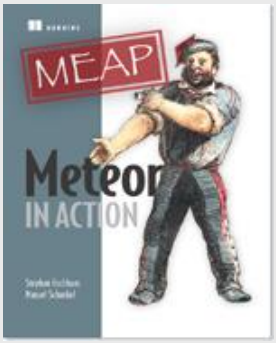

Integrating External APIs into your Meteor application

By Stephan Hochhaus

Many applications rely on external APIs to retrieve data. If APIs must be called from the server, calling the API usually takes longer than executing the method itself. Let's talk about how to integrate an external API via HTTP.

Many applications rely on external APIs to retrieve data. Getting information regarding your friends from Facebook, looking up the current weather in your area, or simply retrieving an avatar image from another website – there are endless uses for integrating additional data. They all share a common challenge: APIs must be called from the server, but an API usually takes longer than executing the method itself. You need to ensure that the result gets back to the client – even if it takes a couple of seconds. Let's talk about how to integrate an external API via HTTP.

Based on the IP address of a visitor, you can tell various information about their current location, e.g., coordinates, city or timezone. There is a simple API that takes an IPv4 address and returns all these tidbits as a JSON object. The API is called Telize.

Making RESTful calls with the http package

In order to communicate with RESTful external APIs such as Telize, you need to add the `http` package: `meteor add http`

While the `http` package allows you to make HTTP calls from both client and server, the API call in this example will be performed from the server only. Many APIs require you to provide an ID as well as a secret key to identify the application that makes an API request. In those cases you should always run your requests from the server. That way you never have to share secret keys with clients.

Figure 1 explains the basic concept. A user requests location information for an IP address (step 1). The client application calls a server method called `geoJsonforIp` (step 2) that makes an (asynchronous) call to the external API using the `HTTP.get()` method (step 3). The response (step 4) is a JSON object with information regarding the geographic location associated with an IP address, which gets sent back to the client via a callback (step 5).

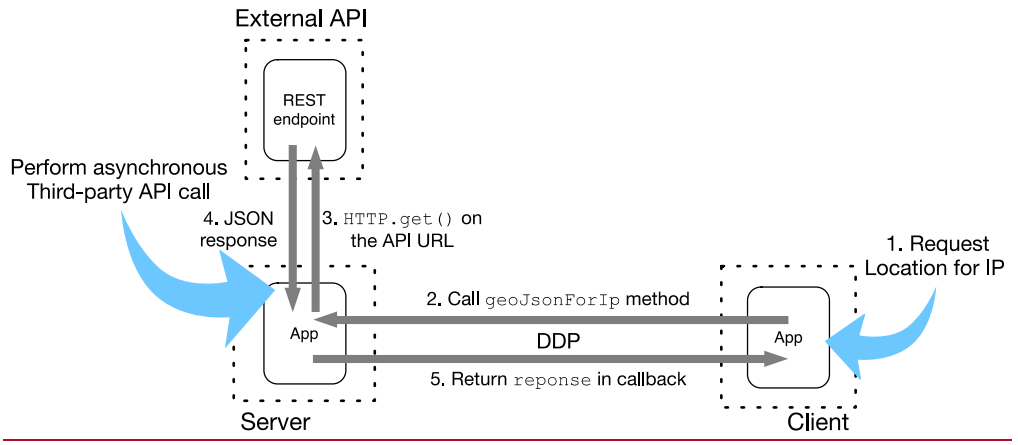

Figure 1: Data flow when making external API calls

Using a synchronous method to query an API

Let's add a method that queries telize.com for a given IP address as shown in listing 1. This includes only the bare essentials for querying an API for now.

Listing 1: Querying an external API using a synchronous method

```
Meteor.methods({
  // The method expects a valid IPv4 address
  'geoJsonForIp': function (ip) {
    console.log('Method.geoJsonForIp for', ip);
    // Construct the API URL
    var apiUrl = 'http://www.telize.com/geoip/' + ip;
    // query the API
    var response = HTTP.get(apiUrl).data;
    return response;
  }
});
```

Once the method is available on the server, querying the location of an IP works simply by calling the method with a callback from the client:

```
Meteor.call('geoJsonForIp', '8.8.8.8', function(err,res){
  console.log(res);
});
```

While this solution appears to be working fine there are two major flaws to this approach:

1. If the API is slow to respond requests will start queuing up.
2. Should the API return an error there is no way to return it back to the UI.

To address the issue of queuing, you can add an `unblock()` statement to the method:

```
this.unblock();
```

Calling an external API should always be done asynchronously. That way you can also return possible error values back to the browser, which will solve the second issue. Let's create a dedicated function for calling the API asynchronously to keep the method itself clean.

Using an asynchronous method to call an API

Listing 2 shows how to issue an `HTTP.get` call and return the result via a callback. It also includes error handling that can be shown on the client.

Listing 2: Dedicated function for asynchronous API calls

```
var apiCall = function (apiUrl, callback) {
  // #A try…catch allows you to handle errors

  try {
    var response = HTTP.get(apiUrl).data;
    // A successful API call returns no error
    // but the contents from the JSON response
    callback(null, response);
  } catch (error) {
    // If the API responded with an error message and a payload
    if (error.response) {
      var errorCode = error.response.data.code;
      var errorMessage = error.response.data.message;
    // Otherwise use a generic error message
    } else {
      var errorCode = 500;
      var errorMessage = 'Cannot access the API';
    }
    // Create an Error object and return it via callback
    var myError = new Meteor.Error(errorCode, errorMessage);
    callback(myError, null);
  }
}
```

Inside a `try…catch` block, you can differentiate between a successful API call (the `try` block) and an error case (the `catch` block). A successful call may return `null` for the error object of the callback, an error will return only an error object and `null` for the actual response.

There are different types of errors and you want to differentiate between a problem with accessing the API and an API call that got an error inside the returned response. This is what the `if` statement checks for – in case the `error` object has a `response` property both code and message for the error should be taken from it; otherwise you can display a generic error 500 that the API could not be accessed.

Each case, success and failure, return a callback that can be passed back to the UI. In order to make the API call asynchronous you need to update the method as shown in listing 3. The improved code unblocks the method and wraps the API call in a `wrapAsync` function.

Listing 3: Updated method for making asynchronous API calls

```
Meteor.methods({
  'geoJsonForIp': function (ip) {
    // avoid blocking other method calls from the same client
    this.unblock();
    var apiUrl = 'http://www.telize.com/geoip/' + ip;
    // asynchronous call to the dedicated API calling function
    var response = Meteor.wrapAsync(apiCall)(apiUrl);
    return response;
  }
});
```

Finally, to allow requests from the browser and show error messages you should add a template similar to listing 4.

Listing 4: Template for making API calls and displaying errors

```
<template name="telize">
  <p>Query the location data for an IP</p>
  <input id="ipv4" name="ipv4" type="text" />
  <button>Look up location</button>
  <!-- set the data context -->
  {{#with location}}
    <!-- if location has an error property, display it -->
    {{#if error}}
      <p>There was an error: {{error.errorType}} {{error.message}}!</p>
    {{else}}
      <p>The IP address {{location.ip}} is in {{location.city}}
         ({{location.country}}).</p>
    {{/if}}
  {{/with}}
</template>
```

The required JavaScript code for connecting the template with the method call is shown in listing 5. A Session variable called `location` is used to store the results from the API call. Clicking the button takes the content of the input box and sends it as a parameter to the `geoJsonForIp` method. The Session variable is set to the value of the callback. As a result you will be able to make calls from the browser just like in Figure 2.

Listing 5: Template helpers for making API calls

```
Template.telize.helpers({
  location: function () {
    //
    return Session.get('location');
  }
});
```

```
Template.telize.events({
  'click button': function (evt, tpl) {
    var ip = tpl.find('input#ipv4').value;
    Meteor.call('geoJsonForIp', ip, function (err, res) {
      // The method call sets the Session variable to the callback value
      if (err) {
        Session.set('location', {error: err});
      } else {
        Session.set('location', res);
        return res;
      }
    });
  }
});
```

And that's how to integrate an external API via HTTP!

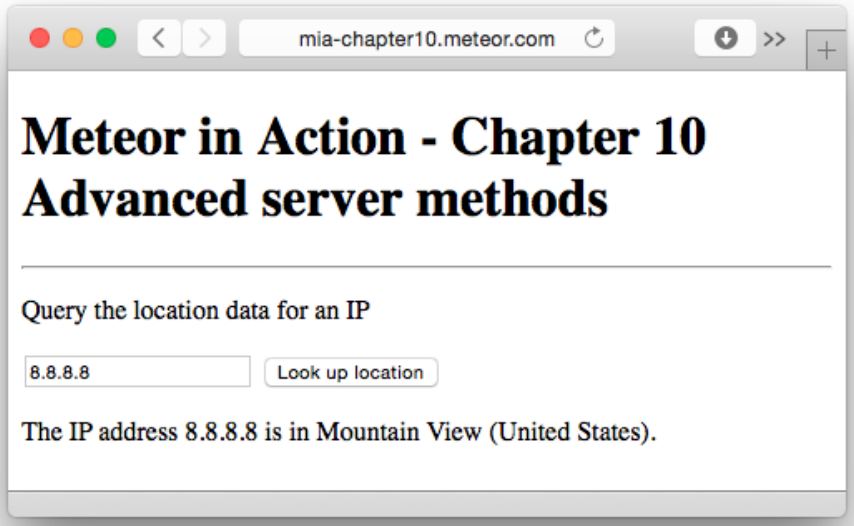

Figure 2: Querying the Telize API from Meteor