Prototyping With Meteor

Tom Coleman on Mar 26th 2013 with 32 Comments

Tutorial Details

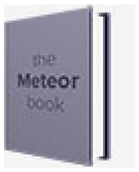
-
- **Difficulty:** Intermediate
- **Completion Time:** 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Meteor is far more than a quick prototyping tool, but it sure is great for prototyping. In this tutorial, we'll walk through the process of turning a simple HTML wireframe into a functional application in a surprisingly simple number of steps.

We'll be creating a simple wireframe of a chatroom application. The process we are trying to demonstrate is starting with a pure HTML/CSS wireframe, done in Meteor for convenience, which can then very easily be transformed into a real application, thanks to the ease of Meteor development.

Note: This tutorial is adapted from The Meteor Book, an upcoming step by step



guide to building Meteor apps from scratch. The book will walk you through building a complete multi-user social news site (think Reddit or Digg), starting from setting up user accounts and user permissions, all the way to managing real-time voting and ranking.

Setting up a More Complex App

A previous Meteor tutorial here on Nettuts+ demonstrated how to install Meteor and build a simple application using the `meteor` command line tool. In this tutorial, we are going to do things a bit differently, and use Meteorite.

Meteorite is a community created wrapper for Meteor that allows us to use non-core packages created by other members of the Meteor community. Although a built-in third party package system is planned for Meteor itself, as of the time of this writing, there is no support, bar the set of packages that are supported by the Meteor core team. So Meteorite was created to allow us (the community) to work around this limitation, and publish our packages on Atmosphere, the Meteor package repository.

For this tutorial, we are going to use some of those community written packages, so we are going to need to use Meteorite. To begin, let's get it installed, using npm.

> Note: You'll need to have a copy of Node and npm installed on your system. If you need assistance with this process, Meteorite's install instructions is a good place to start.

```
npm install Meteorite -g
```

If you're on Windows, setting things up is a bit more complex. We've written a detailed tutorial on our site to help you out.

Now that Meteorite is installed, we use the `mrt` command-line tool (which it installs for us) in place of `meteor`. So let's get started! We'll create an app:

```
mrt create chat
```

Packages and Wireframes

To create our wireframe app, we'll use some basic packages that allow us to develop simple laid out pages quickly and route between them. Let's add the packages now:

```
mrt add bootstrap-updated
mrt add font-awesome
mrt add router
```

Step 1: A Front Page

Now that we've picked up some nice styling for our app, we can make a mockup of the landing screen. Delete the initial HTML, CSS and JS files created by Meteor and create the following two files within a `client` directory (we aren't doing anything on the server yet).

(Alternatively, follow along with the steps from this repository.)

```
<head>
  <title>chat</title>
</head>

<body>
  <div class="row">
    {{> rooms}}
    <div class="span6">
      <h1>Welcome to Meteor Chat</h1>
      <p>Please a select a room to chat in, or create a new one</p
    </div>
  </div>
</body>

<template name="rooms">
  <form name="new_row">
    <table id="rooms" class="table table-striped span6">
      <thead>
        <tr>
          <th>Room Name</th>
          <th>Members online</th>
          <th>Last activity</th>
        </tr>
      </thead>
        <tbody>
```

```
          {{#each rooms}}
            <tr>
              <td>{{name}}</td>
              <td>{{members}}</td>
              <td>{{last_activity}}</td>
            </tr>
          {{/each}}
          <tr>
            <td colspan="3">
              <input type="text" name="name" placeholder="Enter your
              <button type="submit" class="btn btn-primary pull-righ
                <i class="icon-plus"></i> Create Room
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </form>
  </template>
```

client/chat.html

```
var rooms = [
  {name: 'Meteor Talk', members: 3, last_activity: '1 minute ago'},
  {name: 'Meteor Development', members: 2, last</em>activity: '5 mi
  {name: 'Meteor Core', members: 0, last_activity: '3 days ago'}
];

Template.rooms.helpers({
  rooms: rooms
});
```

client/chat.js

After adding this, you should see the following simple (if fake) application, when you browse to http://localhost:3000:

The data in the table of rooms is *fixed* data that we have manually entered into `client/chat.js`, but the advantage to this approach is that it allows us to repeat HTML in our wireframe without having to cut and paste (which is almost universally a bad idea).

Step 2: A Chat Room Page

Now, let's hook up a second page. We are going to use the router to select between two page templates; one with the welcome message, and the other with a message list for the selected room.

Let's start by adding some simple routes. The Router works by mapping URLs to template names. Our case is fairly simple; here's what we add:

```
Meteor.Router.add({
  '/': 'home',
  '/rooms/:id': 'room'
});
```

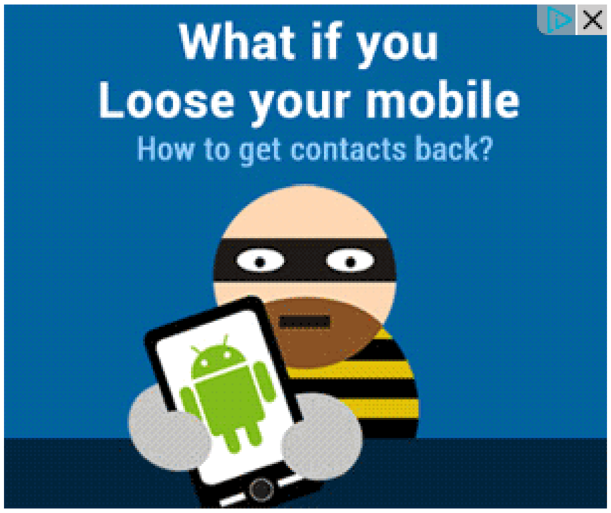
client/chat.js

```
<body>
  <div class="row">
    {{> rooms}}
    {{renderPage}}
  </div>
</body>

<template name="home">
  <div class="span6">
    <h1>Welcome to Meteor Chat</h1>
    <p>Please a select a room to chat in, or create a new one</p>
  </div>
</template>
```

client/chat.html

We use the `{{renderPage}}` helper in our HTML file to indicate where we want that selected template to draw, and, just like that, we can browse between the two URLs and see the content on the right-hand side change. By default, we see the `'home'` template, which is mapped to the route `/`, and shows us a nice message.



If we add a template from the `'room'` route and add some links to specific rooms, we can now follow links:

```
<td><a href="/rooms/7">{{name}}</a></td>
```

client/chat.html 'rooms' template

```
<template name="room">
  <div class="span6">
    <h1>Welcome to a chatroom!</h1>
  </div>
</template>
```

client/chat.html

This works because the router maps urls, like localhost:3000/rooms/7, to the `'room'` template. For now, we won't look at the id (7, in this case), but we will soon!

Step 3: Putting Some Data in the Chat Room

Now that we've routed a URL chatroom, let's actually draw a chat in the room. Again, we are still mocking up, so we'll continue to create fake data in our JavaScript file, and draw it with Handlebars:

```
var rooms = [
  {name: 'Meteor Talk', members: 3, last_activity: '1 minute ago',
    messages: [
      {author: 'Tom', text: 'Hi there Sacha!'},
      {author: 'Sacha', text: 'Hey Tom, how are you?'},
      {author: 'Tom', text: 'Good thanks!'},
    ]},
  {name: 'Meteor Development', members: 2, last</em>activity: '5 m
  {name: 'Meteor Core', members: 0, last_activity: '3 days ago'}
];

Template.room.helpers({
  room: rooms[0]
});
```

client/chat.js

So we've added some chat data to the first room, and we'll simply render it every time (for the moment) on the room template. So:

```
<template name="room">
  <div class="span6">
    {{#with room}}
      <h1>Welcome to {{name}}</h1></p>
      <table id="chat" class="table table-striped">
        <tbody>
          {{#each messages}}
            <tr>
              <td>{{author}} :</td>
              <td>{{text}}</td>
            </tr>
          {{/each}}
          <tr>
            <td colspan="2">
              <form name="new_message">
                <input type="text" name="text"></input>
                <button type="submit" class="btn btn-primary pull-
                  <i class="icon-envelope"></i> Send message
                </button>
              </form>
            </td>
          </tr>
        </tbody>
      </table>
    {{/with}}
  </div>
</template>
```

client/chat.html

Voila! A working demonstration of our chatroom application:

Step 4: Using Real Data Backed by a Collection

Now comes the fun part; we've built a simple wireframe of static data simply enough, but thanks to the power of Meteor `Collections`, we can make it functional in no time at all.

Remember that a Collection takes care of syncing data between the browser and the server, writing that data to a Mongo database on the server, and distributing it to all other connected clients. This sounds like exactly what we need for a chat room!

First, let's add a collection on the client and server, and add some simple fixture data to it:

(Note: We'll put the collections file in the `lib/` directory, so that the code is available both on the client and the server.)

```
var Rooms = new Meteor.Collection('rooms');

if (Meteor.isServer && Rooms.find().count() == 0) {
  var rooms = [
    {name: 'Meteor Talk', members: 3, last_activity: '1 minute ago
      messages: [
        {author: 'Tom', text: 'Hi there Sacha!'},
        {author: 'Sacha', text: 'Hey Tom, how are you?'},
        {author: 'Tom', text: 'Good thanks!'},
      ]},
    {name: 'Meteor Development', members: 2, last</em>activity: '5
    {name: 'Meteor Core', members: 0, last_activity: '3 days ago'}
  ];
  _.each(rooms, function(room) {
    Rooms.insert(room);
  });
}
```

lib/collections.js

We've moved our data into the collection, so we no longer need to manually wire it up within our template helpers. Instead, we can simply grab what we want out of the collection:

```
Meteor.Router.add({
  '/': 'home',
  '/rooms/:id': function(id) {
    Session.set('currentRoomId', id);
    return 'room'
  }
});

Template.rooms.helpers({
  rooms: function() { return Rooms.find(); }
});

Template.room.helpers({
  room: function() { return Rooms.findOne(Session.get('currentRoom
})
```

client/chat.js

We've made a couple of changes here; Firstly, we use `Rooms.find()` to select all rooms to pass into the `'rooms'` template. Secondly, in the `'room'` template, we just select the single room that we are interested in (`Rooms.findOne()`), using the session to pass through the correct `id`.

Hold on! What's the session? How did we pass the `id`? The session is Meteor's global store of *application state*. The contents of the session should contain all that Meteor needs to know in order to re-draw the application in exactly the same state as it is in right now.

One of the primary purposes of the router is to get the session into such a state when parsing URLs. For this reason, we can provide routing functions as endpoints for URLs; and we use those functions to set session variables based on the content of the URL. In our case, the only state our app requires is which room we are currently in – which we parse out of the URL and store in the `'currentRoomId'` session variable. And it works!

Finally, we need to get our links right; so we can do:

```
<td><a href="/rooms/{{ id}}">{{name}}</a></td>
```

client/chat.html

Modifying the data

Now that we have a collection holding our room data, we can begin changing it as we see fit. We can add new chats to a room, like so:

```
Rooms.update(Session.get('currentRoomId'), {$push: {messages: {auth
```

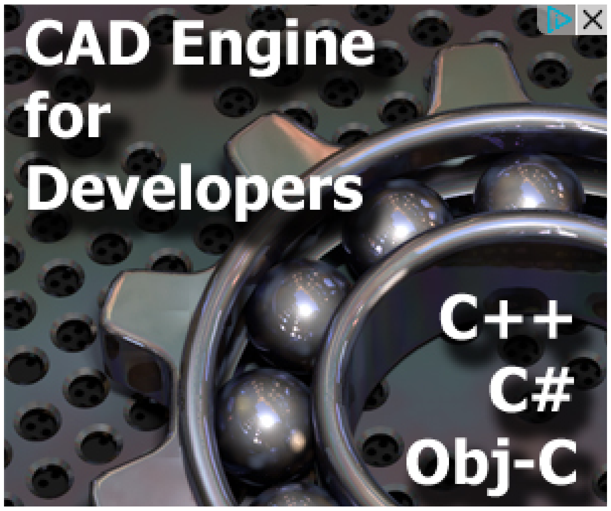
Or, we can even add a new room:

```
Rooms.insert({name: 'A New Room', members: 0, last_activity: 'Never
```

The next challenge is to wire up the forms to perform such transformations, which we'll leave as an exercise to the reader (or perhaps the next tutorial)!

> If you'd like to learn more about Meteor, be sure to check out our upcoming book!

Like 109 people like this. Be the first of your friends.



Tags: meteor

By Tom Coleman

Tom Coleman is one part of Percolate Studio, a web development shop with a focus on quality and user experience. He’s also the co–creator of Meteorite and the Atmosphere package repository.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more