Meteor, is presented and some basic methods for creating, deleting, and updating entries. Then, we show how to deal with events on the application. Throughout the paper, we will use one running example which will be a To-do list type. Lastly, we show how Meteor can be used on resource constrain applications by transferring the heavy workload of the app to a server and letting the user interface respond reactively in a lightweight workload.

The reminder is as follows. Section 2 shows a quick overview on the history and background informations of the Meteor platform. Basic concepts and introduction is given in Section 3. Section 4 shows how Meteor can be used in resource constrain applications. Conclusions are given in Section 5.

2. History and Background

The Meteor platform was created by a company called Skybreak in 2011 (Skybreak, 2012). Later, in 2012, the company changed its name to Meteor. The startup was incubated by YCombinator (Griffith, 2012) and after receiving an investment of $11.2 M, the platform development increased considerably. The platform left beta in October 26th, 2015, with a version that currently provides multiplatform support.

Regarding its internal structure, Meteor is built on top of *Node.js* (Node.js, 2016). That means that Meteor is driven by events, in order to create an asynchronous model. This feature is implemented using callback functions: when an event happens a specific function is called to execute a portion of code. The server-side and client-side are both implemented using javascript. In the client-side, templates are used to design user interfaces. Here, a simple markup language defines the design and the events handled by the application. Meteor has support for more than one template language, being *Blaze* (Meteor, 2016) the official one. In the server-side, Meteor handles data management using collections. The oficial non-SQL database supported is MongoDB, but new ones are being incorporated in future versions (Lardinois, 2016).

3. Getting Started

Now, we will follow a common flow for an app creation process. During this section, we will use a single running example which is an application for todo lists. This todo example can also be found on the official getting started tutorial in (Meteor, 2016). We will try to keep the same code conventions and names always that possible so the reader can use this paper as an extended guide for that tutorial. We assume that the Meteor program and required

The following command line is used to create a new application with Meteor:

```
meteor create todo
```

This creates a folder structure as shown below:

```
todo.js todo.html todo.css .meteor
```

The *todo.js* file is where the javascript code for the server and client stays. Here is where the code throws and handles events, besides business logic. Templates for user interface

is done in the *todo.html* file with pure HTML markup language and a template markup language. For this project Blaze is the template languages used. The *todo.css* file contains styles for the templates. Lastly, the *.meteor* stores settings and the meteor aplication itself in a hidden folder.

3.1 Templates

Templates are defined using a special tag called *template*. Once defined, a template can be included within any HTML code. In this way, a template is a interface module that can be reuses wherever it is required. All the HTML code and the templates have access to the data made available in that part of the HTML code. The basic flow is: the javascript code makes available some data to a specific area of the HTML code (or a specific template), so the template language can access that data and print the ones of interest for the user. Bellow, we show a simple todo list interface.

```
1  <head>
2    <title>Todo List</title>
3  </head>
4
5  <body>
6    <div class="container">
7      <header>
8        <h1>Todo List</h1>
9        <form class="new-task">
10         <input type="text" name="text" placeholder="Type to add new tasks" />
11       </form>
12     </header>
13
14     <ul>
15       {{#each tasks}}
16         {{> task}}
17       {{/each}}
18     </ul>
19   </div>
20 </body>
21
22 <template name="task">
23   <li class="{{#if checked}}checked{{/if}}">
24     <button class="delete">&times;</button>
25     <input type="checkbox" checked="{{checked}}" class="toggle-checked" />
26     <span class="text">{{text}}</span>
27   </li>
28 </template>
```

Here, the HTML code defines in its *body* a title and a list. Notice that Blaze commands are defined within {{ and }}. In the list, the Blaze command *#each* goes through a list of data, as in a loop lace. The data variable *tasks* contains this list of data and was made available to this part of the code by the javascript code. The Blaze command > *name_of_template* prints a template previously defined with name *name_of_template*. In lines 22-28, the template named *task* is defined. This template expect to find available in its scope a data variable called *text*. In line 16, the template is included in that spot

of the HTML code and the data variable *text* is available on the loop scope of the *#each* command. Note that *text* is a field within *tasks*.

On the javascript file we define code for the client and server. In order to distinguish between them, Meteors makes available a global variable called *isClient*, which goes to true if the running environment is the browser and goes to false if it is the Node.js one. If a code is called without being under the conditions of a client or server, the code is run in both environments. Follows the javascript code to create a simple database where the *tasks* are saved.

```
Tasks = new Mongo.Collection("tasks");

if (Meteor.isClient) {

  // This code only runs on the client
  Template.body.helpers({
    tasks: function () {
      return Tasks.find({});
    }
  });

  Template.body.events({
    "submit .new-task": function (event) {
      // Prevent default browser form submit
      event.preventDefault();

      // Get value from form element
      var text = event.target.text.value;

      // Insert a task into the collection
      Tasks.insert({
        text: text,
        createdAt: new Date() // current time
      });

      // Clear form
      event.target.text.value = "";
    }
  });

  Template.task.events({
    "click .toggle-checked": function () {
      // Set the checked property to the opposite of its current value
      Tasks.update(this._id, {
        $set: {checked: ! this.checked}
      });
    },
    "click .delete": function () {
      Tasks.remove(this._id);
    }
  });

}
```

Line 1 runs on the server and on the client, since it is not inside the conditional in line 3. In the server, that command creates a database called *tasks* if it does not exist already. In the client, it creates a cache of the same database where Meteor manages some saved data in order to reuse repeated queries from the user. From line 6 to 9, using the global variable *Template*, we make available to the *body* of the HTML whatever data the not named function in line 7 return. The function returns all the entries in the previously created *tasks* database. These returned data is available to the HTML code on a data variable called *tasks*, as determined in line 7.

3.2 Inserting Data

Regarding the data inclusion, this HTML code has a form which will receive the new input data from the user. Lines 9-11 defines a simple form where the user can input new tasks. When pressing enter, an event is thrown from the HTML code and can be handled in the javascript code.

The javascript code watch for an event for the form submission, as shown in line 13. From here, the default reaction of the browser, which is try to submit the form, is stopped. The, the value of the input text is saved in a variable. Next, the cached database variable *Tasks* is used to insert a new entry on the server database with the text saved from the user input. Finally, the text input is cleared.

Notice in the last insertion process that the cached database was used to update the server database. This is possible due to the way Meteor works in client and server. The client has only a cached version of the database. But whenever this cached version is updated, an event goes to the server (whenever there is connection available) making the server database to update. The other way around works in the same manner: if the server database is updated, an event goes to all client spreading the update.

3.3 Updating and Removing Data

If a task is done, we want to check it out from the list by updating its entry with a done flag. In the HTML code, line 23, the template *task* has a conditional statement checking for a variable called *checked*. If the variable is true, the body of the condition, which is just a string *checked*, is executed. This variable, as expected, is set in the javascript code. Indeed, in line 32, the javascript watch for an event of a click on the checkbox. If it happens, the cached database *Tasks* is updated by setting a field *checked* to the opposite value that the user entered. Notice the use of a special variable *this* which provides context for HTML access from where the event occurred.

The deletion process is similar to updating but a different function is used on the cached database. The javascript code, line 24, watch for the HTML code, line 24. When the user clicks in the delete button, the javascript code is executed by removing the correspondent id of the entry. Again, note the use of *this* as a context variable for the entry where the event occurred.

4. Resource Constrains

Meteor can be used to build resource constrain products. In this type of application, the amount of processing and memory are often restricted. Therefore, the use of parallel computing or decentralized tasks are common and necessary. Meteor can be use to avoid heavy load computation in the limited hardware itself. A server can be used to process all the required heavy workload while the client simply show in real time the results.

One way of providing an interface to a Meteor server side is by using the official server-client protocol, called *Distributed Data Protocol* (DDP). The DDP protocol is a simple specification on how other languages can communicate with a Meteor server.

5. Conclusions

Meteor is a simple platform for building reactive applications on the Web 2.0 paradigma. Its event drive method offers a real time experience for the users, besides a clear division between client and server sides tasks. Thus, Meteor can be used in resource constrain products by providing a solution when splitting workload among the more powerful server and the lightweight client.

References

Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto, September 2014. URL `http://www.reactivemanifesto.org`.

Erin Griffith. First github, now meteor: Andreessen horowitz backs another developer favorite, July 2012. URL `https://pando.com/2012/07/25/first-github-now-meteor-andreessen-horowitz-backs-another-developer-favorites/`.

Frederic Lardinois. Meteor acquires yc alum fathomdb for its development platform, October 2016. URL `http://techcrunch.com/2014/10/07/meteor-acquires-yc-alum-fathomdb-for-its-web-development-platform/`.

Meteor, 2016. URL `https://www.meteor.com`.

Mongo, 2016. URL `https://www.mongodb.org`.

Node.js, 2016. URL `https://nodejs.org`.

Tim O'Reilly. Design patterns and business models for the next generation of software, September 2005. URL `http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html`.

Sugam Sharma, Udoyara Sunday Tim, Johnny S. Wong, Shashi K. Gadia, and Subhash Sharma. A brief review on leading big data models. *Data Science Journal*, 13:138–157, 2014.

Skybreak, 2012. URL `http://info.meteor.com/blog/skybreak-is-now-meteor`.

Christof Strauch. *NoSQL Databases*. Stuttgart Media University, 2012.