FREE eBook

# LEARNING meteor

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#meteor

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: meteor

It is an unofficial and free meteor ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official meteor.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with meteor

## Remarks

Meteor is a **full-stack** JavaScript platform for developing modern Web and mobile applications.

Within *one* project, you are able to build your client (browser and/or hybrid mobile App for Android and/or iOS) *and* server sides.

Reference pages:

- Meteor Guide
- Meteor API Docs
- Meteor Tutorials
- Meteor Forums

## Versions

| Version | Release Date |
|---|---|
| 0.4.0 | 2012-08-30 |
| 0.5.0 | 2013-10-17 |
| 0.6.0 | 2013-04-04 |
| 0.7.0 | 2013-12-20 |
| 0.8.0 | 2014-04-21 |
| 0.9.0 | 2014-08-26 |
| 0.9.1 | 2014-09-04 |
| 0.9.2 | 2014-09-15 |
| 0.9.3 | 2014-09-25 |
| 0.9.4 | 2014-10-13 |
| 1.0.1 | 2014-12-09 |
| 1.0.2 | 2014-12-19 |
| 1.0.3.1 | 2014-12-09 |
| 1.1.0 | 2015-03-31 |

| Version | Release Date |
|---|---|
| 1.2.0 | 2015-09-21 |
| 1.3.0 | 2016-03-27 |
| 1.4.0 | 2016-07-25 |
| 1.5.0 | 2017-05-30 |

## Examples

### Getting Started

## Install Meteor

### On OS X and Linux

Install the latest official Meteor release from your terminal:

```
$ curl https://install.meteor.com/ | sh
```

### On Windows

Download the official Meteor installer here.

## Create your app

Once you've installed Meteor, create a project:

```
$ meteor create myapp
```

## Run it

Run it locally:

```
$ cd myapp
$ meteor npm install
$ meteor
```

**Note:** Meteor server running on: http://localhost:3000/

Then head to http://localhost:3000 to see your new Meteor application.

---

- Read more about getting started with Meteor at the [Meteor Guide].
- Explore Meteor Packages at atmosphere - a modern, fast, well engineered package manager.

## Sample apps

Meteor has several sample apps built-in. You can create a project with one of them and learn from how it was built. To create a sample app, install Meteor (see Getting Started) and then type:

```
meteor create --example <app name>
```

For example to create a sample `todos` app, write:

```
meteor create --example todos
```

To get a list of all sample apps, type:

```
meteor create --list
```

## Managing Packages

Meteor has it's own package repository on atmospherejs.com

You can add new packages from atmosphere by running:

```
meteor add [package-author-name:package-name]
```

For example:

```
meteor add kadira:flow-router
```

Similarly, you can remove the same package by:

```
meteor remove kadira:flow-router
```

To see current packages in your project, type:

```
meteor list
```

List of packages can also be found in the file `./meteor/packages`. To add a package add the package name in this file and to remove delete it.

To add a package locally, (e.g. unpublished packages or edited version of published packages), save the package in `packages` folder in the root.

Starting with version 1.3, Meteor **added support for npm packages**.

You can use the `npm` command inside Meteor project's directory as you would normally do without Meteor, or with the `meteor npm` command, which will use the bundled version of npm.

### Understanding build progress

Sometimes builds take longer than expected. There are a few environment variables you can set to better understand what's happening during the build process.

```
METEOR_DEBUG_BUILD=1        (logs progress)
METEOR_PROFILE=<n>          (logs time spent)
METEOR_DEBUG_SPRINGBOARD=1 (?)
METEOR_DEBUG_SQL=1          (logs SQLITE calls)
METEOR_PROGRESS_DEBUG=1     (? looks like it might be useful, but seems confusing)
```

Where `<n>` is a number of ms. Any process taking longer than this will be logged.

## Linux/OSX Example

```
export METEOR_DEBUG_BUILD=1
export METEOR_PROFILE=100
meteor
```

## Windows Example

```
set METEOR_DEBUG_BUILD=1
set METEOR_PROFILE=100
meteor
```

### Checking the Version of the Meteor Tool & Meteor Projects

## Meteor Tool

To check the installed version of the Meteor tool, just run the following command outside of any Meteor projects:

```
meteor --version
```

To get a list of all official (recommended) Meteor releases, run:

```
meteor show METEOR
```

## Meteor Projects

If you want to check the project version of Meteor, you can also execute the following command inside a project:

```
meteor --version
```

or just print content of the file `.meteor/release`:

```
cat .meteor/release
```

In case you want to check the version of the packages which are currently installed in your Meteor project, print the content of the file `.meteor/versions`:

```
cat .meteor/versions
```

## Meteor Website

To see which version of Meteor a Meteor based website is running, dump the contents of `Meteor.release` in your browsers console while visiting the website:

```
Meteor.release
```

### Updating Meteor Projects & Installed Packages

The Meteor Tool will notify you when a newer release is available.

To update Meteor projects to the latest release, execute the following command inside a Meteor project:

```
meteor update
```

In case you want to update your Meteor project to a specific Meteor release, run the following command inside the project:

```
meteor update --release <release>
```

If you want to update all non-core packages, run:

```
meteor update --packages-only
```

You can also update specific packages by passing their names as a command line argument to `meteor update`, for example:

```
meteor update [packageName packageName2 ...]
```

## Build Mobile Apps

Meteor uses Cordova to package your application into a *hybrid* Mobile App. Once packaged, the App can be distributed like native Apps (through Apple App Store, Google Play Store, etc.)

1. **Add** the target platform(s) to your Meteor project:

```
meteor add-platform android
meteor add-platform ios # Only available with Mac OS
```

2. **Install** the Android SDK and/or Xcode (for iOS, requires Mac OS).

3. **Run** your project (start with development mode):

```
meteor run android # You may need to configure a default Android emulator first
```

For iOS (only available with Mac OS):

```
meteor run ios # This will auto start an iOS simulator
```

4. **Build** your App package for distribution:

```
meteor build <output_folder> --server <url_app_should_connect_to>
```

This will create `android` and/or `ios` folder(s) alongside your server bundle.

- The `android` folder contains the `release-unsigned.apk` file that you need to sign and zip align.
- The `ios` folder contains the Xcode project that you need to sign.

See also the Meteor Mobile Apps topic.
Reference page: Meteor Guide > Build > Mobile

Read Getting started with meteor online: https://riptutorial.com/meteor/topic/439/getting-started-with-meteor

# Chapter 2: Acceptance Testing (with Nightwatch)

## Remarks

Nightwatch has been providing Acceptance and End-to-End testing for Meteor apps since v0.5 days, and has managed migrations from PHP to Spark to Blaze and to React; and all major Continuous Integration platforms. For additional help, please see:

Nightwatch API Documentation
Nightwatch.js Google Group

## Examples

### App Surface Area

At it's most basic level, acceptance testing is essentially black-box testing, which is fundamentally concerned with testing inputs and outputs of a closed system. As such, there are three essential features to acceptance testing: locating a resource, reading data, and writing data. When it comes to browsers and webapps, these three features basically boil down to the following:

1. Load a webpage or application view
2. Inspect user interface elements (i.e. DOM)
3. Trigger an event / simulate a user interaction

We call this the surface area of the application. Surface area is anything that a user sees or experiences. It's the outside of a blackbox system. And since users interact with modern web applications on video screens using web browsers, our surface coverage is defined by universal resource locators (URLs) and viewports. And so our very first walkthrough starts off looking something like the following:

```
module.exports = {
  "Hello World" : function (client) {
    client
      // the location of our Meteor app
      .url("http://localhost:3000")

      // the size of the viewport
      .resizeWindow(1024, 768)

      // test app output
      .verify.elementPresent('h1')
      .verify.containsText('h1', "Welcome to Meteor!")
      .verify.containsText('p', "You've pressed the button 0 times")
      .verify.elementPresent('button')

      // simulate user input
      .click('button').pause(500)
      // test app output again, to make sure input worked
      .verify.containsText('p', "button 1 times")

      // saving a copy of our viewport pixel grid
      .saveScreenshot('tests/nightwatch/screenshots/homepage.png')
      .end();
  }
};
```

## Custom Commands

Nightwatch supports creating custom commands that can simulating keystrokes, mouse clicks, and other inputs. A custom command can be chained with other Nightwatch commands, like so:

```
module.exports = {
  "Login App" : function (client) {
    client
      .url("http://localhost:3000")
      .login("janedoe@somewhere.com", "janedoe123")
      .end();
  }
};
```

To enable this, define a command in `./tests/nightwatch/commands/login` like so:

```
exports.command = function(username, password) {

  this
    .verify.elementPresent('#login')

      // we clear the input in case there's any data remaining from previous visits
      .clearValue("#emailInput")
      .clearValue("#passwordInput")

      // we simulate key presses
      .setValue("#emailInput", username)
      .setValue("#passwordInput", password)

    // and we simulate a mouse click
    .click("#signInToAppButton").pause(1000)

  return this; // allows the command to be chained.
};
```

To make this all work, you will need to add `id` attributes to your login page. At some level, it will need to roughly look something like the following:

```
<template name="login">
  <div id="login">
    <input id="emailInput" name="email" type="email" />
    <input id="passwordInput" name="password" type="password" />
    <button id="#signInToAppButton">Sign In</button>
  </div>
</template>
```

## Inspecting Meteor Objects on the Client

Since Nightwatch has access to the browser console, it's possible to inspect client side objects using the `.execute()` API. In the following example, we're checking the Session object for a particular session variable. First, we begin by creating the file `./tests/nightwatch/api/meteor/checkSession`, where we will keep the following command:

```
// syncrhonous version; only works for checking javascript objects on client
exports.command = function(sessionVarName, expectedValue) {
  var client = this;
  this
    .execute(function(data){
      return Session.get(data);
    }, [sessionVarName], function(result){
      client.assert.ok(result.value);
      if(expectedValue){
        client.assert.equal(result.value, expectedValue);
      }
    })
    return this;
};
```

We can then chain it like so:

```
module.exports = {
  "Check Client Session" : function (client) {
    client
      .url("http://localhost:3000")
      .checkSession("currentUser", "Jane Doe")
      .end();
  }
};
```

## Forms & Input Types

To upload a file, you'll first need to create a /data directory, and add the file you'll want to upload.

```
tests/nightwatch/data/IM-0001-1001.dcm
```

Your form will need an input with type of file. (Some people don't like the styling options this input provides; and a common pattern is to make this input hidden; and to have another button on the page click it on behalf of the user.)

```
<form id="myform">
    <input type="file" id="fileUpload">
    <input type="text" name="first_name">
    <input type="text" name="last_name">

    <input type="date" name="dob_month">
    <input type="date" name="dob_day">
    <input type="date" name="dob_year">

    <input type="radio" name="gender" value="M">
    <input type="radio" name="gender" value="F">
    <input type="radio" name="gender" value="O">

    <input type="select" name="hs_graduation_year">
    <input type="text" name="city">
    <input type="select" name="state">

    <input type="submit" name="submit" value="Submit">
</form>
```

Your tests will then need to use setValue() and resolve the path to the local file asset.

```
module.exports = {
  "Upload Study" : function (client) {
    console.log(require('path').resolve(__dirname +  '/../data' ));

    var stringArray = "Chicago";

    client
      .url(client.globals.url)
      .verify.elementPresent("form#myform")

      // input[type="file"]
      .verify.elementPresent("input#fileUpload")
      .setValue('input#fileUpload', require('path').resolve(__dirname + '/../data/IM-0001-
1001.dcm'))

      // input[type="text"]
      .setValue('input[name="first_name"]', 'First')
      .setValue('input[name="last_name"]', 'Last')

      // input[type="date"]
      .click('select[name="dob_month"] option[value="3"]')
      .click('select[name="dob_day"] option[value="18"]')
      .click('select[name="dob_year"] option[value="1987"]')

      // input[type="radio"]
      .click('input[name="gender"][value="M"]')

      // input[type="number"]
      .click('select[name="hs_graduation_year"] option[value="2002"]')

      // input[type="text"]
      // sometimes Nightwatch will send text faster than the browser can handle
      // which will cause skipping of letters.  In such cases, we need to slow
      // Nightwatch down; which we do by splitting our input into an array
      // and adding short 50ms pauses between each letter
      for(var i=0; i < userIdArray.length; i++) {
        client.setValue('input[name="city"]', stringArray[i]).pause(50)
      }

      // input[type="select"]
      // after an array input above, we need to resume our method chain...
      client.click('select[name="state"] option[value="CA"]')

      // input[type="number"]
      .setValue('input[name="zip"]', '01234')

      //input [ type="submit" ]
      .click('button[type="submit"]')
      .end();
  }
};
```

Credit to Daniel Rinehart for inpsiring this example.

## Components & Page Objects

Page Objects are similar to Custom Commands; except they are collections of custom commands that are associated with a specific UI component. This works extremely well with modern component based design, such as in React.

```
module.exports = {
  url: 'http://localhost:3000/login',
  commands: [{
  login: function(email, password) {
    return this
      .clearValue('input[name="emailAddress"]')
      .clearValue('input[name="password"]')

      .setValue('input[name="emailAddress"]', email)
      .setValue('input[name="password"]', password)

      .verify.elementPresent('#loginButton')
      .click("#loginButton");
  },
  clear: function() {
    return this
      .waitForElementVisible('@emailInput')
      .clearValue('@emailInput')
      .clearValue('@passInput')
      .waitForElementVisible('@loginButton')
      .click('@loginButton')
  },
  checkElementsRendered: function(){
    return this
      .verify.elementPresent("#loginPage")
      .verify.elementPresent('input[name="emailAddress"]')
      .verify.elementPresent('input[name="password"]')
  },
  pause: function(time, client) {
    client.pause(time);
    return this;
  },
  saveScreenshot: function(path, client){
    client.saveScreenshot(path);
    return this;
  }
}],
  elements: {
    emailInput: {
      selector: 'input[name=email]'
    },
    passInput: {
      selector: 'input[name=password]'
    },
    loginButton: {
      selector: 'button[type=submit]'
    }
  }
};
```

The only caveat with using the PageObject pattern in testing components, is that the implementation breaks the method chaining flow that the native Nightwatch `verify.elementPresent` provides. Instead, you'll need to assign the page object to a variable, and instantiate a new method chain for each page. A reasonable price to pay for a consistent and reliable pattern for testing code reuse.

```
module.exports = {
  tags: ['accounts', 'passwords', 'users', 'entry'],
  'User can sign up.': function (client) {

    const signupPage = client.page.signupPage();
    const indexPage = client.page.indexPage();

    client.page.signupPage()
      .navigate()
      .checkElementsRendered()
      .signup('Alice', 'Doe', 'alice@test.org', 'alicedoe')
      .pause(1500, client);

    indexPage.expect.element('#indexPage').to.be.present;
    indexPage.expect.element('#authenticatedUsername').text.to.contain('Alice Doe');
  },
}
```

Read Acceptance Testing (with Nightwatch) online: https://riptutorial.com/meteor/topic/6454/acceptance-testing--with-nightwatch-

# Chapter 3: Accessing Meteor build machines from Windows

## Remarks

On Mac and Linux, the `meteor` command line tool assumes that the `ssh` command line tool, used to make secure connections to other computers, is always present. On Windows, this tool needs to be installed. Below are listed two options for setting it up and using it.

## Examples

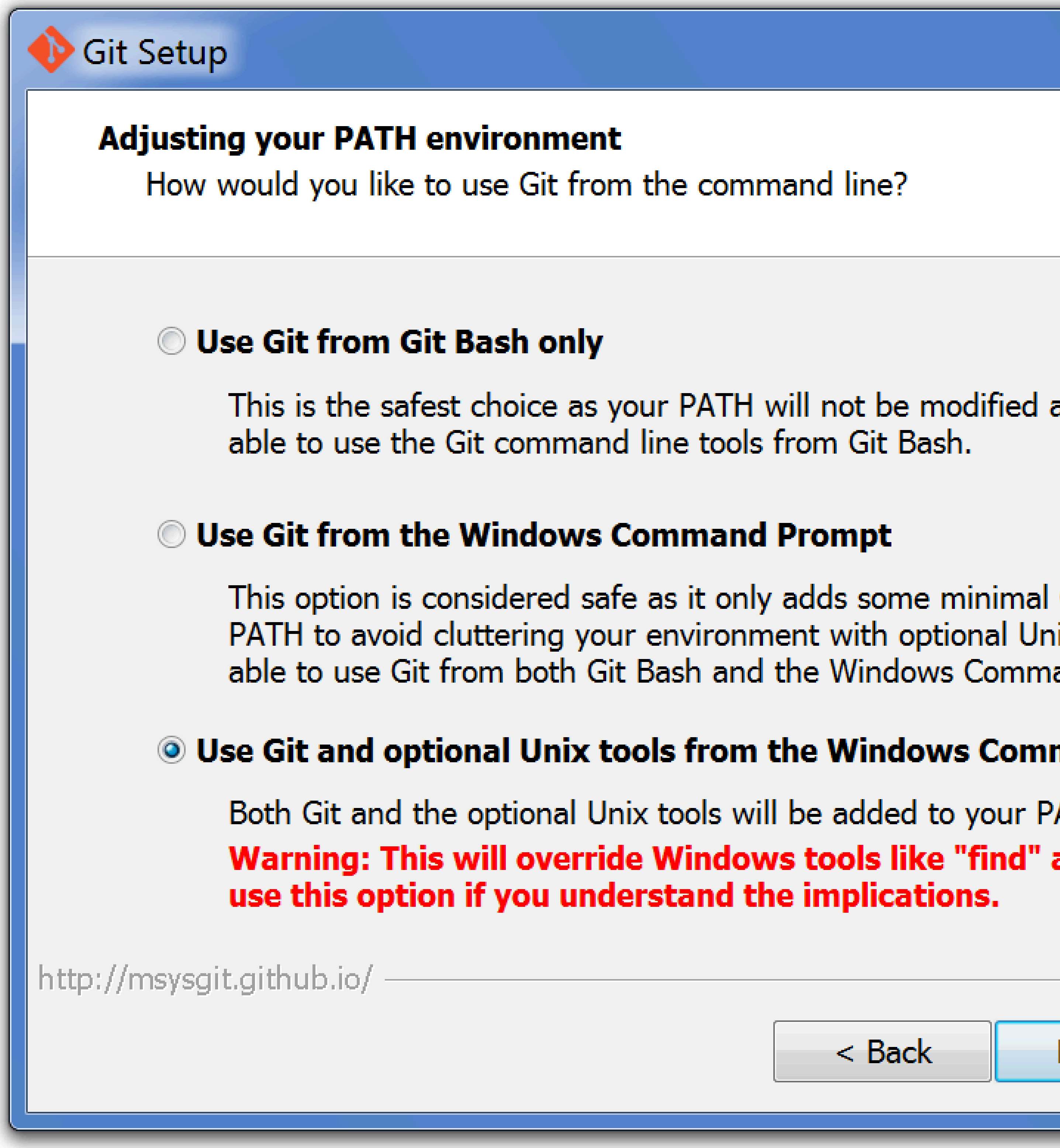
### Using PuTTY (Advanced)

If you don't want to add Unix commands to your PATH on Windows, you can download a standalone SSH client like PuTTY. Download PuTTY here, then follow the instructions below to get a build machine.

1. Call `meteor admin get-machine <os-architecture> --json`
2. Copy and save the private key from the returned JSON data
3. Follow the directions here to convert the private key into a format that PuTTY accepts
4. Enter the hostname, username, and private key into PuTTY, and you're good to go!

### Using Cygwin (Unix tools on Windows)

The easiest way to get up and running is to install Git for Windows from this download page, and select "Use Git and optional Unix tools from the Windows Command Prompt" as in the screenshot below.


Git Setup

**Adjusting your PATH environment**

How would you like to use Git from the command line?

○ **Use Git from Git Bash only**

This is the safest choice as your PATH will not be modified a
able to use the Git command line tools from Git Bash.

○ **Use Git from the Windows Command Prompt**

This option is considered safe as it only adds some minimal
PATH to avoid cluttering your environment with optional Uni
able to use Git from both Git Bash and the Windows Comma

● **Use Git and optional Unix tools from the Windows Comm**

Both Git and the optional Unix tools will be added to your P
**Warning: This will override Windows tools like "find" a
use this option if you understand the implications.**

http://msysgit.github.io/

After this, `meteor admin get-machine <os-architecture>` will work exactly as it does on Linux and Mac. Keep in mind that you might need to restart your terminal to get the new commands.

Read Accessing Meteor build machines from Windows online: https://riptutorial.com/meteor/topic/518/accessing-meteor-build-machines-from-windows

# Chapter 4: Assets

## Examples

### Accessing Assets on the Server

Static server assets must be placed in the `private` directory.

## Text files

Text files can be accessed by using the **`Assets.getText(assetPath, [asyncCallback])`** method. For example, the following JSON file is named `my_text_asset.json` and is located in the `private` directory:

```
{
    "title": "Meteor Assets",
    "type": "object",
    "users": [{
        "firstName": "John",
        "lastName": "Doe"
    }, {
        "firstName": "Jane",
        "lastName": "Doe"
    }, {
        "firstName": "Matthias",
        "lastName": "Eckhart"
    }]
}
```

You can access this file on the server by using the following code:

```
var myTextAsset = Assets.getText('my_text_asset.json');
var myJSON = JSON.parse(myTextAsset);
console.log(myJSON.title); // prints 'Meteor Assets' in the server's console
```

## Binary files

If you want to access assets on the server as an EJSON binary, use the **`Assets.getBinary(assetPath, [asyncCallback])`** method. Here's a code example for accessing an image named `my_image.png` which is located in the `private/img` directory:

```
var myBinaryAsset = Assets.getBinary('img/my_image.png');
```

Read Assets online: https://riptutorial.com/meteor/topic/3379/assets

# Chapter 5: Background tasks

## Remarks

The package **cron-tick** is a very simple package for background tasks but it does not support multiple processes, if you run your app in multiple processes (or containers) use **percolate:synced-cron** instead.

## Examples

### Simple cron

Use the package **percolate:synced-cron**

Define a job:

```
 SyncedCron.add({
  name: 'Find new matches for a saved user filter and send alerts',
  schedule: function(parser) {
    // parser is a later.parse object
    return parser.text('every 10 minutes');
  },
  job: function() {
    user.alerts.map(a => a.findMatchesAndAlert());
  }
});
```

Starting up your defined jobs:

```
SyncedCron.start();
```

It supports syncronizing jobs between multiple processes, like Galaxy with more than 1 container.

Read Background tasks online: https://riptutorial.com/meteor/topic/4772/background-tasks

# Chapter 6: Basic Codeship Setup for Automated Testing

## Examples

### Setup Codeship

- Go to Codeship.com and create an account (or login)
- Create a new project
- Import your project via Github or Bitbucket
- On the screen "Configure Your Tests" use these commands:
  - Select "I want to create my own custom commands" from the "Select your technology to prepopulate basic commands" dropdown.
  - Enter the following commands:

    ```
    curl -o meteor_install_script.sh https://install.meteor.com/
    chmod +x meteor_install_script.sh
    sed -i "s/type sudo >\/dev\/null 2>&1/\ false /g" meteor_install_script.sh
    ./meteor_install_script.sh
    export PATH=$PATH:~/.meteor/
    meteor --version
    meteor npm install
    ```

  - Leave the test commands like this:

    ```
    npm test
    ```

- Push a new commit to Github / Bitbucket
- That's it

### Prepare the Project

- Write some tests
- Install dispatch:mocha-phantomjs:

  ```
  meteor add dispatch:mocha-phantomjs
  ```

- Add a test-command to your package.json.

  ```
  {
    "name": "awesome meteor package",
    "scripts": {
      "test": "meteor test --driver-package dispatch:mocha-phantomjs --once"
    }
  }
  ```

- Make sure that you can run `npm test` in your project root.

Read Basic Codeship Setup for Automated Testing online:
https://riptutorial.com/meteor/topic/6741/basic-codeship-setup-for-automated-testing

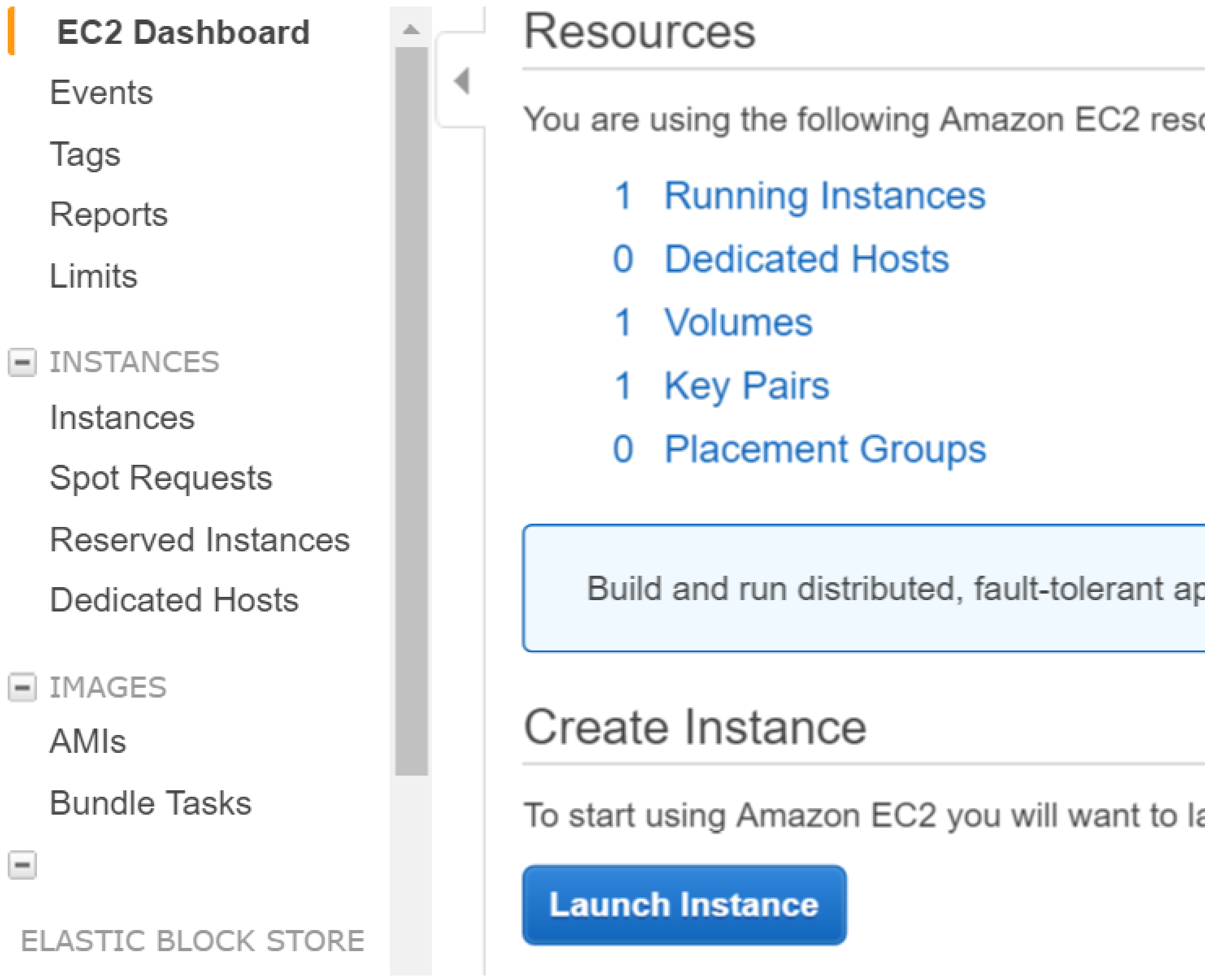
# Chapter 7: Beginner guide to Installing Meteor 1.4 on AWS EC2

## Examples

### Signup for AWS Service

Since lots of beginners are confused about cloud hosting.I am writing this guide to walk through setting meteor on aws with ubuntu os. If you already have your instance running feel free to skip this step and go straight to installing meteor on aws.

Login into AWS Console.Select EC2. Go to EC2 Dashboard. Under Create Instance click launch instance.



Resources

You are using the following Amazon EC2 reso

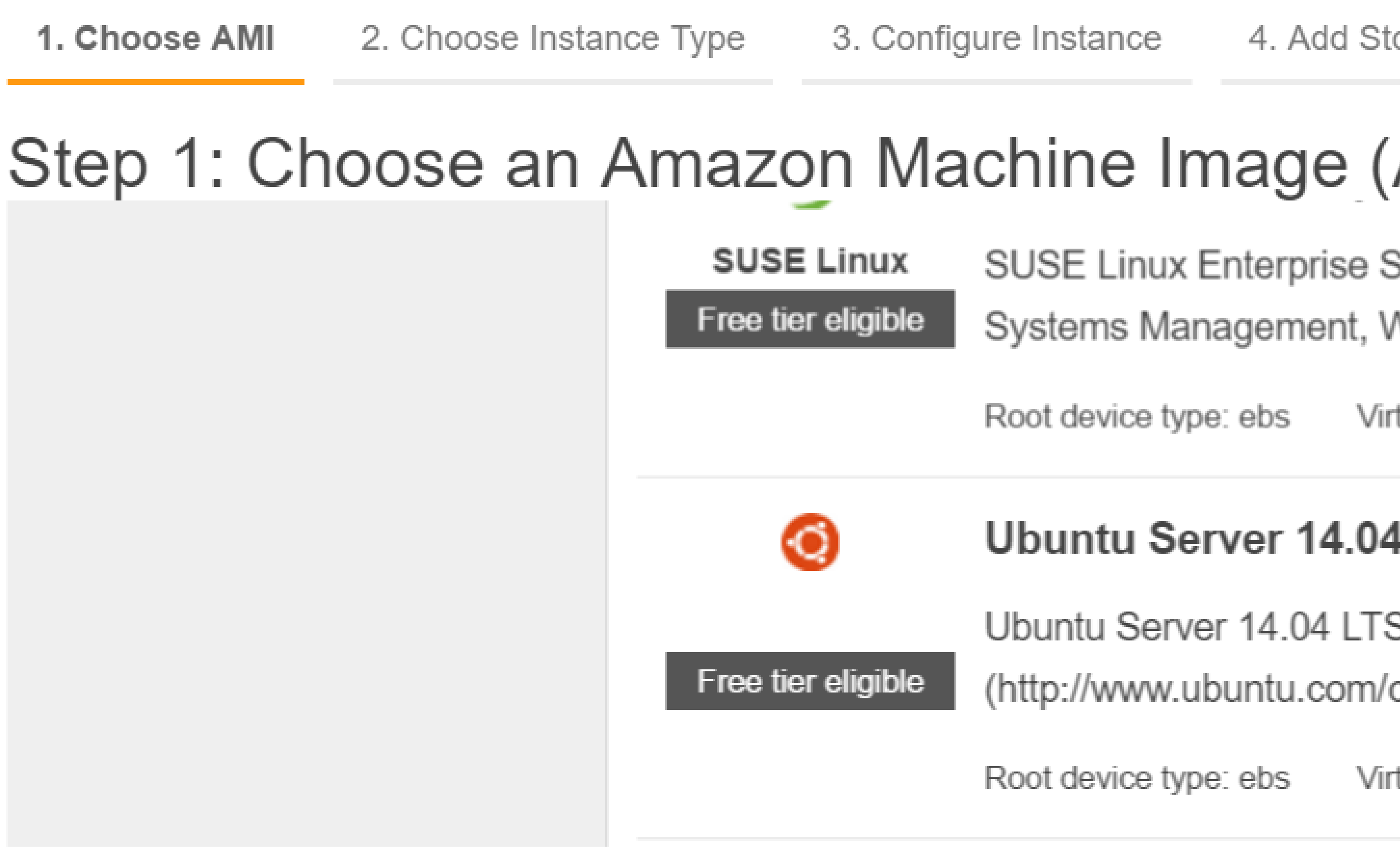
1 Running Instances
0 Dedicated Hosts
1 Volumes
1 Key Pairs
0 Placement Groups

Build and run distributed, fault-tolerant ap

Create Instance

To start using Amazon EC2 you will want to la

Select ubuntu instance in next step



Create key pair & download private key to your local machine.

Login via shell to aws (using private key, make sure private key is in your path or run command from directory which contains private key)

```
ssh -i "myprivatekey.pem" ubuntu@ec2-xx-xx-xx-xx.ap-south-1.compute.amazonaws.com
```

ec2-xx-xx-xx-xx.ap-south-1.compute.amazonaws.com is public dns instance name on amazon console. ubuntu is username. You can also use public ip address.

**STEPS TO INSTALL METEOR ON AWS INSTANCE (using mupx)**

1. copy private key from local machine to aws server ssh folder

example `/home/ubuntu/.ssh/myprivatekey.pem`

2. update packager to latest version

```
sudo apt-get update
```

3. install python software properties

```
 sudo apt-get install python-software-properties
```

4. install npm and node(optionally also install nvm)

```
sudo apt-get install npm
```

Install nvm

```
curl https://raw.githubusercontent.com/creationix/nvm/v0.11.1/install.sh | bash
```

Install node

```
nvm install 4.4.7

nvm use 4.4.7
```

5. Install aws cli

```
sudo apt-get install awscli
```

6. Install meteor up

```
 sudo npm install -g mupx

 sudo npm install -g mupx-letsencrypt
```

(meteor 1.4 is currently available only by mpux-letsencrypt)

7. Initialize mupx by going into your project directory or create new directory if not exists

```
mupx-letsencrypt init
```

If you get error like below , then may legacy node is there you need to create link

```
/usr/bin/env: node: No such file or directory



sudo ln -s /usr/bin/nodejs /usr/bin/node
```

8. Install meteor

```
curl https://install.meteor.com | /bin/sh
```

9. edit mup.json (Make sure to fill username:ubuntu and correct location of private key from step 1)

   use nano file editor (to edit on files on ubuntu, also can use vi)

```
 nano mup.json
```

Example mup.json

```
    {
  // Server authentication info
  "servers": [
    {
      "host": "ec2-xx-xx-xx-xx.ap-south-1.compute.amazonaws.com",
      "username": "ubuntu",
      //"password": "password",
      // or pem file (ssh based authentication)
      "pem": "~/.ssh/myprivatekey.pem",
      // Also, for non-standard ssh port use this
      //"sshOptions": { "port" : 49154 },
      // server specific environment variables
      "env": {}
    }
  ],

  // Install MongoDB on the server. Does not destroy the local MongoDB on future setups
  "setupMongo": true,

  // WARNING: Node.js is required! Only skip if you already have Node.js installed on server.
  "setupNode": false,

  // WARNING: nodeVersion defaults to 0.10.36 if omitted. Do not use v, just the version
number.
  //"nodeVersion": "4.4.7",

  // Install PhantomJS on the server
  "setupPhantom": true,

  // Show a progress bar during the upload of the bundle to the server.
  // Might cause an error in some rare cases if set to true, for instance in Shippable CI
  "enableUploadProgressBar": true,

  // Application name (no spaces).
  "appName": "my-app",

  // Location of app (local directory). This can reference '~' as the users home directory.
  // i.e., "app": "/Users/ubuntu/my-app",
  // This is the same as the line below.
  "app": "/Users/ubuntu/my-app",

  // Configure environment
  // ROOT_URL must be set to https://YOURDOMAIN.com when using the spiderable package & force
SSL
  // your NGINX proxy or Cloudflare. When using just Meteor on SSL without spiderable this is
not necessary
  "env": {
    "PORT": 80,
    "ROOT_URL": "http://myapp.com",
     // only needed if mongodb is on separate server
    "MONGO_URL": "mongodb://url:port/MyApp",
    "MAIL_URL":   "smtp://postmaster%40myapp.mailgun.org:adj87sjhd7s@smtp.mailgun.org:587/"
  },

  // Meteor Up checks if the app comes online just after the deployment.
  // Before mup checks that, it will wait for the number of seconds configured below.
  "deployCheckWaitTime": 60
}
```

10. Setup Meteor including mongo running following command in project directory.

```
mupx-letsencrypt setup
```

11. deploy project using mupx

```
mupx-letsencrypt deploy
```

Some helpful commands

To check mupx logs

```
mupx logs -f
```

To check Docker

```
 docker -D info
```

To check network status

```
netstat -a
```

To check current running process including cpu and memory utilization

```
 top
```

Install mongo client to get mongo shell acccess on aws

```
sudo apt-get install mongodb-clients
```

To run mongodb queries

```
mongo projectName
```

Once Inside mongo shell run

```
 db.version()
 db.users.find()
```

Thanks arunoda for providing wonderful tool https://github.com/arunoda/meteor-up

Thanks mupx-letsencrypt team for good work. https://www.npmjs.com/package/mupx-letsencrypt

Read Beginner guide to Installing Meteor 1.4 on AWS EC2 online: https://riptutorial.com/meteor/topic/4773/beginner-guide-to-installing-meteor-1-4-on-aws-ec2

# Chapter 8: Blaze Templating

## Introduction

Blaze is a powerful library for creating user interfaces by writing dynamic, reactive HTML templates. Blaze templating allows for loops and conditional logic to be used directly in HTML markup. This section explains and demonstrates the proper usage of templating in Meteor.js with Blaze.

## Examples

### Populate a template from a method call

```
<template name="myTemplate">
  {{#each results}}
    <div><span>{{name}}</span><span>{{age}}</span></div>
  {{/each}}
</template>
```

```
Template.myTemplate.onCreated(function() {
  this.results = new ReactiveVar();
  Meteor.call('myMethod', (error, result) => {
    if (error) {
      // do something with the error
    } else {
      // results is an array of {name, age} objects
      this.results.set(result);
    }
  });
});

Template.myTemplate.helpers({
  results() {
    return Template.instance().results.get();
  }
});
```

### Data context of a template

Whenever a template is called upon, the default data context of the template is implicitly gained from the caller as in example the childTemplate gains the data context of the parentTemplate i.e caller template

```
<template name="parentTemplate">
    {{#with someHelperGettingDataForParentTemplate}}
    <h1>My name is {{firstname}} {{lastname}}</h1>
    //some stuffs here
    {{> childTemplate}}
    {{/with}}
</template>
```

In the above situation,whatever data the helper extracts for parent template are automatically gained by childTemplate.For example,the {{firstname}} and {{lastname}} can be accessed from childTemplate as well as shown below.

```
<template name="childTemplate">
<h2>My name is also {{firstname}} {{lastname}}</h2>
</template>
```

We can even explicitly define the data context of the childTemplate by passsing arguments to the template like in below example.

```
<template name="parentTemplate">
    {{#with someHelperGettingDataForParentTemplate}}
    <h1>My name is {{firstname}} {{lastname}}</h1>
    //some stuffs here
    {{> childTemplate childData=someHeplerReturningDataForChild}}
    {{/with}}
</template>
```

Assuming the helper **someHelperReturningDataForChild** returns object like {profession:"Meteor Developer",hobby:"stackoverflowing"},this particular object will be the explicit data context for the childTemplate. Now in child template we can do something like

```
<template name="childTemplate">
    <h2>My profession is {{profession}}</h2>
    <h3>My hobby is {{hobby}}</h3>
</template>
```

## Template Helpers

Template helpers are an essential part of Blaze and provide both business logic and reactivity to a Template. It is important to remember that Template helpers are actually reactive computations that are rerun whenever their dependencies change. Depending on your needs, Template helpers can be defined globally or scoped to a specific template. Examples of each Template helper definition approach is provided below.

1. Example of a Template helper scoped to a single template.

First define your template:

```
<template name="welcomeMessage">
  <h1>Welcome back {{fullName}}</h1>
</template>
```

Then define the Template helper. This assumes that the data context of the template contains a firstName and lastName property.

```
Template.welcomeMessage.helpers({
  fullName: function() {
    const instance = Template.instance();
    return instance.data.firstName + ' ' + instance.data.lastName
  },
});
```

2. Example of a global Template helper (this helper can be used from within any Template)

First register the helper:

```
Template.registerHelper('equals', function(item1, item2) {
  if (!item1 || !item2) {
    return false;
  }

  return item1 === item2;
});
```

With the `equals` helper defined, I can now use it within any template:

```
<template name="registration">
  {{#if equals currentUser.registrationStatus 'Pending'}}
    <p>Don't forget to complete your registration!<p>
  {{/if}}
</template>
```

Read Blaze Templating online: https://riptutorial.com/meteor/topic/2434/blaze-templating

# Chapter 9: Blaze User Interface Recipes (Bootstrap; No jQuery)

## Remarks

The above Blaze examples are highly compatible with the http://bootsnipp.com/ library, which only provides the HTML and CSS for components, and leaves the javascript up to the developer. This allows for components to share the same underlying sorting, filtering, query, and cursor methods.

## Examples

### Drop Down Menu

The following example creates a Bootstrap Drop-Down menu, using only Blaze and no JQuery.

**Document Object Model**

```
    <nav class="nav navbar-nav">
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">{{getSelectedValue}} <span
class="glyphicon glyphicon-user pull-right"></span></a>
        <ul class="fullwidth dropdown-menu">
          <li id="firstOption" class="fullwidth"><a href="#">15 Minutes <span class="glyphicon
glyphicon-cog pull-right"></span></a></li>
          <li class="divider"></li>
          <li id="secondOption"><a href="#">30 Minutes <span class="glyphicon glyphicon-stats
pull-right"></span></a></li>
          <li class="divider"></li>
          <li id="thirdOption"><a href="#">1 Hour <span class="badge pull-right"> 42
</span></a></li>
          <li class="divider"></li>
          <li id="fourthOption"><a href="#">4 Hour <span class="glyphicon glyphicon-heart
pull-right"></span></a></li>
          <li class="divider"></li>
          <li id="fifthOption"><a href="#">8 Hours <span class="glyphicon glyphicon-log-out
pull-right"></span></a></li>
        </ul>
      </li>
    </nav>
```

**Javascript**

```
Template.examplePage.helpers({
  getSelectedValue:function(){
    return Session.get('selectedValue');
  }
});
Template.dropDownWidgetName.events({
  'click #firstOption':function(){
    Session.set('selectedValue', 1);
  },
  'click #secondOption':function(){
    Session.set('selectedValue', "blue");
  },
  'click #thirdOption':function(){
    Session.set('selectedValue', $('#thirdOption').innerText);
  },
  'click #fourthOption':function(){
    Session.set('selectedValue', Session.get('otherValue'));
  },
  'click #fifthOption':function(){
    Session.set('selectedValue', Posts.findOne(Session.get('selectedPostId')).title);
  },
});
```

## Navbars

A very common task is to create responsive navbars and to create action/footer bars that have different controls based on what page a user is on, or what role a user belongs to. Lets go over how to make these controls.

### Router

```
Router.configure({
  layoutTemplate: 'appLayout',
});
Router.route('checklistPage', {
    path: '/lists/:_id',
    onBeforeAction: function() {
      Session.set('selectedListId', this.params._id);
      this.next();
    },
    yieldTemplates: {
      'navbarFooter': {
        to: 'footer'
      }
    }
  });
```

### Create a Navbar Template

```
<template name="navbarFooter">
  <nav id="navbarFooterNav" class="navbar navbar-default navbar-fixed-bottom"
role="navigation">
    <ul class="nav navbar-nav">
      <li><a id="addPostLink"><u>A</u>dd Post</a></li>
      <li><a id="editPostLink"><u>E</u>dit Post</a></li>
      <li><a id="deletePostLink"><u>D</u>elete Post</a></li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li><a id="helpLink"><u>H</u>elp</a></li>
    </ul>
  </nav>
</template>
```

### Define Yields in the Layout

```
<template name="appLayout">
  <div id="appLayout">
    <header id="navbarHeader">
      {{> yield 'header'}}
    </header>

      <div id="mainPanel">
        {{> yield}}
      </div>

    <footer id="navbarFooter" class="{{getTheme}}"">
      {{> yield 'footerActionElements' }}
    </footer>
  </div>
</template>
```

## Modals

This follwing is a pure-Blaze approach to toggling UI elements into and outof existence. Think of this as a replacement for modal dialogs. In fact, there are a number of ways to implement modal dialogs using this method (simply add background masks and animations).

### Document Object Model

```
<template name="topicsPage">
  <div id="topicsPage" class="container">
    <div class="panel">
      <div class="panel-heading">
        Nifty Panel
      </div>
      <!-- .... -->
      <div class="panel-footer">
        <!-- step 1.  we click on the button object -->
        <div id="createTopicButton" class="btn {{ getPreferredButtonTheme }}">Create
Topic</div>
      </div>
    </div>

    <!-- step 5 - the handlebars gets activated by the javascript controller -->
    <!-- and toggle the creation of new objects in our model -->
    {{#if creatingNewTopic }}
    <div>
      <label for="topicTextInput"></label>
      <input id="topicTextInput" value="enter some text..."></input>
      <button class="btn btn-warning">Cancel</button>
      <button class="btn btn-success">OK</button>
    </div>
    {{/if}}
  </div>
</template>
```

### Javascript

```
// step 2 - the button object triggers an event in the controller
// which toggles our reactive session variable
Template.topicsPage.events({
  'click #createTopicButton':function(){
    if(Session.get('is_creating_new topic'){
      Session.set('is_creating_new_topic', false);
    }else{
      Session.set('is_creating_new_topic', true);
    }
  }
});

// step 0 - the reactive session variable is set false
Session.setDefault('is_creating_new_topic', false);

// step 4 - the reactive session variable invalidates
// causing the creatNewTopic function to be rerun
Template.topicsPage.creatingNewTopic = function(){
  if(Session.get('is_creating_new_topic')){
    return true;
  }else{
    return false;
  }
}
```

## Tagging

**The Database Layer** First, we want to set up the Data Distribution Protocol, to make sure that we can persist data to the database, and get it to the client. Three files need to be created... one on the server, one on the client, and one shared between both.

```
// client/subscriptions.js
Meteor.subscribe('posts');

//lib/model.js
Posts  = new Meteor.Collection("posts");
Posts.allow({
    insert: function(){
        return true;
    },
    update: function () {
        return true;
    },
    remove: function(){
        return true;
    }
});


// server.publications.js
Meteor.publish('posts', function () {
  return Posts.find();
});
```

This example assumes the following document schema for the tagging pattern:

```
{
  _id: "3xHCsDexdPHN6vt7P",
  title: "Sample Title",
  text: "Lorem ipsum, solar et...",
  tags: ["foo", "bar", "zkrk", "squee"]
}
```

**Document Object Model**

Second, we want to create our object model in the application layer. The following is how you would use a Bootstrap panel to render a post with title, text, and tags. Note that `selectedPost`, `tagObjects`, and `tag` are all helper functions of the blogPost template. `title` and `text` are fields from our document record.

```
<template name="blogPost">
  {{#with selectedPost }}
    <div class="blogPost panel panel-default">
      <div class="panel-heading">
        {{ title }}
      </div>
        {{ text }}
      <div class="panel-footer">
        <ul class="horizontal-tags">
          {{#each tagObjects }}
          <li class="tag removable_tag">
            <div class="name">{{tag}}<i class="fa fa-times"></i></div>
          </li>
          {{/each}}
          <li class="tag edittag">
            <input type="text" id="edittag-input" value="" /><i class="fa fa-plus"></i>
          </li>
        </ul>
      </div>
    </div>
  {{/with}}
</template>
```

**Javascript**

Next, we want to set up some controllers to return data, implement some data input, and so forth.

```
// you will need to set the selectedPostId session variable
// somewhere else in your application
Template.blogPost.selectedPost = function(){
  return Posts.findOne({_id: Session.get('selectedPostId')});
}

// next, we use the _.map() function to read the array from our record
// and convert it into an array of objects that Handlebars/Spacebars can parse
Template.blogPost.tagObjects = function () {
    var post_id = this._id;
    return _.map(this.tags || [], function (tag) {
        return {post_id: post_id, tag: tag};
    });
};

// then we wire up click events
Template.blogPost.events({
    'click .fa-plus': function (evt, tmpl) {
        Posts.update(this._id, {$addToSet: {tags: value}});
    },
    'click .fa-times': function (evt) {
        Posts.update({_id: this._id}, {$pull: {tags: this.tag}});
    }
});
```

**Styling**
Lastly, we want to define some different Views for phone, tablet, and desktops; and some basic UI styling depending on user input. This example uses the Less precompiler, although the syntax should be roughly the same for Sass and Stylus.

```
// default desktop view
.fa-plus:hover{
  cursor: pointer;
}
.fa-times:hover{
  cursor: pointer;
}
// landscape orientation view for tablets
@media only screen and (min-width: 768px) {
  .blogPost{
     padding: 20px;
  }
}
// portrait orientation view for tablets
@media only screen and (max-width: 768px) {
  .blogPost{
     padding: 0px;
       border: 0px;
  }
}
// phone view
@media only screen and (max-width: 480px) {
  blogPost{
   .panel-footer{
       display: none;
    }
  }
}
```

## Alerts and Errors

Alerts and errors are nearly the simplest of all Meteor component patterns. They're so simple, in fact, that they barely register as a pattern in of themselves. Instead of adding FlashAlert modules or patterns, all you really need to do is style a Handlebar template appropriate, add a helper, and wire it up to a reactive Session variable.

**Prerequisites**
The following code requires the LESS precompiler and Bootstrap-3, respectively. You will need to run the following commands at the command prompt to get them to work.

```
meteor add less
meteor add ian:bootstrap-3
```

**Document Object Model: Define Alert Object** Start by adding some elements to your document object model. In this case, we want to create a div element for our alert, that's wired up to two Handlebar helpers.

```
<template name="postsPage">
  <div id="postsPage" class="page">
    <div id="postsPageAlert" class="{{alertColor}}">{{alertMessage}}</div>
    <div class="postsList">
      <!-- other code you can ignore in this example -->
    </div>
    <div id="triggerAlertButton" class="btn btn-default">
  </div>
</template>
```

**Javascript: Define Template Helpers** Then we want to wire up some controllers that will populate the object model with data. We do so with two reactive session variables, and two handlebar helpers.

```
Session.setDefault('alertLevel', false);
Session.setDefault('alertMessage', "");

Template.postsPage.alertColor = function(){
 if(Session.get('alertLevel') == "Success"){
  return "alert alert-success";
 }else if(Session.get('alertLevel') == "Info"){
  return "alert alert-info";
 }else if(Session.get('alertLevel') == "Warning"){
  return "alert alert-warning";
 }else if(Session.get('alertLevel') == "Danger"){
  return "alert alert-danger";
 }else{
  return "alert alert-hidden"
 }
}

Template.postsPage.alertMessage = function(){
  return Session.get('alertMessage');
}
```

**Styling: Define DOM Visibility** Then we want to go back to our CSS, and define two views of the postsPage element. In the first View, we display all of the contents in our object model. In the second view, only some of the contents of our object model are displayed.

```
#postsPage{
  .alert{
    display: block;
  }
  .alert-hidden{
    display: none;
  }
}
```

**Javascript: Triggering the Alert**
Lastly, we go back to our controllers, and we define an event controller, which will trigger our alert when clicked.

```
Template.postsPage.events({
  'click #triggerAlertButton':function(){
    Session.set('alertLevel', 'Success');
    Session.set('alertMessage', 'You successfully read this important alert message.');
  }
});
```

And that's all there is to it! Super simple, right? You can now set the `alertLevel` and `alertMessage` session variables anywhere in your codebase, and your application will reactively show alerts and error messages! :)

## Tabbed Workflow

**Document Object Model**
Start by creating your tabs and panes in your Object Model...

```
<template name="samplePage">
  <div id="samplePage" class="page">
    <ul class="nav nav-tabs">
      <li id="firstPanelTab"><a href="#firstPanel">First</a></li>
      <li id="secondPanelTab"><a href="#secondPanel">Second</a></li>
    </ul>

    <div id="firstPanel" class="{{firstPanelVisibility}}">
      {{> firstPanel }}
    </div>
    <div id="secondPanel" class="{{secondPanelVisibility}}">
      {{> secondPanel }}
    </div>
  </div>
</template>
```

**Javascript**

```
// this variable controls which tab is displayed and associated application state
Session.setDefault('selectedPanel', 1);

Template.name.helpers({
  firstPanelVisibility: function (){
    if(Session.get('selectedPanel') === 1){
      return "visible";
    }else{
      return "hidden";
    }
  },
  secondPanelVisibility: function (){
    if(Session.get('selectedPanel') === 2){
      return "visible";
    }else{
      return "hidden";
    }
  },
  thirdPanelVisibility: function (){
    if(Session.get('selectedPanel') === 3){
      return "visible";
    }else{
      return "hidden";
    }
  },
  firstPanelActive: function (){
    if(Session.get('selectedPanel') === 1){
      return "active panel-tab";
    }else{
      return "panel-tab";
    }
  },
  secondPanelActive: function (){
    if(Session.get('selectedPanel') === 2){
      return "active panel-tab";
    }else{
      return "panel-tab";
    }
  },
  thirdPanelActive: function (){
    if(Session.get('selectedPanel') === 3){
      return "active panel-tab";
    }else{
      return "panel-tab";
    }
  }
});
```

**Styling**

```
.visible {
  display: block;
  visibility: visible;
}
.hidden {
  display: none;
  visibility: hidden;
}
```

**Active Tab** For added effect, you can extend this pattern by injecting classes to indicate the active tab.

```
<li id="firstPanelTab" class="{{firstPanelActive}}"><a href="#firstPanel">First</a></li>
<li id="secondPanelTab" class="{{secondPanelActive}}"><a href="#secondPanel">Second</a></li>
```

```
Template.firstPanel.helpers({
  firstPanelActive: function (){
    if(Session.get('selectedPanel') === 1){
      return "active";
    }else{
      return "";
    }
  },
  secondPanelActive: function (){
    if(Session.get('selectedPanel') === 2){
      return "active";
    }else{
      return "";
    }
  },
});
```

Read Blaze User Interface Recipes (Bootstrap; No jQuery) online:
https://riptutorial.com/meteor/topic/4202/blaze-user-interface-recipes--bootstrap--no-jquery-

# Chapter 10: Continuous Deployment to Galaxy from Codeship

## Remarks

This topic is heavily inspired by Nate Strausers Migrating Meteor Apps from Modulus to Galaxy with Continuous Deployment from Codeship.

## Examples

### Setup

- Create a `deployment_token.json`:

  ```
  METEOR_SESSION_FILE=deployment_token.json meteor login
  ```

- Create the following environment variables on Codeship: ( https://codeship.com/projects/PROJECT_NUMBER/configure_environment)
  - METEOR_TARGET: your.domain.com
  - METEOR_TOKEN: Copy/Paste the contents of deployment_token.json. Something like: `{"sessions": {"www.meteor.com": {"session": "12345 ...`
  - METEOR_SETTING: Copy/Paste the contents of your settings.json. Something like: `{"private": {...`

- Create a new deployment pipeline here https://codeship.com/projects/YOUR_PROJECT_NUMBER/deployment_branches/new
  - We deploy only the master branch. So set: Branch is exactly: master.

- Add a "Custom Script" as your deployment with the following content:

```
echo $METEOR_TOKEN > deployment_token.json
echo $METEOR_SETTINGS > deployment_settings.json
meteor npm prune --production
DEPLOY_HOSTNAME=galaxy.meteor.com METEOR_SESSION_FILE=deployment_token.json meteor deploy
$METEOR_TARGET --settings deployment_settings.json
```

Read Continuous Deployment to Galaxy from Codeship online: https://riptutorial.com/meteor/topic/6743/continuous-deployment-to-galaxy-from-codeship

# Chapter 11: Continuous Integration & Device Clouds (with Nightwatch)

## Remarks

Nightwatch has been providing Acceptance and End-to-End testing for Meteor apps since v0.5 days, and has managed migrations from PHP to Spark to Blaze and to React; and all major Continuous Integration platforms. For additional help, please see:

Nightwatch API Documentation
Nightwatch.js Google Group

## Examples

### Travis

Travis is the original Continuous Integration service that became popular in the Meteor community. It's solid and reliable, has long had a open-source hosting tier, and has run hundreds of thousands of Nightwatch tests over the years.

**.travis.yml**
Simply put a `.travis.yml` file in the root of your application, like so:

```
# this travis.yml file is for the leaderboard-nightwatch example, when run standalone
language: node_js

node_js:
  - "0.10.38"

services:
  - mongodb

sudo: required

env:
  global:
    - TRAVIS=true
    - CONFIG_PREFIX=`npm config get prefix`
    - DISPLAY=:99.0
    - NODE_ENV=`travis`
  matrix:

cache:
  directories:
    - .meteor/local/build/programs/server/assets/packages
    - .meteor

before_install:
  # set up the node_modules dir, so we know where it is
  - "mkdir -p node_modules &"
  # install nightwatch, selenium, , so we can launch nightwatch and selenium
  - "meteor npm install nightwatch selenium-server-standalone-jar chromedriver"

  # fire up xvfb on port :99.0
  - "sh -e /etc/init.d/xvfb start"

  # set the xvfb screen size to 1280x1024x16
  - "/sbin/start-stop-daemon --start --quiet --pidfile /tmp/custom_xvfb_99.pid --make-pidfile
--background --exec /usr/bin/Xvfb -- :99 -ac -screen 0 1280x1024x16"

  # install meteor
  - "curl https://install.meteor.com | /bin/sh"

  # give meteor a few seconds after installing
  - "sleep 10"

  # setup Meteor app
  - "cd webapp"
  - "meteor &"

  # give Meteor some time to download packages, init data, and to start
  - "sleep 60"

# then run nightwatch using the chromedriver
script: "nightwatch -c .meteor/nightwatch.json"
```

## Circle

Circle is the newer Continuous Integration service that's become popular among Meteorites. It's got all of the latest bells and whistles, as far as continuous integration goes. The following script supports many new features, including:

- screenshots
- artifacts
- git submodules
- environment detection
- directory caching
- parallelism optimization
- npm scripts
- continuous deployment
- webhooks

### .circle.yml

```
## Customize the test machine
machine:

  # Timezone
  timezone:
    America/Los_Angeles # Set the timezone

  # Add some environment variables
  environment:
    CIRCLE_ENV: test
    CXX: g++-4.8
    DISPLAY: :99.0
    NPM_PREFIX: /home/ubuntu/nvm/v0.10.33
    INITIALIZE: true
    NODE_ENV: circle


## Customize checkout
checkout:
 post:
   #- git submodule sync
   #- git submodule update --init --recursive # use submodules

general:
  build_dir: webapp
  artifacts:
    - "./tests/nightwatch/screenshots" # relative to the build directory

## Customize dependencies
dependencies:
  cache_directories:
    - "~/.meteor" # relative to the user's home directory
    - ~/nvm/v0.10.33/lib/node_modules/starrynight
    - ~/nvm/v0.10.33/bin/starrynight

  pre:
    # Install Starrynight unless it is cached
    - if [ ! -e ~/nvm/v0.10.33/bin/starrynight ]; then npm install -g starrynight; else echo
"Starrynight seems to be cached"; fi;
    # Install  Meteor
    - mkdir -p ${HOME}/.meteor
    # If Meteor is already cached, do not need to build it again.
    - if [ ! -e ${HOME}/.meteor/meteor ]; then curl https://install.meteor.com | /bin/sh; else
echo "Meteor seems to be cached"; fi;
    # Link the meteor executable into /usr/bin
    - sudo ln -s $HOME/.meteor/meteor /usr/bin/meteor
    # Check if the helloworld directory already exists, if it doesn't, create the helloworld
app
    # The following doesn't work, because it should be checking ${HOME}/active-
entry/helloworld
    # - if [ ! -e ${HOME}/helloworld ]; then meteor create --release METEOR@1.1.0.3
helloworld; else echo "helloworld app seems to be cached"; fi;

  override:
    #- meteor list

## Customize test commands
test:
  pre:
    #- starrynight fetch
    #- cd packages && rm -rf temp
    #- cd packages && ls -la
    #- starrynight autoconfig
    - meteor update --release METEOR@1.3.3
    - meteor npm install --save jquery bootstrap react react-dom react-router react-bootstrap
react-komposer
    - cat .meteor/nightwatch.json
    - meteor:
          background: true
    - sleep 60
  override:
    - meteor npm run-script nightwatch


## Customize deployment commands
#deployment:
#  production:
#    branch: master
#    commands:
#      - printf "<Meteor username>\n<Meteor password>\n" | meteor deploy myapp.meteor.com

## Custom notifications
#notify:
  #webhooks:
    # A list of hashes representing hooks. Only the url field is supported.
    #- url: https://someurl.com/hooks/circle
```

## SauceLabs

SauceLabs is an Automated Testing Platform for the enterprise. It supports both continuous integration, cross browser testing, and a mobile device cloud. Costs are higher than with Travis, Circle, or BrowserStack, hwoever.

```
{
  "selenium" : {
    "start_process" : false,
    "host" : "ondemand.saucelabs.com",
    "port" : 80,
  },
  "test_settings" : {
    "chrome_saucelabs": {
      "selenium_host": "ondemand.saucelabs.com",
      "selenium_port": 80,
      "username": "${SAUCE_USERNAME}",
      "access_key": "${SAUCE_ACCESS_KEY}",
      "use_ssl": false,
      "silent": true,
      "output": true,
      "screenshots": {
        "enabled": false,
        "on_failure": true,
        "path": ""
      },
      "desiredCapabilities": {
        "name": "test-example",
        "browserName": "chrome"
      },
      "globals": {
        "myGlobal": "some_sauce_global"
      }
    },
  }
}
```

## BrowserStack

BrowserStack uses a device cloud for cross-browser testing. The intent is to allow testing of

Selenium scripts on every device possible.

```
{
  "selenium" : {
    "start_process" : false,
    "host" : "hub.browserstack.com",
    "port" : 80,
  },

  "test_settings" : {
    "default" : {
      "launch_url" : "http://hub.browserstack.com",
      "selenium_port"  : 80,
      "selenium_host"  : "hub.browserstack.com",
      "silent": true,
      "screenshots" : {
        "enabled" : false,
        "path" : "",
      },
      "desiredCapabilities": {
        "browserName": "firefox",
        "javascriptEnabled": true,
        "acceptSslCerts": true,
        "browserstack.user": "USERNAME",
        "browserstack.key": "KEY"
      }
    }
  }
}
```

Read Continuous Integration & Device Clouds (with Nightwatch) online:
https://riptutorial.com/meteor/topic/6550/continuous-integration---device-clouds--with-nightwatch-

# Chapter 12: Debugging

## Examples

### Browser Debuggers

Both Chrome and Safari have built in debuggers. With Chrome, all you have to do is right-click on a web page and 'Inspect Element'. With Safari, you'll have to go into Preferences > Advanced and click on 'Show Develop menu in menu bar'.

With Firefox, you'll need to install Firebug

### Add Debugger Breakpoints to your App

You'll need to add `debugger` statements to your code:

```
Meteor.methods({
  doSomethingUself: function(){
    debugger;
    niftyFunction();
  }
});
```

### Server Side Debugging with Node Inspector

For server side debugging, you'll need to use a tool like Node Inspector. Before you get started, check out some of these useful tutorials.

HowToNode - Debugging with Node Inspector
Strongloop - Debugging Applications
Easily Debugging Meteor.js Walkthrough with Screenshots of Using Node Inspector with Meteor

tl;dr - there are a number of utilities in the Meteor ecosystem which are designed to be run at the same time as your Meteor application. They only work if your Meteor app is up and running and they can connect to a running website. meteor mongo, Robomongo, Nightwatch... these are all utilities that need your application to already be running. NodeInspector is one of these utilities.

```
# install node-inspector
terminal-a$  npm install -g node-inspector

# start meteor
terminal-a$  NODE_OPTIONS='--debug-brk --debug' mrt run

# alternatively, some people report this syntax being better
terminal-a$  sudo NODE_OPTIONS='--debug' ROOT_URL=http://myapp.com meteor --port 80

# launch node-inspector along side your running app
terminal-b$  node-inspector
```

```
# go to the URL given by node-inspector
http://localhost:8080/debug?port=5858
```

## Server Side Debugging with npm debug

Besides Node Inspector, some people have reported success with a npm utility called `debug`.

MeteorHacks - Debugging Meteor with npm debug

## Meteor Shell

As of Meteor 1.0.2, there's a new command shell which you can use to do interactive debugging and manage your app from the server side, just like you do with the Chrome Console on the client side! Check it out:

```
meteor shell
```

## Other Debugging Utilities

Meteor Dump
Meteor Toys
Constellation

Meteor DevTools

Read Debugging online: https://riptutorial.com/meteor/topic/3378/debugging

# Chapter 13: Deployment with Upstart

## Examples

### Upstart Service

This deployment guide assumes you're using an Ubuntu server, and are either self-hosting or using an Infrastructure as a Service (IaaS) provider, such as Amazon Web Services or Rackspace. Your Ubuntu server needs to be running a daemon for launching other apps, for which we recommend the Upstart service. You can find more about Upstart with the following links:

Upstart - Getting Started
Getting Started with Upstart Scripts on Ubuntu
UbuntuBootupHowTo
Upstart Intro, Cookbook, and Best Practices
Run NodeJS As a Service on Ubuntu Karmic

### Copying Files To Your Server Then Build

One favored approach to deploying to a server is to use Git or GitHub. This basically involves logging into your server, moving to the directory you want to run your app from, then cloning your files directly from GitHub. You then build your app on the server. This approach ensures that platform specific files get built correctly, but requires that Meteor is installed on the server (500+ MB), and can result in slightly different builds wind up in production if your servers are slightly different.

```
cd /var/www
sudo git clone http://github.com/myaccount/myapp.git
cd /var/www/myapp
meteor build --directory ../myapp-production
sudo service myapp restart
```

### Bundle Then Copy To Server

Alternatively, you may want to build your application, and then deploy it..

```
cd myapp
meteor build --directory ../output
cd ..
scp output -r username@destination_host:/var/www/myapp-production
```

### Writing Your Upstart Script

You'll need an upstart script in your `/etc/init/ directory`. Name it with your app's name, ending in `.conf`, such as `/etc/init/myapp.conf`. The basic upstart script looks something like this:

```
## /etc/init/myapp.conf
description "myapp.mydomain.com"
author      "somebody@gmail.com"

# Automatically Run on Startup
start on started mountall
stop on shutdown

# Automatically Respawn:
respawn
respawn limit 99 5

script
    export HOME="/root"
    export MONGO_URL='mongodb://myapp.compose.io:27017/meteor'
    export ROOT_URL='http://myapp.mydomain.com'
    export PORT='80'

    exec /usr/local/bin/node /var/www/myapp/main.js >> /var/log/myapp.log 2>&1
end script
```

## Upstart Script For Replica Sets

If you're running a replica set or have a need to shard your database, you'll want an upstart script that looks something like this:

```
# /etc/init/myapp.conf
description "myapp.mydomain.com"
author      "somebody@gmail.com"

# used to be: start on startup
# until we found some mounts weren't ready yet while booting:
start on started mountall
stop on shutdown

# Automatically Respawn:
respawn
respawn limit 99 5

script
    # upstart likes the $HOME variable to be specified
    export HOME="/root"

    # our example assumes you're using a replica set and/or oplog integreation
    export MONGO_URL='mongodb://mongo-a,mongo-b,mongo-c:27017/?replicaSet=meteor'

    # root_url and port are the other two important environment variables to set
    export ROOT_URL='http://myapp.mydomain.com'
    export PORT='80'

    exec /usr/local/bin/node /var/www/production/main.js >> /var/log/node.log 2>&1
end script
```

## Running Your Upstart Script

Finally, you'll need to start the Upstart daemon, and initialize your app as a service.

```
sudo service myapp start
```

## Setting up a Server to Host Multiple Meteor Apps

https://www.phusionpassenger.com/
https://github.com/phusion/passenger
https://github.com/phusion/passenger/wiki/Phusion-Passenger:-Meteor-tutorial#wiki-installing

Read Deployment with Upstart online: https://riptutorial.com/meteor/topic/3377/deployment-with-upstart

# Chapter 14: Development Tools

## Examples

### Integrated Development Environments

Development usually begins with an editor or an Integrated Development Environment. The following IDEs are known to support Meteor to some extent:

- Atom - Javascript IDE that can fully leverage Meteor's isomorphic javascript framework. If you want to be able to hack on your editor itself, this is the one to choose.
- Cloud9 - The newest Cloud Development offering that supports Meteor, with a tutorial.
- MeteorDevTools - Chrome extension for Blaze, DDP, and Minimongo.
- Sublime - Light-weight and popular text editor.
- WebStorm - The most full featured IDE currently available for Meteor.

### Database Tools

Once you get past your 'Hello World' app, you'll need to start paying attention to your collection and document schemas, and will need some tools for managing your database.

- Robomongo - A longtime community favorite for managing Mongo. Highly recommended.
- JSON Generator - Invaluable utility for generating sample datasets.
- MacOSX Mongo Preference Page - Preferences GUI for MacOSX.
- MongoHub - Another Mongo GUI, similar to RoboMongo. MacOSX only.
- Mongo3 - One of the few cluster management tools around. Able to visualize replication sets. Only downside is it's built in Ruby.
- Mongo Monitoring Service - Once you're ready to bring something into production, MMS is invaluable. Now known as MongoDB Atlas.
- Mongo Express - Web-based MongoDB admin interface, written with Node.js and express

### Remote Collaboration Utilities for Distributed Developers

Developing Meteor apps usually means developing multi-client reactivity, which requires collaboration tools. The following tools have proven to be popular within the Meteor community.

- Google Hangouts - Video conferencing and chat.
- Zenhub.io - Kanban boards for GitHub.
- InVision - Collaborative wireframing and prototyping.
- Meeting Hero - Collaborative meeting planning.
- Hackpad - Collaborative document editing.
- Slack - Collaborative project tracking feeds.
- MadEye - Collaborative web editor.
- Screenhero - Collaborative screen sharing.
- Proto.io - Wireframing and prototyping.

- HuBoard - Kanban boards for GitHub.
- Zapier - The best apps. Together.
- Teamwork.com - Traditional project management & gannt charts.
- Sprint.ly - More kanban boards and sprint planning that works with GitHub.
- LucidChart - Online Visio alternative.
- Waffle.io - Trello/ZenHub alternative that integrates with GitHub.

### REST Clients

If you want to integrate Meteor with an external API, it's likely that it's going to exposed as a REST interface. We tend to use the following Chrome apps for testing REST APIs.

- Postman
- DHC Rest Client

Online tools:

- Hurl.it
- RequestBin

### Debuggers

Most debugging happens in the terminal or in the Chrome or Safari develop tools, which are plenty sophisticated enough for 99% of your needs. However, if you want to debug on Firefox or need extra server debugging functionality, there are a few extra utilities you can use.

- Firefox - Firebug
- Node-Inspector
- Meteor Toys or add directly- `meteor add meteortoys:allthings`

### Mobile Coding on iOS

Texttastic Code Editor - Code editor with syntax highlighting for iOS devices.

Working Copy - Clone Github repositories to your iPad and code on the go.

CodeHub - Browse and maintain your GitHub repositories. Management tool.

iOctocat - Social utility for following Github projects.

iMockups for iPad - Wireframes and mockups. Supports wireframes for desktops and mobile.

Blueprint - iOS wireframing and mockups. Primarily for iOS development, but somewhat usable for web apps.

JSON Designer - Data architecture and data schema design.

Read Development Tools online: https://riptutorial.com/meteor/topic/4200/development-tools

# Chapter 15: Directory Structure

## Introduction

Before the release of Meteor 1.3, Meteor developers were frustrated with Meteor.js' handling of file dependencies and global variables. In response, Meteor set new standards for project structures in order to make the project dependency system more streamlined. This topic explains the standardized project structure and the principles behind it.

## Remarks

**client**
All code in the client directory is run only in the client-side, or web browser.

**client/compatibility**
The compatibility directory contains legacy or 3rd party code, such as jQuery libraries, etc.

**lib**
The lib directory is loaded before other directories in your Meteor project, and is loaded on both the server and client. This is the preferred place to define data models, isomorphic libraries, and business logic.

**imports**
The imports directory is a directory on the server that is available to both the server and client, but only before the client bundle gets shipped to the client.

**packages**
The packages directory is where custom packages are stored during local development. When using the standard command `meteor add package:name` to add a package, Meteor will look first in this directory if a local package has the corresponding description name in its `package.js` file. If not, it will poll Atmosphere as usual.

**private**
The private directory contains static files that should only be available on the web server.

**public**
The public directory contains static files that are only available on the application client. This may including branding assets, etc.

**server**
The server directory contains server-side assets. This can include authentication logic, methods, and other code that may need security consideration.

**tests**
The tests directory is omitted by default when your application is bundled and deployed.

As suggested by Richard Silverton it is a convenient idea to put not only the meteor project directory under version control, but also its parent directory.

That way you can keep files under version control without having meteor to deal with it.

## Examples

### Classic Directory Structures

The first thing you need to know when structuring your apps is that the Meteor tool has some directories that are hard-coded with specific logic. At a very basic level, the following directories are "baked in" the Meteor bundler.

```
client/                                 # client application code
client/compatibility/                   # legacy 3rd party javascript libraries
imports/                                # for lazy loading feature
lib/                                    # any common code for client/server.
packages/                               # place for all your atmosphere packages
private/                                # static files that only the server knows about
public/                                 # static files that are available to the client
server/                                 # server code
tests/                                  # unit test files (won't be loaded on client or
server)
```

Reference page: Meteor Guide > Special directories

### Package-Only Directory Structure

Many people find themselves eventually supporting multiple applications, and desire to share code between apps. This leads to the concept of microservice architecture, and all-package apps. Essentially, the code from the entire classic directory structure is refactored out into packages.

Even though there is no hard-coded logic for directories in packages, we find that it's a good practice to use the classic directory structure when creating packages. This creates a natural refactor path as features are prototyped in the app, and then extracted into packages to be published and shared. The directory names are shared, so there's less confusion among team members.

```
client/                                 # client application code
packages/                               # place for all your atmosphere packages
packages/foo/client                     # client application code
packages/foo/lib                        # any common code for client/server
packages/foo/server                     # server code
packages/foo/tests                      # tests
server/                                 # server code
```

### Imports/Modules Directory Structure

The most recent versions of Meteor ship with support for `ecmascript`, aka ES6 or ES2015. Instead of packages, Javascript now supports `import` statements and modules, which replaces the need for package-only applications. The latest directory structure is similar to the package-only structure, but uses the `/imports` directory instead of `/packages`.

```
imports                                   #
imports/api                               # isomorphic methods
imports/lib                               # any common code for client/server
imports/client                            # client application code
imports/server                            # server code
```

### Mixed-Mode Directory Structure

And, of course, you can mix these approaches, and use both packages and imports along side your application specific code. A mix-mode structure is most common in three situations: a franken-app, which is just sort of pulling a bit from here-and-there without any overall strategy; an app that's being actively refactored from either Classic or Package-Only structures to the Imports/Modules structure.

```
client/                                   # client application code
client/compatibility/                     # legacy 3rd party javascript libraries
imports                                   #
imports/api                               # isomorphic methods
imports/lib                               # any common code for client/server
imports/client                            # client application code
imports/server                            # server code
lib/                                      # any common code for client/server.
packages/                                 # place for all your atmosphere packages
packages/foo/client                       # client application code
packages/foo/lib                          # any common code for client/server
packages/foo/server                       # server code
packages/foo/tests                        # tests
private/                                  # static files that only the server knows about
public/                                   # static files that are available to the client
server/                                   # server code
tests/                                    # unit test files (won't be loaded on client or
server)
```

### Directory load order

HTML template files are always loaded before everything else

Files beginning with *main.* are loaded last

Files inside any lib/ directory are loaded next

Files with deeper paths are loaded next

Files are then loaded in alphabetical order of the entire path

Reference Link

Reference page: Meteor Guide > Application Structure > Default file load order

Read Directory Structure online: https://riptutorial.com/meteor/topic/3072/directory-structure

# Chapter 16: Electrify - Compiling Meteor as a Locally Installable App

## Examples

### Installing Electrify for a Meteor application

Electron ports HTML web applications to native applications for a range of devices, including creating native desktop applications. It's also very easy to get started!

To begin, we must have `electron`, `nodejs`, `npm`, `git` and `meteor` installed. Familiarity with these tools is vital for working with Meteor, so make sure you know about these things first.

---

#### Electron

```
npm install -g electrify
```

- `electron` is what we're using! Read more here.
- `electrify` is a tool for packaging Meteor apps. Read mode here.

---

#### Other requirements for installing and using Electrify with Meteor

#### Meteor

```
curl https://install.meteor.com/ | sh
```

There are many ways to install Meteor, see here.

- `meteor` is the JavaScript framework we'll be using for building our application. It provides us with a lot of coding simplifications for some rather conceptually hard problems in web applications; its simplicity has been noted as useful for prototypical projects. Read more here .

#### NodeJS

```
apt-get install nodejs build-essentials
```

There are many ways to install, depending on your OS. Find out which way you need here.

- `nodejs` is the package for Node.js, which is a Javascript environment for running JavaScript on the server side. Read more here.

#### npm

`npm` should be bundled with the `nodejs` installation. Check it is by running the command `npm -v` after installing `nodejs`.

- `npm` is the Node Package Manager. It's a huge collection of open source modules that you can easily add into your Node projects. Read more here.

## Using Electrify on a Meteor Application

Let's download a Meteor Todos example project, using a Linux shell (command line) script, to test out Electrifying a project for the first time:

---

**Requirements for this section:**

**Git**

```
apt-get install git-all
```

There are many ways to install Git. Check them out here.

- `git` is a version control system for files. They can be stored remotely (i.e., online) in public repositories (GitHub being a rather famous one) or private repositories (BitBucket provides limited free private repositories, as an example). Read more [here][5].

---

```
#!/usr/bin/bash

# Change this parameter to choose where to clone the repository to.
TODOSPATH="/home/user/development/meteor-todos"

# Download the repository to the $TODOSPATH location.
git clone https://github.com/meteor/todos.git "$TODOSPATH"

# Change directory (`cd`) into the Todos project folder.
cd "$TODOSPATH"
```

We should now have a project folder named 'meteor-todos', at the location specified within the TODOSPATH parameter. We've also changed directory (`cd`) into the project folder, so let's add Electrify to this project!

```
# It's really this simple.
electrify
```

That's right - a single word command, and our project is ready. Permissions may cause errors for you when trying to run `electrify` as a command, in wihch case try `sudo electrify` to override the permissions.

However, do attempt to resolve these permission issues - it is not good practice to unnecessarily `sudo` (which I'd elaborate upon, but I could write a whole other topic on why that is!)

Read Electrify - Compiling Meteor as a Locally Installable App online:

https://riptutorial.com/meteor/topic/2526/electrify---compiling-meteor-as-a-locally-installable-app

# Chapter 17: Environment Detection

## Examples

### Advanced Environment Configurations

For more complex applications, you'll want to build up a ``settings.json` object using multiple environment variables.

```
if(Meteor.isServer){
  Meteor.startup(function()){
    // this needs to be run on the server
    var environment, settings;

    environment = process.env.METEOR_ENV || "development";

    settings = {
      development: {
        public: {
          package: {
            name: "jquery-datatables",
            description: "Sort, page, and filter millions of records. Reactively.",
            owner: "LumaPictures",
            repo: "meteor-jquery-datatables"
          }
        },
        private: {}
      },
      staging: {
        public: {},
        private: {}
      },
      production: {
        public: {},
        private: {}
      }
    };

    if (!process.env.METEOR_SETTINGS) {
      console.log("No METEOR_SETTINGS passed in, using locally defined settings.");
      if (environment === "production") {
        Meteor.settings = settings.production;
      } else if (environment === "staging") {
        Meteor.settings = settings.staging;
      } else {
        Meteor.settings = settings.development;
      }
      console.log("Using [ " + environment + " ] Meteor.settings");
    }
  });
}
```

### Specifying app parameters with METEOR_SETTINGS

The METEOR_SETTINGS environment variable can accept JSON objects, and will expose that object in the `Meteor.settings` object. First, add a `settings.json` to your app root with some configuration info.

```
{
  "public":{
    "ga":{
      "account":"UA-XXXXXXX-1"
    }
  }
}
```

Then you'll need to launch your application using your settings file.

```
# run your app in local development mode with a settings file
meteor --settings settings.json

# or bundle and prepare it as if you're running in production
# and specify a settings file
meteor bundle --directory /path/to/output
cd /path/to/output
MONGO_URL="mongodb://127.0.0.1:27017" PORT=3000 METEOR_SETTINGS=$(cat /path/to/settings.json)
node main.js
```

These settings can then be accessed from Meteor.settings and used in your app.

```
Meteor.startup(function(){
  if(Meteor.isClient){
    console.log('Google Analytics Account', Meteor.settings.public.ga.account);
  }
});
```

## Environment Detection on the Server

Environment variables are also available to the server via the `process.env` object.

```
if (Meteor.isServer) {
  Meteor.startup(function () {
    // detect environment by getting the root url of the application
    console.log(JSON.stringify(process.env.ROOT_URL));

    // or by getting the port
    console.log(JSON.stringify(process.env.PORT));

    // alternatively, we can inspect the entire process environment
    console.log(JSON.stringify(process.env));
  });
}
```

## Client Environment Detection using Meteor Methods

To detect the environment on the server, we have to create a helper method on the server, as the server will determine which environment it is in, and then call the helper method from the client.

Basically, we just relay the environment info from the server to the client.

```
//---------------------------------------------------------------------------------------------
----------
// server/server.js
// we set up a getEnvironment method

Meteor.methods({
  getEnvironment: function(){
    if(process.env.ROOT_URL == "http://localhost:3000"){
        return "development";
    }else{
        return "staging";
    }
  }
 });

//---------------------------------------------------------------------------------------------
----------
// client/main.js
// and then call it from the client

Meteor.call("getEnvironment", function (result) {
  console.log("Your application is running in the " + result + "environment.");
});
```

## Client Environment Detection using NODE_ENV

As of Meteor 1.3, Meteor now exposes the `NODE_ENV` variable on the client by default.

```
if (Meteor.isClient) {
  Meteor.startup(function () {
    if(process.env.NODE_ENV === "testing"){
      console.log("In testing...");
    }
    if(process.env.NODE_ENV === "production"){
      console.log("In production...");
    }
  });
}
```

Read Environment Detection online: https://riptutorial.com/meteor/topic/4198/environment-detection

# Chapter 18: Environment Variables

## Parameters

| Parameter | Details |
|---|---|
| PORT | Port that the Meteor app will be available on. |
| MONGO_URL | URL to connect to the Mongo instance. |
| ROOT_URL | ... |
| OPLOG_URL | ... |
| MONGO_OPLOG_URL | ... |
| METEOR_ENV | ... |
| NODE_ENV | ... |
| NODE_OPTIONS | ... |
| DISABLE_WEBSOCKETS | ... |
| MAIL_URL | ... |
| DDP_DEFAULT_CONNECTION_URL | ... |
| HTTP_PROXY | ... |
| HTTPS_PROXY | ... |
| METEOR_OFFLINE_CATALOG | ... |
| METEOR_PROFILE | ... |
| METEOR_DEBUG_BUILD | ... |
| TINYTEST_FILTER | ... |
| MOBILE_ROOT_URL | ... |
| NODE_DEBUG | ... |
| BIND_IP | ... |
| PACKAGE_DIRS | ... |

| Parameter | Details |
|---|---|
| DEBUG | ... |
| METEOR_PRINT_CONSTRAINT_SOLVER_INPUT | ... |
| METEOR_CATALOG_COMPRESS_RPCS | ... |
| METEOR_MINIFY_LEGACY | ... |
| METEOR_DEBUG_SQL | ... |
| METEOR_WAREHOUSE_DIR | ... |
| AUTOUPDATE_VERSION | ... |
| USE_GLOBAL_ADK | ... |
| METEOR_AVD | ... |
| DEFAULT_AVD_NAME | ... |
| METEOR_BUILD_FARM_URL | ... |
| METEOR_PACKAGE_SERVER_URL | ... |
| METEOR_PACKAGE_STATS_SERVER_URL | ... |
| DEPLOY_HOSTNAME | ... |
| METEOR_SESSION_FILE | ... |
| METEOR_PROGRESS_DEBUG | ... |
| METEOR_PRETTY_OUTPUT | ... |
| APP_ID | ... |
| AUTOUPDATE_VERSION | ... |
| CONSTRAINT_SOLVER_BENCHMARK | ... |
| DDP_DEFAULT_CONNECTION_URL | ... |
| SERVER_WEBSOCKET_COMPRESSION | ... |
| USE_JSESSIONID | ... |
| METEOR_PKG_SPIDERABLE_PHANTOMJS_ARGS | ... |
| WRITE_RUNNER_JS | ... |

| Parameter | Details |
|---|---|
| TINYTEST_FILTER | ... |
| METEOR_PARENT_PID | ... |
| METEOR_TOOL_PATH | ... |
| RUN_ONCE_OUTCOME | ... |
| TREE_HASH_DEBUG | ... |
| METEOR_DEBUG_SPRINGBOARD | ... |
| METEOR_TEST_FAIL_RELEASE_DOWNLOAD | ... |
| METEOR_CATALOG_COMPRESS_RPCS | ... |
| METEOR_TEST_LATEST_RELEASE | ... |
| METEOR_WATCH_POLLING_INTERVAL_MS | ... |
| EMACS | ... |
| METEOR_PACKAGE_STATS_TEST_OUTPUT | ... |
| METEOR_TEST_TMP | ... |

## Examples

### Using Environment Variables with Meteor

Environment variables can be defined before the meteor call, like so:

```
PORT=4000 meteor
NODE_ENV="staging" meteor
```

### Setting Meteor SMTP server

#### Gmail Example

```
MAIL_URL=smtp://username%40gmail.com:password@smtp.gmail.com:465/
```

Note: This setup only allows 2000 emails to be sent per day. Please see https://support.google.com/a/answer/176600?hl=en for alternative configurations.

Read Environment Variables online: https://riptutorial.com/meteor/topic/3154/environment-variables

# Chapter 19: ES2015 modules (Import & Export)

## Remarks

MDN documentation for imports: https://developer.mozilla.org/en/docs/web/javascript/reference/statements/import MDN documentation for exports: https://developer.mozilla.org/en/docs/web/javascript/reference/statements/export ExploringJS chapter on modules: http://exploringjs.com/es6/ch_modules.html

## Examples

### Importing in app modules

Node modules

```
import url from 'url';
import moment from 'moment';
```

Meteor packages

```
import { Meteor } from 'meteor/meteor';
import { SimpleSchema } from 'meteor/aldeed:simple-schema';
```

### Importing in Meteor packages

In package.js:

```
Npm.depends({
  moment: "2.8.3"
});
```

In a package file:

```
import moment from 'moment';
```

### Exporting variables from app modules

```
// Default export
export default {};

// Named export
export const SomeVariable = {};
```

### Exporting symbols from Meteor packages

In your mainModule file:

```
export const SomeVar = {};
```

Read ES2015 modules (Import & Export) online: https://riptutorial.com/meteor/topic/3763/es2015-modules--import---export-

# Chapter 20: ESLint

## Examples

### Adding eslint to your Meteor project

We'll use the popular `eslint-config-airbnb` as a starter as well as Meteor specific rules using `eslint-import-resolver-meteor`.

We also need to install `babel-parser` to lint Meteor enabled ES7 features such as async/await.

```
cd my-project
npm install --save-dev eslint-config-airbnb eslint-plugin-import eslint-plugin-react eslint-plugin-jsx-a11y eslint babel-eslint eslint-import-resolver-meteor
touch .eslintrc.json
```

Then simply use this boilerplate `.eslintrc.json` to get started, you can override the rules as you wish.

```
{
  "parser": "babel-eslint",
  "settings": {
    "import/resolver": "meteor"
  },
  "extends": "airbnb",
  "rules": {}
}
```

### Using an npm script to lint your code

Edit your `package.json` to add the following script :

```
{
  "scripts": {
    "lint": "eslint .;exit 0"
  }
}
```

Then run it using `npm run lint`

We use `exit 0` as a trick to gracefully terminate the script when linting fails, otherwise `npm` will use `eslint` return code and crash.

Read ESLint online: https://riptutorial.com/meteor/topic/3772/eslint

# Chapter 21: File Uploading

## Remarks

The CollectionFS package has been shelved and discontinued by it's author; however, since there's no alternative package in Atmosphere or the Meteor ecosystem for using Mongo's GridFS functionality, and the code still works perfectly fine; we recommend not removing the example from StackOverflow Documentation until some other GridFS solution can be documented as it's replacement.

**Additional Research**
Filepicker.io Uploads and Image Conversion
Dario's Save File Pattern
Micha Roon's File Upload Pattern
EventedMind File Upload Package

## Examples

### Server/Client

Uploading files can be easy or really complicated, depending on what you're wanting to do. In general, transfering a file itself isn't all that difficult. But there are lots of edge cases around attachments, binary files, and the like. And the real sticking point is horizontal scaling, and creating a solution that works when the server is cloned a second, third, and nth time.

Let's start with a basic server/client upload model. We begin by adding a file input element to the document object model.

```
<template name="example">
  <input type=file />
</template>
```

Then attach an event to the input element within your controller, and call a local Meteor method ``startFileTransfer'' to initiate the transfer.

```
// client/example.js
Template.example.events({
  'change input': function(ev) {
    _.each(ev.srcElement.files, function(file) {
      Meteor.startFileTransfer(file, file.name);
    });
  }
});

// client/save.js
/**
 * @blob (https://developer.mozilla.org/en-US/docs/DOM/Blob)
 * @name the file's name
 * @type the file's type: binary, text (https://developer.mozilla.org/en-
US/docs/DOM/FileReader#Methods)
 *
 * TODO Support other encodings: https://developer.mozilla.org/en-
US/docs/DOM/FileReader#Methods
 * ArrayBuffer / DataURL (base64)
 */
Meteor.startFileTransfer = function(blob, name, path, type, callback) {
  var fileReader = new FileReader(),
    method, encoding = 'binary', type = type || 'binary';
  switch (type) {
    case 'text':
      // TODO Is this needed? If we're uploading content from file, yes, but if it's from an
input/textarea I think not...
      method = 'readAsText';
      encoding = 'utf8';
      break;
    case 'binary':
      method = 'readAsBinaryString';
      encoding = 'binary';
      break;
    default:
      method = 'readAsBinaryString';
      encoding = 'binary';
      break;
  }
  fileReader.onload = function(file) {
    Meteor.call('saveFileToDisk', file.srcElement.result, name, path, encoding, callback);
  }
  fileReader[method](blob);
}
```

The client will then call the saveFileToDisk server method, which does the actual transfer and puts everything to disk.

```
//
/**
 * TODO support other encodings:
 * http://stackoverflow.com/questions/7329128/how-to-write-binary-data-to-a-file-using-node-js
 */
Meteor.methods({
  saveFileToDisk: function(blob, name, path, encoding) {
    var path = cleanPath(path), fs = __meteor_bootstrap__.require('fs'),
      name = cleanName(name || 'file'), encoding = encoding || 'binary',
      chroot = Meteor.chroot || 'public';
    // Clean up the path. Remove any initial and final '/' -we prefix them-,
    // any sort of attempt to go to the parent directory '..' and any empty directories in
    // between '/////' - which may happen after removing '..'
    path = chroot + (path ? '/' + path + '/' : '/');

    // TODO Add file existance checks, etc...
    fs.writeFile(path + name, blob, encoding, function(err) {
      if (err) {
        throw (new Meteor.Error(500, 'Failed to save file.', err));
      } else {
        console.log('The file ' + name + ' (' + encoding + ') was saved to ' + path);
      }
    });

    function cleanPath(str) {
      if (str) {
        return str.replace(/\.\./g,'').replace(/\/+/g,'').
          replace(/^\/+/,'').replace(/\/+$/,'');
      }
    }
    function cleanName(str) {
      return str.replace(/\.\./g,'').replace(/\//g,'');
    }
  }
});
```

That's sort of the bare-bones approach, and it leaves a lot to be desired. It's maybe good for uploading a CSV file or something, but that's about it.

## Dropzone (with iron:router)

If we want something a bit more polished, with an integrated Dropzone UI and a REST endpoint, we're going to need to start adding custom REST routes and packages with UI helpers.

Lets begin by importing Iron Router and Dropzone.

```
 meteor add iron:router
 meteor add awatson1978:dropzone
```

And configure the uploads url route that's specified in the dropzone helper.

```
Router.map(function () {
    this.route('uploads', {
      where: 'server',
      action: function () {
        var fs = Npm.require('fs');
        var path = Npm.require('path');
        var self = this;

        ROOT_APP_PATH = fs.realpathSync('.');

        // dropzone.js stores the uploaded file in the /tmp directory, which we access
        fs.readFile(self.request.files.file.path, function (err, data) {

          // and then write the file to the uploads directory
          fs.writeFile(ROOT_APP_PATH + "/assets/app/uploads/" +self.request.files.file.name,
data, 'binary', function (error, result) {
            if(error){
              console.error(error);
            }
            if(result){
              console.log('Success! ', result);
            }
          });
        });
      }
    });
  });
```

Cool! We have a file uploader with snazzy UI and a programmable REST endpoint. Unfortunately, this doesn't scale particularly well.

## Filepicker.io

To scale things, we have to stop using local storage on our server, and start using either a dedicated file storage service or implement a horizontal storage layer. The easiest way to get started with scalable file storage is to use a solution like Filepicker.io, which supports S3, Azure, Rackspace, and Dropbox. loadpicker has been a popular Filerpicker unipackage for awhile.

```
meteor add mrt:filepicker
```

The Filepicker pattern is rather different than the other solutions, because it's really about 3rd party integration. Begin by adding a filepicker input, which you'll see relies heavily on data-* attributes, which is a fairly uncommon pattern in Meteor apps.

```
<input type="filepicker"
  id="filepickerAttachment"
  data-fp-button-class="btn filepickerAttachment"
  data-fp-button-text="Add image"
  data-fp-mimetypes="image/*"
  data-fp-container="modal"
  data-fp-maxsize="5000000"
  data-fp-services="COMPUTER,IMAGE_SEARCH,URL,DROPBOX,GITHUB,GOOGLE_DRIVE,GMAIL">
```

You'll walso want to set an API key, construct the filepicker widget, trigger it, and observe it's outputs.

```
if(Meteor.isClient){
  Meteor.startup(function() {
    filepicker.setKey("YourFilepickerApiKey");
  });
  Template.yourTemplate.rendered = function(){
    filepicker.constructWidget($("#filepickerAttachment"));
  }
  Template.yourTemplate.events({
  'change #filepickerAttachment': function (evt) {
    console.log("Event: ", evt, evt.fpfile, "Generated image url:", evt.fpfile.url);
  });
});
```

## CollectionFS

However, if you're really serious about storage, and you want to store millions of images, you're going to need to leverage Mongo's GridFS infrastructure, and create yourself a storage layer. For that, you're going to need the excellent CollectionFS subsystem.

Start by adding the necessary packages.

```
meteor add cfs:standard-packages
meteor add cfs:filesystem
```

And adding a file upload element to your object model.

```
<template name="yourTemplate">
    <input class="your-upload-class" type="file">
</template>
```

Then add an event controller on the client.

```
Template.yourTemplate.events({
    'change .your-upload-class': function(event, template) {
        FS.Utility.eachFile(event, function(file) {
            var yourFile = new FS.File(file);
            yourFile.creatorId = Meteor.userId(); // add custom data
            YourFileCollection.insert(yourFile, function (err, fileObj) {
                if (!err) {
                   // do callback stuff
                }
            });
        });
    }
});
```

And define your collections on your server:

```
YourFileCollection = new FS.Collection("yourFileCollection", {
    stores: [new FS.Store.FileSystem("yourFileCollection", {path: "~/meteor_uploads"})]
});
YourFileCollection.allow({
    insert: function (userId, doc) {
        return !!userId;
    },
    update: function (userId, doc) {
        return doc.creatorId == userId
    },
    download: function (userId, doc) {
        return doc.creatorId == userId
    }
});
```

Thanks to Raz for this excellent example. You'll want to check out the complete CollectionFS Documentation for more details on what all CollectionFS can do.

## Server Uploads

The following scripts are for uploading a file from the server filesystem into the server. Mostly for config files and filewatchers.

```
//https://forums.meteor.com/t/read-file-from-the-public-folder/4910/5

// Asynchronous Method.
Meteor.startup(function () {
    console.log('starting up');

    var fs = Npm.require('fs');
    // file originally saved as public/data/taxa.csv
    fs.readFile(process.cwd() + '/../web.browser/app/data/taxa.csv', 'utf8', function (err,
data) {
        if (err) {
            console.log('Error: ' + err);
            return;
        }

        data = JSON.parse(data);
        console.log(data);
    });
});


// Synchronous Method.
Meteor.startup(function () {
    var fs = Npm.require('fs');
    // file originally saved as public/data/taxa.csv
    var data = fs.readFileSync(process.cwd() + '/../web.browser/app/data/taxa.csv', 'utf8');

    if (Icd10.find().count() === 0) {
        Icd10.insert({
            date:  new Date(),
            data:  JSON.parse(data)
        });
    }
});


Meteor.methods({
  parseCsvFile:function (){
    console.log('parseCsvFile');

    var fs = Npm.require('fs');
    // file originally saved as public/data/taxa.csv
    var data = fs.readFileSync(process.cwd() + '/../web.browser/app/data/taxa.csv', 'utf8');
    console.log('data', data);
  }
});
```

Read File Uploading online: https://riptutorial.com/meteor/topic/3119/file-uploading

# Chapter 22: Full Installation - Mac OSX

## Examples

### Install Node & NPM

This quickstart is written for Mac OSX Mavericks, and is a bit more verbose than other installation instructions. It should hopefully cover a few edge cases, such as setting your path, and configuring NPM, which can cause an installation to go awry.

```
    # install node
    # as of OSX Mavericks, we need the GUI installer (?!)
    # when a good command line alternative is found, we'll post it
    http://nodejs.org/download/

    # install npm
    curl -0 -L https://npmjs.org/install.sh | sh

    # check node is installed correctly
    node --version

    # check npm is installed correctly
    npm -version

    # find your npm path
    which npm

    # make sure npm is in your path
    sudo nano ~/.profile
      export PATH=$PATH:/usr/local/bin
```

### Meteor Installation Walkthrough

This quickstart is written for Mac OSX Mavericks, and is a bit more verbose than other installation instructions. It should hopefully cover a few edge cases, such as setting your path, and configuring NPM, which can cause an installation to go awry.

```
# install meteor
    curl https://install.meteor.com | sh

    # check it's installed correctly
    meteor --version

    # install node
    # as of OSX Mavericks, we need the GUI installer (?!)
    # when a good command line alternative is found, we'll post it
    http://nodejs.org/download/

    # install npm
    curl -0 -L https://npmjs.org/install.sh | sh

    # check node is installed correctly
    node --version
    # check npm is installed correctly
    npm -version

    # find your npm path
    which npm

    # make sure npm is in your path
    sudo nano ~/.profile
      export PATH=$PATH:/usr/local/bin
```

## Mongo Installation

Meteor doesn't exist in isolation, and it's common to install a number of extra tools for development, such as Mongo, Robomongo, Atom, Linters, etc.

```
# make sure mongo is in your local path
nano ~/.profile
  export PATH=$PATH:/usr/local/mongodb/bin

# or install it to the global path
nano /etc/paths
  /usr/local/mongodb/bin

# create mongo database directory
mkdir /data/
mkdir /data/db
chown -R username:admin /data

# run mongodb server
mongod
ctrl-c

# check that you can connect to your meteor app with stand-alone mongo
terminal-a$ meteor create helloworld
terminal-a$ cd helloworld
terminal-a$ meteor

terminal-b$ mongo -port 3001

# install robomongo database admin tool
http://robomongo.org/

# check you can connect to your mongo instance with robomongo
terminal-a$ meteor create helloworld
terminal-a$ cd helloworld
terminal-a$ meteor

Dock$ Robomongo > Create > localhost:3001
```

## Other Development Tools

```
# install node-inspector
terminal-a$  npm install -g node-inspector

# start meteor
terminal-a$  cd helloworld
```

```
terminal-a$  NODE_OPTIONS='--debug-brk --debug' mrt run

# alternatively, some people report this syntax being better
terminal-a$  sudo NODE_OPTIONS='--debug' ROOT_URL=http://helloworld.com meteor --port 80

# launch node-inspector along side your running app
terminal-b$  node-inspector

# go to the URL given by node-inspector and check it's running
http://localhost:8080/debug?port=5858

# install jshint
npm install -g jshint
```

Read Full Installation - Mac OSX online: https://riptutorial.com/meteor/topic/3294/full-installation---mac-osx

# Chapter 23: Horizontal Scaling

## Examples

### Deploying an Application with Separated Database (MONGO_URL)

You'll need to separate out your application layer from your database layer, and that means specifying the MONGO_URL. Which means running your app through the bundle command, uncompressing it, setting environment variables, and then launching the project as a node app. Here's how...

```
#make sure you're running the node v0.10.21 or later
npm cache clean -f
npm install -g n
sudo n 0.10.21

# bundle the app
mkdir myapp
cd myapp
git clone http://github.com/myaccount/myapp
meteor bundle --directory ../deployPath
cd ../deployPath

# make sure fibers is installed, as per the README
export MONGO_URL='mongodb://127.0.0.1:27017/mydatabase'
export PORT='3000'
export ROOT_URL='http://myapp.com'

# run the site
node main.js
```

### Replica Set Configuration

Then go into the mongo shell and initiate the replica set, like so:

```
mongo

> rs.initiate()
PRIMARY> rs.add("mongo-a")
PRIMARY> rs.add("mongo-b")
PRIMARY> rs.add("mongo-c")
PRIMARY> rs.setReadPref('secondaryPreferred')
```

### Configuring a Replica Set to Use Oplogging

The replica set will need an oplog user to access the database.

```
mongo

PRIMARY> use admin
PRIMARY>
```

```
db.addUser({user:"oplogger",pwd:"YOUR_PASSWORD",roles:[],otherDBRoles:{local:["read"]}});
PRIMARY> show users
```

## Oplog Upstart Script

Your upstart script will need to be modified to use multiple IP addresses of the replica set.

```
start on started mountall
stop on shutdown

respawn
respawn limit 99 5

script
    # our example assumes you're using a replica set and/or oplog integreation
    export MONGO_URL='mongodb://mongo-a:27017,mongo-b:27017,mongo-c:27017/meteor'

    # here we configure our OPLOG URL
    export MONGO_OPLOG_URL='mongodb://oplogger:YOUR_PASSWORD@mongo-a:27017,mongo-
b:27017,mongo-c:27017/local?authSource=admin'

    # root_url and port are the other two important environment variables to set
    export ROOT_URL='http://myapp.mydomain.com'
    export PORT='80'

    exec /usr/local/bin/node /var/www/production/main.js >> /var/log/node.log 2>&1
end script
```

## Sharding

Oplog Tailing on Sharded Mongo

Read Horizontal Scaling online: https://riptutorial.com/meteor/topic/3706/horizontal-scaling

# Chapter 24: Integration of 3rd Party APIs

## Examples

### Basic HTTP Call

Conceptually, integrate 3rd party REST APIs can be as simple as adding the `http` package and making a call to the external endpoint.

```
meteor add http
```

```
HTTP.get('http://foo.net/api/bar/');
```

### Create A Package For Your API Wrapper

Basic HTTP calls don't provide code-reusability, however. And they can get confused with all the other features you're trying to implement. For those reasons, it's common to implement an API wrapper.

```
Foo = {
  identify: function(input){
    return Http.get('http://foo.net/api/identify/' + input);
  },
  record_action_on_item: function(firstInput, secondInput){
    return Http.put('http://foo.net/api/record_action_on_item/' + firstInput + '&' +
secondInput);
  }
}
```

Meteor supports Http.get(), Http.post(), Http.put(), etc, so that's undoubtably the best way to call your REST API. http://docs.meteor.com/#http_get

If the API is chatty and verbose, you may receive multiple packets; in which case you'll need to reassemble them. This is a big hassle. If you think the API is returning multiple packets, you're probably going to want to use the 'request' npm module on the server. You'll want to use a `Npm.require('request')`. https://github.com/mikeal/request

### Create an Atmosphere Package For Your API Wrapper

After creating an API wrapper, it's likely that you may want to create an Atmosphere package to redistribute it and share it between applications. The files of your package will probably look something like this.

```
packages/foo-api-wrapper/package.js
packages/foo-api-wrapper/readme.md
packages/foo-api-wrapper/foo.api.wrapper.js
```

In particular, your `foo-api-wrapper/package.js` file will want to look something like this:

```
Package.describe({
  summary: "Atmosphere package that impliments the Foo API.",
  name: "myaccount:foo",
  version: '0.0.1'
});

Package.on_use(function (api) {
    api.export('Foo');
    api.addFiles('foo.api.wrapper.js', ["client","server"]);
});
```

And your `foo-api-wrapper/foo.api.wrapper.js` should contain the `Foo` API wrapper object.

## Include the API Package in your Application

At this point, you're still building your package, so you'll need to add the package to your application:

```
meteor add myaccount:foo
```

And eventually publish it to Atmosphere:

```
meteor publish myaccount:foo
```

## Using the API Wrapper Object in your App

Now that we have all those pieces put together, you should now be able to make calls like the following from within your app:

```
Foo.identify('John');
Foo.record_action_on_item('view', "HackerNews');
```

Obviously you'll want to adjust function names, arguments, urls, and the like, to create the proper syntax for the API.

Read Integration of 3rd Party APIs online: https://riptutorial.com/meteor/topic/3118/integration-of-3rd-party-apis

# Chapter 25: Logging

## Examples

### Basic Server Side Logging

The first step to logging is simply to run Meteor from the shell, and you'll get the server logs in the command console.

```
meteor
```

The next step is to pipe the contents of std_out and std_err to a logfile, like so:

```
meteor > my_app_log.log 2> my_app_err.log
```

### Client Side Logging Tools

Once you have your server side logging in place, it's time to hop over to the client side. If you haven't explored the console API, be prepared for a treat. There's actually all sorts of things that you can do with the built in Console API that's native to every Chrome and Safari installation. So much so, in fact, that you may find yourself not needing Winston or other logging frameworks.

The first thing you'll want to do is install client side logging and developer tools. Chrome and Safari both ship with them, but Firefox requires the Firebug extension.

Firebug Extension

Then, you'll want to check out the Console API documentation. The following two documents are invaluable resources for learning console logging.

Chrome Developer Tools

Firebug (Client)

### Advanced Server Logging Tools

Once you have both your server-side logging running, and your client side development tools, you can start looking at Meteor specific extensions like the Meteor Chrome DevTools Extension. This lets you actually observe server logging in the client! Because the database is everywhere. As is logging.

Chrome DevTools Extension (Server)

### Logging error on database flap

The following example is from 0.5 - 0.7 days, and illustrates how to log an error when the database hasn't populated the client side cursor yet.

```
Template.landingPage.postsList = function(){
  try{
    return Posts.find();
  }catch(error){
    //color code the error (red)
    console.error(error);
  }
}
```

## Logging info on the data context in a template helper

The following uses the Chrome Logging API. If the `.group()` syntax is used in multiple templates, it will graphically organize the console logs from different templates into a hierarchical tree.

You can also see how to inspect the current data context, and how to stringify data.

```
Template.landingPage.getId = function(){
  // using a group block to illustrate function scoping
  console.group('coolFunction');

  // inspect the current data object that landingPage is using
  console.log(this);

  // inspect a specific field of the locally scoped data object
  console.log(JSON.stringify(this._id);

  // close the function scope
  console.groupEnd();
  return this._id;
}
```

## Logging events and user interactions

Simple example of using the Chrome Logging API.

```
Template.landingPage.events({
  'click .selectItemButton':function(){
    // color code and count the user interaction (blue)
    console.count('click .selectItemButton');
  }
});
```

## Logging with log level variables

Logging can often clutter up the console, so it's common to define log levels to control what detail of data is getting logged. A common pattern is to specify a log level variables.

```
var DEBUG = false;
var TRACE = false;
Template.landingPage.events({
  'click .selectItemButton':function(){
    TRACE && console.count('click .selectItemButton');

    Meteor.call('niftyAction', function(errorMessage, result){
        if(errorMessage){
            DEBUG && console.error(errorMessage);
        }
    });
  }
});
```

## Disable Logging in Production

Some teams find that they want to leave console log statements in their code, but not have them display in production. They will override the logging functions if a variable isn't set (possibly an environment variable). Additionally, this may qualify as a security feature in some situations.

```
if (!DEBUG_MODE_ON) {
    console = console || {};
    console.log = function(){};

    console.log = function(){};
    console.error = function(){};
    console.count = function(){};
    console.info = function(){};
}
```

## Winston

If you need something more powerful than the default logging options, you might want to look at a tool like Winston. Go to Atmosphere, and simply search for one of the many Winston packages available.

https://atmospherejs.com/?q=winston

Be warned, however - Winston is a sophisticated product, and while it exposes a lot of functionality, it will also add a layer of complexity to your application.

## Loglevel

A special mention should be made for the community developed LogLevel package. It appears to strike a balance between being lightweight and simple to use, while working well with Meteor's bundle pipeline and preserving line numbers and filenames.

https://atmospherejs.com/practicalmeteor/loglevel

Read Logging online: https://riptutorial.com/meteor/topic/3376/logging

# Chapter 26: Meteor + React

## Remarks

React is a JavaScript library for building user interfaces. It's open source, developed and maintained by Facebook. Meteor has production-ready support for React.

Resources:

- React tutorial
- Meteor + React tutorial

## Examples

### Setup and "Hello World"

Add React to your project:

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

Create the `client/helloworld.jsx` file to display a simple React component:

```
import React, { Component } from 'react';
import { mount } from 'react-mounter';

// This component only renders a paragraph containing "Hello World!"
class HelloWorld extends Component {
  render() {
    return <p>Hello World!</p>;
  }
}

// When the client application starts, display the component by mounting it to the DOM.
Meteor.startup(() => {
  mount(HelloWorld);
});
```

### Create reactive container using createContainer

Let's say there's a collection called `Todos` and the `autopublish` package is added. Here is the basic component.

```
import { createContainer } from 'meteor/react-meteor-data';
import React, { Component, PropTypes } from 'react';
import Todos from '/imports/collections/Todos';

export class List extends Component {
  render() {
    const { data } = this.props;
    return (
      <ul className="list">
        {data.map(entry => <li {...entry} />)}
      </ul>
    )
  }
}

List.propTypes = {
  data: PropTypes.array.isRequired
};
```

At the bottom, you can add a container to feed the reactive data into the component. It would look like this.

```
export default createContainer(() => {
  return {
    data: Todos.find().fetch()
  };
}, List);
```

## Displaying a MongoDB collection

This example shows how a MongoDB collection can be displayed in a React component. The collection is continuously synchronized between server and client, and the page instantly updates as database contents change.

To connect React components and Meteor collections, you'll need the `react-meteor-data` package.

```
$ meteor add react-meteor-data
$ meteor npm install react-addons-pure-render-mixin
```

A simple collection is declared in `both/collections.js`. Every source file in `both` directory is both client-side and server-side code:

```
import { Mongo } from 'meteor/mongo';

// This collection will contain a list of random numbers
export const Numbers = new Mongo.Collection("numbers");
```

The collection needs to be published on the server. Create a simple publication in `server/publications.js`:

```
import { Meteor } from 'meteor/meteor';
import { Numbers } from '/both/collections.js';

// This publication synchronizes the entire 'numbers' collection with every subscriber
Meteor.publish("numbers/all", function() {
  return Numbers.find();
});
```

Using the `createComponent` function we can pass reactive values (like the `Numbers` collection) to a

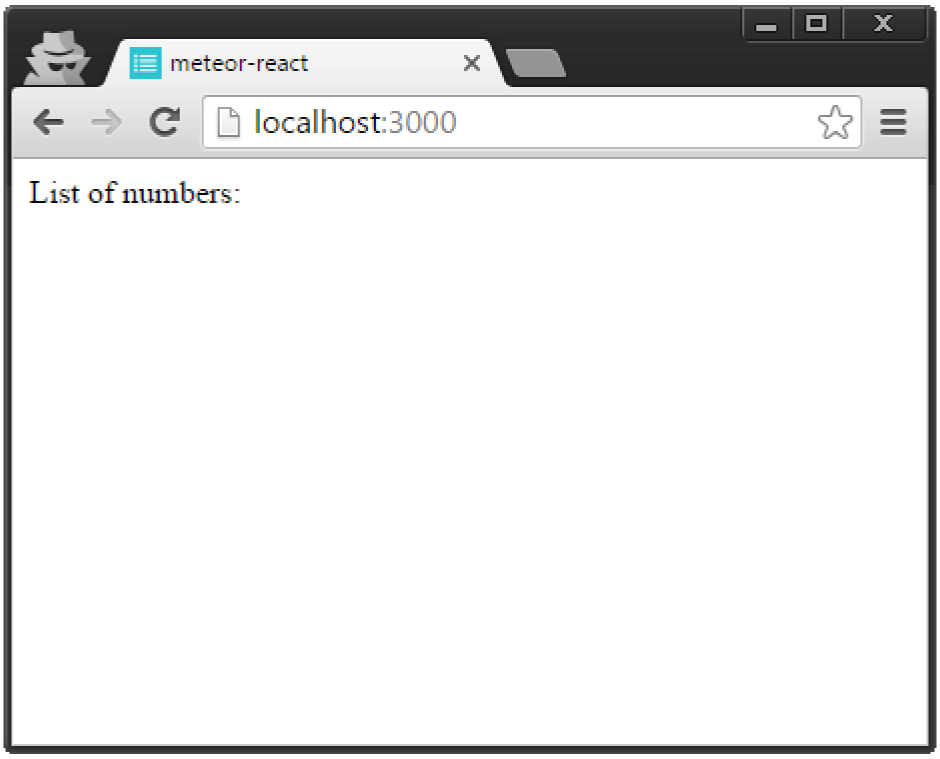
React component. `client/shownumbers.jsx`:

```
import React from 'react';
import { createContainer } from 'meteor/react-meteor-data';
import { Numbers } from '/both/collections.js';

// This stateless React component renders its 'numbers' props as a list
function _ShowNumbers({numbers}) {
  return <div>List of numbers:
    <ul>
      // note, that every react element created in this mapping requires
      // a unique key - we're using the _id auto-generated by mongodb here
      {numbers.map(x => <li key={x._id}>{x.number}</li>)}
    </ul>
  </div>;
}

// Creates the 'ShowNumbers' React component. Subscribes to 'numbers/all' publication,
// and passes the contents of 'Numbers' as a React property.
export const ShowNumbers = createContainer(() => {
  Meteor.subscribe('numbers/all');
  return {
    numbers: Numbers.find().fetch(),
  };
}, _ShowNumbers);
```

Initially the database is probably empty.



List of numbers:

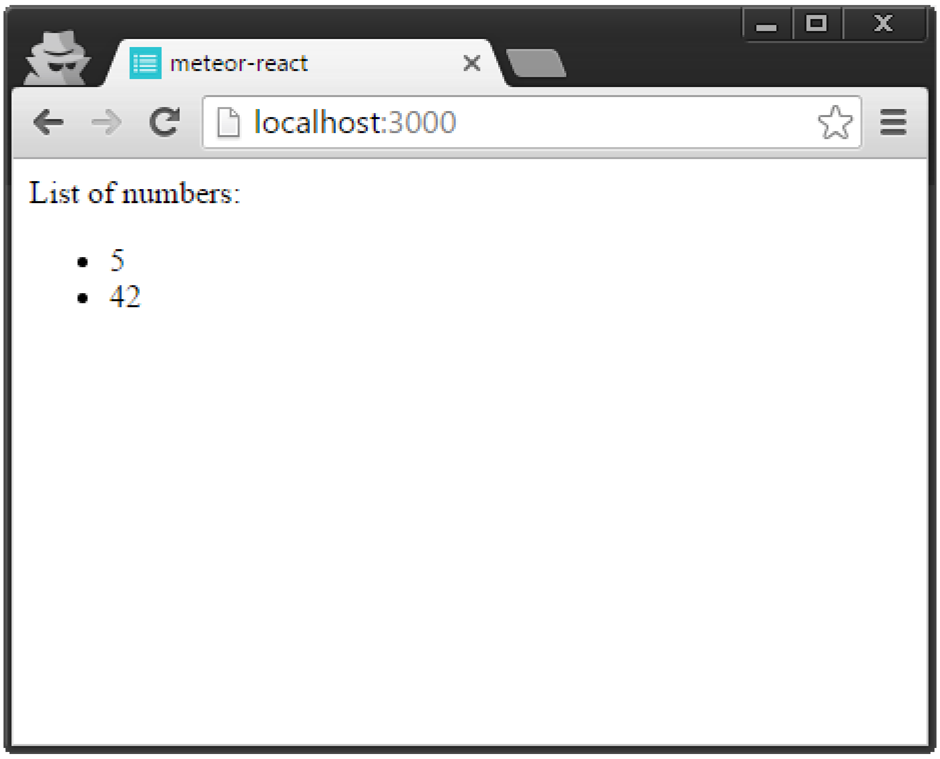
Add entries to MongoDB and watch as the page updates automatically.

```
$ meteor mongo
MongoDB shell version: 3.2.6
connecting to: 127.0.0.1:3001/meteor

meteor:PRIMARY> db.numbers.insert({number: 5});
```

```
WriteResult({ "nInserted" : 1 })

meteor:PRIMARY> db.numbers.insert({number: 42});
WriteResult({ "nInserted" : 1 })
```



List of numbers:

- 5
- 42

Read Meteor + React online: https://riptutorial.com/meteor/topic/3121/meteor-plus-react

# Chapter 27: Meteor + React + ReactRouter

## Introduction

This document will show how to use ReactRouter with Meteor and React. From zero to a working app, including roles and authentication.

I'll show each step with an example

1- Create the project

2- Add React + ReactRouter

3- Add Accounts

4- Add Roles packages

## Examples

### Create the project

1- First all, install https://www.meteor.com/install

2- Create a project. (`--bare` is to create an empty project)

```
meteor create --bare MyAwesomeProject
```

3- Create the minimal file structure (`-p` to create intermediate directories):

```
cd MyAwesomeProject

mkdir -p client server imports/api imports/ui/{components,layouts,pages}
imports/startup/{client,server}
```

4- Now, create an HTML file in client/main.html

```
<head>
   <meta charset="utf-8">
   <title>My Awesome Meteor_React_ReactRouter_Roles App</title>
</head>

<body>
  Welcome to my Meteor_React_ReactRouter_Roles app
</body>
```

5- Make sure it's working: (3000 is the default port, so you can actually skip the '-p 3000')

```
meteor run -p 3000
```

and opening your browser on 'localhost:3000'

## Note:

- I'm skipping some other files that you will need to create, to make things shorter. Specifically, you will need to create some index.js files in *client* , *imports/startup/{client,server}* and *server* directories.
- You can view a full example in https://github.com/rafa-lft/Meteor_React_Base. Look for tag *Step1_CreateProject*

### Add React + ReactRouter

If necessary, change to your project directory `cd MyAwesomeProject`

1- Add react and react-router

```
meteor npm install --save react-router@3.0.0 react@15.5.4 react-dom@15.5.4
```

2- Edit client/main.html, and replace the content will be:

```
<body>
   <div id="react-root"></div>
</body>
```

Whatever the reactRouter decides to show, it will show it in the '#react-root' element

3- Create the Layouts file in imports/ui/layouts/App.jsx

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';


class App extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <div>
        {this.props.children}
      </div>
    );
  }
}

App.propTypes = {
  children: PropTypes.node
};

export default App;
```

4- Create the Routes file in imports/startup/client/Routes.jsx

```
import ReactDOM from 'react-dom';
import React, { Component } from 'react';
import { Router, Route, IndexRoute, browserHistory } from 'react-router';

import App from '../../ui/layouts/App.jsx';

import NotFound from '../../ui/pages/NotFound.jsx';
import Index from '../../ui/pages/Index.jsx';


class Routes extends Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <Router history={ browserHistory }>
        <Route path="/" component={ App }>
          <IndexRoute name="index" component={ Index }/>
          <Route path="*" component={ NotFound }/>
        </Route>
      </Router>
    );
  }
}

Routes.propTypes = {};


Meteor.startup(() =>{
  ReactDOM.render(
    <Routes/>,
    document.getElementById('react-root')
  );
});
```

## Note:

- I'm skipping some other files that you will need to create, to make things shorter. Specifically, check for imports/ui/pages{Index.jsx,NotFound.jsx}.
- You can view a full example in https://github.com/rafa-lft/Meteor_React_Base. Look for tag *Step2_ReactRouter*

### Step 3- Add Accounts

If necessary, change to your project directory `cd MyAwesomeProject`

1- Add accounts packages: `meteor add accounts-base accounts-password react-meteor-data`

2- Add the routes to *login* and *signup* pages in imports/startup/Routes.jsx The *render()* method will be as follows:

```
   render() {
     return (
       <Router history={ browserHistory }>
         <Route path="/" component={ App }>
           <IndexRoute name="index" component={ Index }/>
           <Route name="login" path="/login" component={ Login }/>
           <Route name="signup" path="/signup" component={ Signup }/>
           <Route name="users" path="/users" component={ Users }/>
           <Route name="editUser" path="/users/:userId" component={ EditUser }/>
           <Route path="*" component={ NotFound }/>
         </Route>
       </Router>
     );
   }
```

## Note:

- I'm skipping some other files that you will need, to make things shorter. Specifically, check imports/startup/server/index.js imports/ui/layouts/{App,NavBar}.jsx and import/ui/pages/{Login,Signup,Users,EditUser}.jsx

- You can view a full example in https://github.com/rafa-lft/Meteor_React_Base. Look for tag *Step3_Accounts*

### Add roles

1- Add roles package ( https://github.com/alanning/meteor-roles)

```
meteor add alanning:roles
```

2- Create some roles constants. In file imports/api/accounts/roles.js

```
const ROLES = {
  ROLE1: 'ROLE1',
  ROLE2: 'ROLE2',
  ADMIN: 'ADMIN'
};

export default ROLES;
```

3- I'll not show how to add/update roles on a user, just will mention that on server side, you can set user roles by `Roles.setUserRoles(user.id, roles);` Check for more info in https://github.com/alanning/meteor-roles and http://alanning.github.io/meteor-roles/classes/Roles.html

4- Assuming you already setup all the accounts and roles files (see full example in https://github.com/rafa-lft/Meteor_React_Base. Look for tag *Step4_roles*) we can now create a method that will be in charge of allowing (or not) access to the different routes. In imports/startup/client/Routes.jsx

```
class Routes extends Component {
  constructor(props) {
    super(props);
  }

  authenticate(roles, nextState, replace) {
    if (!Meteor.loggingIn() && !Meteor.userId()) {
      replace({
        pathname: '/login',
        state: {nextPathname: nextState.location.pathname}
      });
      return;
    }
    if ('*' === roles) { // allow any logged user
      return;
    }
    let rolesArr = roles;
    if (!_.isArray(roles)) {
      rolesArr = [roles];
    }
    // rolesArr = _.union(rolesArr, [ROLES.ADMIN]);// so ADMIN has access to everything
    if (!Roles.userIsInRole(Meteor.userId(), rolesArr)) {
      replace({
        pathname: '/forbidden',
        state: {nextPathname: nextState.location.pathname}
      });
    }
  }

  render() {
    return (
      <Router history={ browserHistory }>
        <Route path="/" component={ App }>
          <IndexRoute name="index" component={ Index }/>
          <Route name="login" path="/login" component={ Login }/>
          <Route name="signup" path="/signup" component={ Signup }/>

          <Route name="users" path="/users" component={ Users }/>

          <Route name="editUser" path="/users/:userId" component={ EditUser }
                 onEnter={_.partial(this.authenticate, ROLES.ADMIN)} />


          {/* *******************
           Below links are there to show Roles authentication usage.
           Note that you can NOT hide them by
           { Meteor.user() && Roles.userIsInRole(Meteor.user(), ROLES.ROLE1) &&
           <Route name=.....
           }
           as doing so will change the Router component on render(), and ReactRouter will
complain with:
           Warning: [react-router] You cannot change <Router routes>; it will be ignored

           Instead, you can/should hide them on the NavBar.jsx component... don't worry: if
someone tries to access
           them, they will receive the Forbidden.jsx component
           *************/ }
          <Route name="forAnyOne" path="/for_any_one" component={ ForAnyone }/>

          <Route name="forLoggedOnes" path="/for_logged_ones" component={ ForLoggedOnes }
                 onEnter={_.partial(this.authenticate, '*')} />
          <Route name="forAnyRole" path="/for_any_role" component={ ForAnyRole }
                 onEnter={_.partial(this.authenticate, _.keys(ROLES))}/>

          <Route name="forRole1or2" path="/for_role_1_or_2" component={ ForRole1or2 }
                 onEnter={_.partial(this.authenticate, [ROLES.ROLE1, ROLES.ROLE2])} />

          <Route name="forRole1" path="/for_role1" component={ ForRole1 }
                 onEnter={_.partial(this.authenticate, ROLES.ROLE1)}/>

          <Route name="forRole2" path="/for_role2" component={ ForRole2 }
                 onEnter={_.partial(this.authenticate, ROLES.ROLE2)} />


          <Route name="forbidden" path="/forbidden" component={ Forbidden }/>

          <Route path="*" component={ NotFound }/>
        </Route>
      </Router>
    );
  }
}
```

We added an *onEnter* trigger to some routes. For those routes, we are also passing which Roles are allowed to enter. Note that the onEnter callback, receives 2 params originally. We are using underscore's partial (http://underscorejs.org/#partial), to add another one (roles) The *authenticate* method (called by onEnter) receives the roles and:

- Check if the user is logged in at all. If not, redirects to '/login'.
- If roles === '*' we assume any logged in user can enter, so we allow it
- Else, we verify if the user is allowed (Roles.userIsInRole) and if not, we redirect to forbidden.
- Optionally, you can uncomment a line, so ADMIN has access to everything.

The code has several examples of different routes that are allowed for anyone (no onEnter callback), for any logged user, for any logged user with at least 1 role, and for specific roles.

Also note, that ReactRouter (at least on version 3), doesn't allow to modificate the routes on Render. So you can not hide the routes within the Routes.jsx. For that reason, we redirects to /forbidden in the authenticate method.

5- A common bug with ReactRouter and Meteor, relates to user status updates not being shown. For example the user logged out, but we are still showing his/her name on the nav-bar. That happens because Meteor.user() has changed, but we are not re-rendering.

That bug can be solved by calling Meteor.user() in the createContainer. Here is an example of it, used in imports/ui/layouts/NavBar.jsx:

```
export default createContainer((/* {params}*/) =>{
  Meteor.user(); // so we render again in logout or if any change on our User (ie: new roles)
  const loading = !subscription.ready();
  return {subscriptions: [subscription], loading};
}, NavBar);
```

## Note:

- I'm skipping some other files that you will need, to make things shorter. Specifically, check imports/startup/server/index.js imports/ui/layouts/{App,NavBar}.jsx and import/ui/pages/{Login,Signup,Users,EditUser}.jsx
- You can view a full example in https://github.com/rafa-lft/Meteor_React_Base. Look for tag *Step4_roles*

Read Meteor + React + ReactRouter online: https://riptutorial.com/meteor/topic/10114/meteor-plus-react-plus-reactrouter

# Chapter 28: Meteor User Accounts

## Examples

### Meteor accounts package

You have a few options when it comes to logging in with Meteor. The most common method is using `accounts` for Meteor.

## Accounts-password

If you want users to be able to create and register on your site, you can use `accounts-password`.

Install the package using `meteor add accounts-password`.

To create a user, you need to use `Accounts.createUser(options, [callback])`

`options` has to be an object with the following properties:

- `username`: The user's username as a string..
- `email`: The user's email as a string.
- `password`: The user's (not encrypted) password as a string.
- `profile`: The user's optional extra data as an object. This can be for example the user's first and last name. `profile` is optional, however.

The callback returns 1 variable if there is an error, which is a Meteor.Error object.

You are only required to use either the `username` or the `email`, so you can create a user with username but no email, and vice versa. You can also use both.

It returns the newly created user ID if everything went correctly.

So, you can for example use this:

```
// server side
var id = Accounts.createUser({
    username: "JohnDoe",
    email: "JohnDoe@gmail.com",
    password: "TheRealJohn123",
    profile: {
        firstName: "John",
        lastName: "Doe"
    }
}, function(err) {
    console.log(err.reason);
});
```

It will automatically log you in as well if the user was succesfully created.

That is the creating part. To log in you need to use `Meteor.loginWithPassword(identifier, password, [callback])` on the client side.

`identifier` is the `username`, `email` or `userId` as a string from your user. `password` is the (not encrypted) `password` of the user.

The callback returns one variable if there is an error, which is a Meteor.Error object.

Example:

```
// client side
Meteor.loginWithPassword("JohnDoe", "TheRealJohn123", function(err) {
    console.log(err.reason);
});
```

And that is it for the basic creating of accounts and logging in.

## Accessing user data

You can check on the client side if the user is logged in by calling `Meteor.userId()` which will return their `userId` if they are logged in, and `undefined` if they are not logged in.

You can get some of the info from `Meteor.user()`. It will return undefined if the user is not logged in, and some user data if they are. It will not give you any passwords by default, by default it will show the userId of the user, the username and the profile object.

If you want to check if a user is logged in on a page, you can also use the `currentUser` helper. It will return the contents of `Meteor.user()`. Example:

```
{{#if currentUser}}
    <h1>Hello there, {{currentUser.username}}!</h1>
{{else}}
    <h1>Please log in.</h1>
{{/if}}
```

## Other accounts functions

There are some other functions that work for every accounts package.

You can log out using `Meteor.logout()`

### Don’t use the default profile field

There’s a tempting existing field called `profile` that is added by default when a new user registers. This field was historically intended to be used as a scratch pad for user-specific data - maybe their image avatar, name, intro text, etc. Because of this, **the `profile` field on every user is automatically writable by that user from the client**. It’s also automatically published to the client for that particular user.

It turns out that having a field writable by default without making that super obvious might not be the best idea. There are many stories of new Meteor developers storing fields such as `isAdmin` on `profile`… and then a malicious user can easily set that to true whenever they want, making themselves an admin. Even if you aren't concerned about this, it isn't a good idea to let malicious users store arbitrary amounts of data in your database.

Rather than dealing with the specifics of this field, it can be helpful to just ignore its existence entirely. You can safely do that as long as you deny all writes from the client:

```
// Deny all client-side updates to user documents
Meteor.users.deny({
  update() { return true; }
});
```

Even ignoring the security implications of profile, it isn't a good idea to put all of your app's custom data onto one field. Meteor's data transfer protocol doesn't do deeply nested diffing of fields, so it's a good idea to flatten out your objects into many top-level fields on the document.

Read Meteor User Accounts online: https://riptutorial.com/meteor/topic/6219/meteor-user-accounts

# Chapter 29: Mobile Apps

## Examples

### Page Layout on Different Devices - CSS

If your application is going to run on different devices, it's going to need to render to different ViewPorts, based on the device size. You can deal with this in two ways: with javascript rules, or CSS media styles. If you've been using a MVC or MVVM library, such as Angular or Ember (or Blaze, for that matter) and have only been targeting a single device or hardware platform, you may need to rethink your MVC model as different hardware ViewPorts are introduced to your application.

```
// desktop
@media only screen and (min-width: 960px) {
}

// landscape orientation
@media only screen and (min-width: 768px) {
}

// portrait orientation
@media only screen and (min-width: 480px) {
}
```

You'll need to figure out if you want to break the styles at 768px (portrait mode) or at 1024 pixels (landscape). That's assuming your target mobile device is the iPad, which uses a 3:4 ratio. Otherwise, you'll need to work out the aspect ratios of the devices you do want to target, and figure out the threshold levels from there.

### Fixed Sized Windows

If you're going to be designing layouts with fixed size screens for different mobile devices, you may want to mirror that design when running your app on a desktop. The following method fixes the size of the window OUTSIDE of PhoneGap, giving a fixed-sized window on the desktop. Sometimes it's easiest to manage user's expectations and UI design by limiting options!

```
// create a window of a specific size
var w=window.open('','', 'width=100,height=100');
w.resizeTo(500,500);

// prevent window resize
var size = [window.width,window.height];  //public variable
$(window).resize(function(){
    window.resizeTo(size[0],size[1]);
});
```

### Offline Caching

To get all of this to work, you'll probably need offline support, which means caching application data and user data.

```
meteor add appcache
meteor add grounddb
```

## Disable Scroll-Bounce

On desktop apps, you may want to disable scroll-bounce, to give your app a more native feel. You can do this with javascript, by disabling how the browser controls the DOM:

```
// prevent scrolling on the whole page
// this is not meteorish; TODO: translate to meteor-centric code
document.ontouchmove = function(e) {e.preventDefault()};

// prevent scrolling on specific elements
// this is not meteorish; TODO: translate to meteor-centric code
scrollableDiv.ontouchmove = function(e) {e.stopPropagation()};
```

Alternatively, you can use css, and the overflow and scrolling styles.

```
#appBody {
  overflow: hidden;
}

#contentContainer {
  .content-scrollable {
    overflow-y: auto;
    -webkit-overflow-scrolling: touch;
  }
}
```

The object model needed for the above to work looks something like this:

```
<div id="appBody">
  <div id="contentContainer">
    <div class="content-scrollable">
      <!-- content -->
    </div>
  </div>
</div>
```

## Multitouch & Gestures

Mobile devices generally don't have keyboards, so you'll need to add some haptic controllers to your application. The two popular packages that people seem to be using is FastClick and Hammer. Installation is easy.

```
meteor add fastclick
meteor add hammer:hammer
```

FastClick requires nearly no configuration, while Hammer requires a bit of work to wire up. The cononical example from the Todos app looks like this:

```
Template.appBody.onRendered(function() {
  if (Meteor.isCordova) {
    // set up a swipe left / right handler
    this.hammer = new Hammer(this.find('#appBody'));
    this.hammer.on('swipeleft swiperight', function(event) {
      if (event.gesture.direction === 'right') {
        Session.set(MENU_KEY, true);
      } else if (event.gesture.direction === 'left') {
        Session.set(MENU_KEY, false);
      }
    });
  }
});
```

## Create your Icons and Splash Screen Assets

Before you compile your app and run it on your device, you'll need create some icons and splash screens, and add a `mobile-config.js` file to your app.

```
App.icons({
  // iOS
  'iphone': 'resources/icons/icon-60x60.png',
  'iphone_2x': 'resources/icons/icon-60x60@2x.png',
  'ipad': 'resources/icons/icon-72x72.png',
  'ipad_2x': 'resources/icons/icon-72x72@2x.png',

  // Android
  'android_ldpi': 'resources/icons/icon-36x36.png',
  'android_mdpi': 'resources/icons/icon-48x48.png',
  'android_hdpi': 'resources/icons/icon-72x72.png',
  'android_xhdpi': 'resources/icons/icon-96x96.png'
});

App.launchScreens({
  // iOS
  'iphone': 'resources/splash/splash-320x480.png',
  'iphone_2x': 'resources/splash/splash-320x480@2x.png',
  'iphone5': 'resources/splash/splash-320x568@2x.png',
  'ipad_portrait': 'resources/splash/splash-768x1024.png',
  'ipad_portrait_2x': 'resources/splash/splash-768x1024@2x.png',
  'ipad_landscape': 'resources/splash/splash-1024x768.png',
  'ipad_landscape_2x': 'resources/splash/splash-1024x768@2x.png',

  // Android
  'android_ldpi_portrait': 'resources/splash/splash-200x320.png',
  'android_ldpi_landscape': 'resources/splash/splash-320x200.png',
  'android_mdpi_portrait': 'resources/splash/splash-320x480.png',
  'android_mdpi_landscape': 'resources/splash/splash-480x320.png',
  'android_hdpi_portrait': 'resources/splash/splash-480x800.png',
  'android_hdpi_landscape': 'resources/splash/splash-800x480.png',
  'android_xhdpi_portrait': 'resources/splash/splash-720x1280.png',
  'android_xhdpi_landscape': 'resources/splash/splash-1280x720.png'
});
```

## Meteor Cordova Architecture Pipeline

Now it's time to go through the Meteor Cordova Phonegap Integration documentation.

Since that documentation was written, XCode and Yosemite have been released, which has caused some hiccups in installation. Here are the steps we had to go through to get Meteor compiled to an iOS device.

- Upgrade to Yosemite.
- Delete XCode (drag from Applications folder to Trashcan)
- Install XCode 6.1 from app store.
- Agree to various terms and conditions.

```
# 5.  clone and rebuild the ios-sim locally
#     (this step will not be needed in future releases)
git clone https://github.com/phonegap/ios-sim.git
cd ios-sim
rake build

# 6.  make sure we can update the .meteor/packages locations
#     (this step will not be needed in future releases)
sudo chmod -R 777 ~/.meteor/packages

# 7.  copy the new build into Meteor locations
#     (this step will not be needed in future releases)
for i in `find ~/.meteor/packages/meteor-tool/ -name ios-sim -type f`; do
  cp -R ./build/Release/ios-sim "$i"
done

# 8.  install the ios platform to your app
cd myapp
meteor list-platforms
meteor add-platform ios
meteor list-platforms

# 9.  and that there aren't dead processes
ps -ax
kill -9 <pid>
# /Users/abigailwatson/.meteor/packages/meteor-
tool/.1.0.35.wql4jh++os.osx.x86_64+web.browser+web.cordova/meteor-tool-
os.osx.x86_64/dev_bundle/mongodb/bin/mongod
# tail -f /Users/abigailwatson/Code/Medstar/dart/webapp/.meteor/local/cordova-
build/platforms/ios/cordova/console.log

# 10.  make sure there are correct permissions on the application (important!)
sudo chmod -R 777 .meteor/local/

# 11.  run app
meteor run ios

# 12.  if that doesn't work, clear the directory
sudo rm -rf .meteor/local

# 13a.  run meteor again to create the default browser build
meteor

# 13b.  run it a second time so bootstrap and other packages get downloaded into the browser
build
ctrl-x
meteor
```

```
# 14.  then run the ios version
ctrl-x
meteor run ios
```

XCode should launch during the process. Select your simulator and press the 'Play' button.

## IOS Development

- Register your Apple Developer Account
- Register an App ID for your app
- Register the UUID of your testing devices
- Generate an iOS App Development provisioning profile
    - Generate a CertificateSigningRequest from KeychainAccess
    - Submit CertificateSigningRequest to https://developer.apple.com/account/ios/profile/profileCreate.action
    - Download and doubleclick the certificate to import into Keychain
- Go to XCode > Preferences > Accounts and register your Apple Developer Account

## IOS Device Testing

- Make sure your development workstation and iPhone are connected to the same WiFi network. Tethering, hotspots, and other ad-hoc networking won't work.
- Run `sudo meteor run ios-device`
- Deploy to your device!

## Configure your Cordova project (config.xml)

Meteor reads a `mobile-config.js` file in the root of your app directory during build, and uses the settings specified there to generate Cordova’s `config.xml`.

```
Project_folder
 ├── /.meteor
 └── mobile-config.js
```

Most configurations can be achieved with `mobile-config.js` (app metadata, preferences, icons and launchscreens, as well as Cordova plugins installation parameters).

```
App.info({
  id: 'com.example.matt.uber',
  name: 'über',
  description: 'Get über power in one button click',
  author: 'Matt Development Group',
  email: 'contact@example.com',
  website: 'http://example.com'
});

// Set up resources such as icons and launch screens.
App.icons({
  'iphone': 'icons/icon-60.png',
  'iphone_2x': 'icons/icon-60@2x.png',
  // ... more screen sizes and platforms ...
});

App.launchScreens({
  'iphone': 'splash/Default~iphone.png',
  'iphone_2x': 'splash/Default@2x~iphone.png',
  // ... more screen sizes and platforms ...
});

// Set PhoneGap/Cordova preferences
App.setPreference('BackgroundColor', '0xff0000ff');
App.setPreference('HideKeyboardFormAccessoryBar', true);
App.setPreference('Orientation', 'default');
App.setPreference('Orientation', 'all', 'ios');

// Pass preferences for a particular PhoneGap/Cordova plugin
App.configurePlugin('com.phonegap.plugins.facebookconnect', {
  APP_ID: '1234567890',
  API_KEY: 'supersecretapikey'
});
```

*Do not* manually edit the `/.meteor/local/cordova-build/config.xml` file, as it will be regenerated at every `meteor run ios/android` or `meteor build`, hence you will lose all your modifications.

Reference page: Meteor Guide > Build > Mobile > Configuring your app

## Detecting the deviceready event

Of course, the best way to detect mobile is for the hardware to notify you directly. Cordova PhoneGap exposes a 'deviceready' event, that you can add an event listener to.

```
 document.addEventListener('deviceready', function(){
  Session.set('deviceready', true);
 }, false);
```

Read Mobile Apps online: https://riptutorial.com/meteor/topic/3705/mobile-apps

# Chapter 30: Mongo Collections

## Remarks

A useful way to think about Mongo collections is in terms of Who, What, When, Where, Why, and How. Mongo has the following optimizations for different types of data:

**Where** - GeoJSON
**When** - ObjectID timestamps
**Who** - Meteor Account Strings
**How** - JSON for decision trees

Which leaves the default document in Mongo roughly representing a 'What'.

## Examples

### Creating Records in a Legacy Database

You can default to the normal Mongo format by defining your collections with the idGeneration field.

```
MyCollection = new Meteor.Collection('mycollection', {idGeneration : 'MONGO'});
```

### Inserting data into a document

Many beginners to Mongo struggle with basics, such as how to insert an array, date, boolean, session variable, and so forth into a document record. This example provides some guidance on basic data inputs.

```
Todos.insert({
  text: "foo",                        // String
  listId: Session.get('list_id'),     // String
  value: parseInt(2),                 // Number
  done: false,                        // Boolean
  createdAt: new Date(),              // Dimestamp
  timestamp: (new Date()).getTime(),  // Time
  tags: []                            // Array
});
```

### Getting the _id of the most recently created document

You can get it either synchronously:

```
var docId = Todos.insert({text: 'foo'});
console.log(docId);
```

Or asynchronously:

```
Todos.insert({text: 'foo'}, function(error, docId){
  console.log(docId);
});
```

## Timeseries Data

Using MongoDB for time series data is a very well document and established use-case, with official whitepapers and presentations. Read and watch the official documentation from MongoDB before trying to invent your own schemas for time series data.

MongoDB for Time Series Data

In general, you'll want to create "buckets" for your timeseries data:

```
DailyStats.insert({
   "date" : moment().format("MM-DD-YYYY"),
   "dateIncrement" : moment().format("YYYYMMDD"),
   "dailyTotal" : 0,
   'bucketA': 0,
   'bucketB': 0,
   'bucketC': 0
   });
```

And then increment those buckets as data feeds into your application. This increment can be put in a Meteor Method, a collection observer, a REST API endpoint, and various other places.

```
DailyStats.update({_id: doc._id}, {$inc: {bucketA: 1} });
```

For a more complete Meteor example, see the examples from the Clinical Meteor track:

Realtime Analytics Pipeline
Clinical Meteor - Graphs - Dailystats

## Filtering with Regexes

Simple pattern for filtering subscriptions on the server, using regexes, reactive session variables, and deps autoruns.

```
// create our collection
WordList =  new Meteor.Collection("wordlist");

// and a default session variable to hold the value we're searching for
Session.setDefault('dictionary_search', '');

Meteor.isClient(function(){
    // we create a reactive context that will rerun items when a Session variable gets updated

    Deps.autorun(function(){
        // and create a subscription that will get re-subscribe to when Session variable gets
updated
```

```
        Meteor.subscribe('wordlist', Session.get('dictionary_search'));
    });

    Template.dictionaryIndexTemplate.events({
        'keyup #dictionarySearchInput': function(evt,tmpl){
            // we set the Session variable with the value of our input when it changes
            Session.set('dictionary_search', $('#dictionarySearchInput').val());
        },
        'click #dictionarySearchInput':function(){
            // and clear the session variable when we enter the input
            Session.set('dictionary_search', '');
        },
    });
});
Meteor.isServer(function(){
    Meteor.publish('wordlist', function (word_search) {
        // this query gets rerun whenever the client subscribes to this publication
        return WordList.find({
            // and here we do our regex search
            Word: { $regex: word_search, $options: 'i' }
        },{limit: 100});
    });
});
```

And the HTML that is used on the client:

```
<input id="dictionarySearchInput" type="text" placeholder="Filter..." value="hello"></input>
```

This pattern itself is pretty straight forward, but the regexes may not be. If you're not familiar with regexes, here are some useful tutorials and links:

Regular Expression Tutorial
Regular Expression Cheat Sheet
Regular Expressions in Javascript

## Geospatial Collections - Learning More

Geospatial collections generally involve storing GeoJSON in the Mongo database, streaming that data to the client, accessing the browser's `window.navigator.geolocation`, loading up a Map API, converting GeoJSON to LatLngs, and plotting on the map. Preferably all in realtime. Here are a list of resources to get you started:

- mongodb optimally stores it's data in geoJSON
- geojson.org
- window.navigator.geolocation
- HTML geolocation
- Maps API picker
- google.maps.LatLng
- Google map.data.loadGeoJson
- meteor-cordova-geolocation-background
- phonegap-googlemaps-plugin
- LatLng

- maps.documentation
- google.maps.LatLng
- 2dsphere indexes
- create a 2dsphere index
- query a 2dsphere index
- geospatial indexes and queries

## Auditing Collection Queries

The following example will log all of your collection queries to the server console in realtime.

```
Meteor.startup(
  function () {
    var wrappedFind = Meteor.Collection.prototype.find;

    // console.log('[startup] wrapping Collection.find')

    Meteor.Collection.prototype.find = function () {
      // console.log(this._name + '.find', JSON.stringify(arguments))
      return wrappedFind.apply(this, arguments);
    }
  },

  function () {
    var wrappedUpdate = Meteor.Collection.prototype.update;

    // console.log('[startup] wrapping Collection.find')

    Meteor.Collection.prototype.update = function () {
      console.log(this._name + '.update', JSON.stringify(arguments))
      return wrappedUpdate.apply(this, arguments);
    }
  }
);
```

## Observers & Worker Functions

If the Node event loop acts like a bicycle chain, the server-side collection observer is like a derailleur. It's a gearing mechanism that is going to sit on the data collection as the data comes in. It can be very performant, as all race bicycles have derailleurs. But it's also a source for breaking the whole system. It's a high speed reactive function, which can blow up on you. Be warned.

```
Meteor.startup(function(){
  console.log('starting worker....');

  var dataCursor = Posts.find({viewsCount: {$exists: true}},{limit:20});

  var handle = dataCursor.observeChanges({
    added: function (id, record) {
      if(record.viewsCount > 10){
         // run some statistics
         calculateStatistics();

         // or update a value
         Posts.update({_id: id}, {$set:{
           popular: true
         }});

      }
    },
    removed: function () {
      console.log("Lost one.");
    }
  });
});
```

Note the limit of 20 is the size of the derailleur.... how many teeth it has; or, more specifically, how many items are in the cursor as it's walking over the collection. Be careful about using the 'var' keyword in this kind of function. Write as few objects to memory as possibly, and focus on object reuse inside the added method. When the opslog is turned on, and this thing is going full speed, it's a prime candidate for exposing nasty memory leaks if it's writing down objects onto the memory heap faster than the Node garbage collector is able to clean things up.

The above solution won't scale horizontally well, because each Meteor instance will be trying to update the same record. So, some sort of environment detection is necessary for this to scale horizontally.

See the `percolatestudios:synced-cron` package for an excellent example of synchronizing service workers across multiple machines in a cluster.
meteor-synced-cron

Read Mongo Collections online: https://riptutorial.com/meteor/topic/5120/mongo-collections

# Chapter 31: Mongo Database Management

## Remarks

If you're not opposed to running a Ruby utility, Genghis is a classic: http://genghisapp.com/

But for scalable production use, get yourself to MongoHQ.
http://www.mongohq.com/

Also, the Mongo Monitoring Service, from 10Gen, the makers of Mongo:
https://mms.mongodb.com/

MongoClient is written in Meteor, Completely Free, Open Source And Cross-Platform.

RoboMongo Native cross-platform MongoDB management tool

## Examples

### Analyzing An Inherited Database

There's two great utilities for black-box analysis of databases. First is variety.js, which will give you a high-level overview. The second is schema.js, which will let you dig into the collections for more detail on the individual fields. When inheriting a production Mongo database, these two utilities can help you make sense of what's going on and how the collections and documents are structured.

variety.js

```
mongo test --eval "var collection = 'users'" variety.js
```

schema.js

```
mongo --shell schema.js
```

### Connect To A Database on *.meteorapp.com

The --url flag can be tricky to use. There is a 60 second window to authenticate, and then the username/password randomly resets. So be sure to have robomongo open and ready to configure a new connection when you run the command.

```
# get the MONGO_URL string for your app
meteor mongo --url $METEOR_APP_URL
```

### Download a Database from *.meteor.com

Same thing as before, but you have to copy the info into the mongodump command. You have to run the following commands rediculously fast, and it requires hand/eye coordination. Be warned! This is a rediculously hacky! But fun! Think of it as a video game! :D

```
# get the MONGO_URL string for your app
meteor mongo --url $METEOR_APP_URL

# then quickly copy all the info into the following command
mongodump -u username -p password --port 27017 --db meteor_app_url_com --host production-db-
b1.meteor.io
```

## Export Data from local Meteor development instance?

This command will create a /dump directory, and store each collection in a separate BSON blob file. This is the best way to backup or transfer databases between systems.

```
mongodump --db meteor
```

## Restore Data from a Dumpfile

The analog to the `meteordump` command is `meteorrestore`. You can do a partial import by selecting the specific collection to import. Particularly useful after running a drop command.

```
# make sure your app is running
meteor

# then import your data
mongorestore --port 3001 --db meteor /path/to/dump

# a partial import after running > db.comments.drop()
mongorestore --port 3001 --db meteor /path/to/dump -c comments.bson
```

## Export a Collection to JSON

Run meteor, open another terminal window, and run the following command.

```
mongoexport --db meteor --collection foo --port 3001 --out foo.json
```

## Import a JSON File into Meteor

Importing into a default Meteor instance is fairly easy. Note that you can add a --jsonArray option if your json file is exported as an array from another system.

```
mongoimport --db meteor --port 3001 --collection foo --file foo.json
```

## Copying Data Between Staging and Local Databases

Mongo supports database-to-database copying, which is useful if you have large databases on a staging database that you want to copy into a local development instance.

```
// run mongod so we can create a staging database
// note that this is a separate instance from the meteor mongo and minimongo instances
mongod

// import the json data into a staging database
// jsonArray is a useful command, particularly if you're migrating from SQL
mongoimport -d staging -c assets < data.json --jsonArray

// navigate to your application
cd myappdir

// run meteor and initiate it's database
meteor

// connect to the meteor mongodb
meteor mongo --port 3002

// copy collections from staging database into meteor database
db.copyDatabase('staging', 'meteor', 'localhost');
```

## Compact a Mongo Database on an Ubuntu Box

Preallocation. Mongo sets aside disk-space in empty containers, so when the time comes to write something to disk, it doesn't have to shuffle bits out of the way first. It does so by a doubling algorithm, always doubling the amount of disk space preallocated until it reaches 2GB; and then each prealloc file from thereon is 2GB. Once data is preallocated, it doesn't unallocate unless you specifically tell it to. So observable MongoDB space usage tends to go up automatically, but not down.

Some research on the Mongo preallocation...
reducing-mongodb-database-file-size
mongo-prealloc-files-taking-up-room

```
// compact the database from within the Mongo shell
db.runCommand( { compact : 'mycollectionname' } )

// repair the database from the command line
mongod --config /usr/local/etc/mongod.conf --repair --repairpath /Volumes/X/mongo_repair --
nojournal

// or dump and re-import from the command line
mongodump -d databasename
echo 'db.dropDatabase()' | mongo databasename
mongorestore dump/databasename
```

## Reset a Replica Set

Delete the local database files. Just exit the Mongo shell, navigate to the /dbpath (wherever you set it up), and delete the files within that directory.

## Connect Remotely to a Mongo Instance on *.meteor.com

Did you know about the `--url` flag? Very handy.

```
meteor mongo --url YOURSITE.meteor.com
```

## Accessing Mongo Log Files on a Local Meteor Instance

They're not easily accessible. If you run the 'meteor bundle' command, you can generate a tar.gz file, and then run your app manually. Doing that, you should be able to access the mongo logs... probably in the .meteor/db directory. If you really need to access mongodb log files, set up a regular mongodb instance, and then connect Meteor to an external mongo instance, by setting the MONGO_URL environment variable:

```
MONGO_URL='mongodb://user:password@host:port/databasename'
```

Once that's done, you should be able to access logs in the usual places...

```
/var/log/mongodb/server1.log
```

## Rotate Log Files on an Ubuntu Box

Gotta rotate those log files, or they'll eventually eat up all of your disk space. Start with some research...
mongodb-log-file-growth
rotate-log-files

Log files can be viewed with the following command...

```
ls /var/log/mongodb/
```

But to set up log-file rotation, you'll need to do the following...

```
// put the following in the /etc/logrotate.d/mongod file
/var/log/mongo/*.log {
    daily
    rotate 30
    compress
    dateext
    missingok
    notifempty
    sharedscripts
    copytruncate
    postrotate
        /bin/kill -SIGUSR1 `cat /var/lib/mongo/mongod.lock 2> /dev/null` 2> /dev/null || true
    endscript
}

// to manually initiate a log file rotation, run from the Mongo shell
use admin
db.runCommand( { logRotate : 1 } )
```

Read Mongo Database Management online: https://riptutorial.com/meteor/topic/3707/mongo-database-management

# Chapter 32: Mongo Schema Migrations

## Remarks

It's often necessary to run maintenance scripts on your database. Fields get renamed; data structures get changed; features that you used to support get removed; services get migrated. The list of reasons why you might want to change your schema is pretty limitless. So, the 'why' is pretty self explanatory.

The 'how' is a little more unfamiliar. For those people accustomed to SQL functions, the above database scripts will look strange. But notice how they're all in javascript, and how they're using the same API as we use throughout Meteor, on both the server and client. We have a consistent API through our database, server, and client.

Run the schema migration commands from the meteor mongo shell:

```
# run meteor
meteor

# access the database shell in a second terminal window
meteor mongo
```

## Examples

### Add Version Field To All Records in a Collection

```
db.posts.find().forEach(function(doc){
    db.posts.update({_id: doc._id}, {$set:{'version':'v1.0'}}, false, true);
});
```

### Remove Array From All Records In A Collection

```
db.posts.find().forEach(function(doc){
    if(doc.arrayOfObjects){
        // the false, true at the end refers to $upsert, and $multi, respectively
        db.accounts.update({_id: doc._id}, {$unset: {'arrayOfObjects': "" }}, false, true);
    }
});
```

### Rename Collection

```
db.originalName.renameCollection("newName" );
```

### Find Field Containing Specific String

With the power of regex comes great responsibility....

```
db.posts.find({'text': /.*foo.*|.*bar.*/i})
```

## Create New Field From Old

```
db.posts.find().forEach(function(doc){
    if(doc.oldField){
        db.posts.update({_id: doc._id}, {$set:{'newField':doc.oldField}}, false, true);
    }
});
```

## Pull Objects Out of an Array and Place in a New Field

```
db.posts.find().forEach(function(doc){
    if(doc.commenters){
        var firstCommenter = db.users.findOne({'_id': doc.commenters[0]._id });
        db.clients.update({_id: doc._id}, {$set:{'firstPost': firstCommenter }}, false, true);

        var firstCommenter = db.users.findOne({'_id': doc.commenters[doc.commenters.length -
1]._id });
        db.clients.update({_id: doc._id}, {$set:{'lastPost': object._id }}, false, true);
    }
});
```

## Blob Record From One Collection Into Another Collection (ie. Remove Join & Flatten)

```
db.posts.find().forEach(function(doc){
    if(doc.commentsBlobId){
        var commentsBlob = db.comments.findOne({'_id': commentsBlobId });
        db.posts.update({_id: doc._id}, {$set:{'comments': commentsBlob }}, false, true);
    }
});
```

## Make Sure Field Exists

```
db.posts.find().forEach(function(doc){
    if(!doc.foo){
        db.posts.update({_id: doc._id}, {$set:{'foo':''}}, false, true);
    }
});
```

## Make Sure Field has Specific Value

```
db.posts.find().forEach(function(doc){
    if(!doc.foo){
        db.posts.update({_id: doc._id}, {$set:{'foo':'bar'}}, false, true);
    }
});
```

## Remove Record if Specific Field is Specific Value

```
db.posts.find().forEach(function(doc){
    if(doc.foo === 'bar'){
        db.posts.remove({_id: doc._id});
    }
});
```

## Change Specific Value of Field to New Value

```
db.posts.find().forEach(function(doc){
    if(doc.foo === 'bar'){
        db.posts.update({_id: doc._id}, {$set:{'foo':'squee'}}, false, true);
    }
});
```

## Unset Specific Field to Null

```
db.posts.find().forEach(function(doc){
    if(doc.oldfield){
        // the false, true at the end refers to $upsert, and $multi, respectively
        db.accounts.update({_id: doc._id}, {$unset: {'oldfield': "" }}, false, true);
    }
});
```

## Convert ObjectId to String

```
db.posts.find().forEach(function(doc){
     db.accounts.update({_id: doc._id}, {$set: {'_id': doc._id.str }}, false, true);
});
```

## Convert Field Values from Numbers to Strings

```
var newvalue = "";
db.posts.find().forEach(function(doc){
     if(doc.foo){
         newvalue = '"' + doc.foo + '"';
         db.accounts.update({_id: doc._id}, {$set: {'doc.foo': newvalue}});
     }
});
```

## Convert Field Values from Strings to Numbers

```
var newvalue = null;
db.posts.find().forEach(function(doc){
     if(doc.foo){
         newvalue = '"' + doc.foo + '"';
         db.accounts.update({_id: doc._id}, {$set: {'doc.foo': newvalue}});
     }
});
```

## Create a Timestamp from an ObjectID in the _id Field

```
db.posts.find().forEach(function(doc){
    if(doc._id){
        db.posts.update({_id: doc._id}, {$set:{ timestamp: new
Date(parseInt(doc._id.str.slice(0,8), 16) *1000) }}, false, true);
    }
});
```

### Create an ObjectID from a Date Object

```
var timestamp = Math.floor(new Date(1974, 6, 25).getTime() / 1000);
var hex       = ('00000000' + timestamp.toString(16)).substr(-8); // zero padding
var objectId  = new ObjectId(hex + new ObjectId().str.substring(8));
```

### Find All the Records that Have Items in an Array

What we're doing here is referencing the array index using dot notation

```
db.posts.find({"tags.0": {$exists: true }})
```

Read Mongo Schema Migrations online: https://riptutorial.com/meteor/topic/3708/mongo-schema-migrations

# Chapter 33: MongoDB

## Introduction

MongoDB is a free and open-source cross-platform document orient database program. Unlike classic SQL databases, MongoDB uses BSON (like JSON) to store data. Meteor was designed to use MongoDB for database storage and this topic explains how to implement MongoDB storage into Meteor applications.

## Examples

### Export a Remote Mongo DB, Import Into a Local Meteor Mongo DB

Helpful when you want to grab a copy of a production database to play around with locally.

1. `mongodump --host some-mongo-host.com:1234 -d DATABASE_NAME -u DATABASE_USER -p DATABASE_PASSWORD` This will create a local `dump` directory; within that directory you'll see a directory with your `DATABASE_NAME`.
2. With your local meteor app running, from within the `dump` directory, run: `mongorestore --db meteor --drop -h localhost --port 3001 DATABASE_NAME`

### Get the Mongo URL of Your Local Meteor Mongo DB

While your Meteor app is running locally:

```
meteor mongo --url
```

### Connect Your Local Meteor App to an Alternative Mongo DB

Set the `MONGO_URL` environment variable before starting your local Meteor app.

## Linux/MacOS Example:

```
MONGO_URL="mongodb://some-mongo-host.com:1234/mydatabase" meteor
```

or

```
export MONGO_URL="mongodb://some-mongo-host.com:1234/mydatabase"
meteor
```

## Windows Example

Note: don't use `"`

```
set MONGO_URL=mongodb://some-mongo-host.com:1234/mydatabase
meteor
```

## NPM

```
//package.json

"scripts": {
    "start": "MONGO_URL=mongodb://some-mongo-host.com:1234/mydatabase meteor"
}

$ npm start
```

### Running Meteor without MongoDB

Set `MONGO_URL` to any arbitrary value except for a database URL and ensure no collections are defined in your Meteor project (including collections defined by Meteor packages) to run Meteor without MongoDB.

Note that without MongoDB, server/client methods alongside any packages related to Meteor's user-account system will be undefined. Ex: `Meteor.userId()`

**Linux/Mac:**

```
MONGO_URL="none" meteor
```

*or*

```
export MONGO_URL="none"
meteor
```

**Windows:**

```
set MONGO_URL=none
meteor
```

### Getting Started

You can start the **`mongo`** shell by running the following command inside your Meteor project:

```
meteor mongo
```

**Please note:** Starting the server-side database console only works while Meteor is running the application locally.

After that, you can list all collections by executing the following command via the **`mongo`** shell:

```
show collections
```

You can also run basic MongoDB operations, like querying, inserting, updating and deleting documents.

---

## Query Documents

Documents can be queried by using the `find()` method, e.g.:

```
db.collection.find({name: 'Matthias Eckhart'});
```

This will list all documents that have the `name` attribute set to `Matthias Eckhart`.

## Inserting Documents

If you want to insert documents in a collection, run:

```
db.collection.insert({name: 'Matthias Eckhart'});
```

## Updating Documents

In case you want to update documents, use the `update()` method, for instance:

```
db.collection.update({name: 'Matthias Eckhart'}, {$set: {name: 'John Doe'}});
```

Executing this command will update a **single** document by setting the value `John Doe` for the field `name` (initially the value was `Matthias Eckhart`).

If you want to update **all** documents that match a specific criteria, set the `multi` parameter to `true`, for example:

```
db.collection.update({name: 'Matthias Eckhart'}, {$set: {name: 'John Doe'}}, {multi: true});
```

Now, all documents in the collection that had initially the `name` attribute set to `Matthias Eckhart` have been updated to `John Doe`.

## Deleting Documents

Documents can be easily removed by using the `remove()` method, for example:

```
db.collection.remove({name: 'Matthias Eckhart'});
```

This will remove all documents that match the value specified in the `name` field.

Read MongoDB online: https://riptutorial.com/meteor/topic/1874/mongodb

# Chapter 34: MongoDB Aggregation

## Remarks

**Server Aggregation**

Average Aggregation Queries in Meteor

is it possible to package a 'real' mongodb library for use on the *server* side only in meteor 0.6

**Client Aggregation (Minimongo)**

https://github.com/utunga/pocketmeteor/tree/master/packages/mongowrapper

## Examples

### Server Aggregation

Andrew Mao's solution. Average Aggregation Queries in Meteor

```
Meteor.publish("someAggregation", function (args) {
    var sub = this;
    // This works for Meteor 0.6.5
    var db = MongoInternals.defaultRemoteCollectionDriver().mongo.db;

    // Your arguments to Mongo's aggregation. Make these however you want.
    var pipeline = [
        { $match: doSomethingWith(args) },
        { $group: {
            _id: whatWeAreGroupingWith(args),
            count: { $sum: 1 }
        }}
    ];

    db.collection("server_collection_name").aggregate(
        pipeline,
        // Need to wrap the callback so it gets called in a Fiber.
        Meteor.bindEnvironment(
            function(err, result) {
                // Add each of the results to the subscription.
                _.each(result, function(e) {
                    // Generate a random disposable id for aggregated documents
                    sub.added("client_collection_name", Random.id(), {
                        key: e._id.somethingOfInterest,
                        count: e.count
                    });
                });
                sub.ready();
            },
            function(error) {
                Meteor._debug( "Error doing aggregation: " + error);
            }
        )
    );
});
```

## Aggregation in a Server Method

Another way of doing aggregations is by using the `Mongo.Collection#rawCollection()`

This can only be run on the Server.

Here is an example you can use in Meteor 1.3 and higher:

```
Meteor.methods({
   'aggregateUsers'(someId) {
      const collection = MyCollection.rawCollection()
      const aggregate = Meteor.wrapAsync(collection.aggregate, collection)

      const match = { age: { $gte: 25 } }
      const group = { _id:'$age', totalUsers: { $sum: 1 } }

      const results = aggregate([
         { $match: match },
         { $group: group }
      ])

      return results
   }
})
```

Read MongoDB Aggregation online: https://riptutorial.com/meteor/topic/4199/mongodb-aggregation

# Chapter 35: Nightwatch - Configuration & Setup

## Remarks

Nightwatch has been providing Acceptance and End-to-End testing for Meteor apps since v0.5 days, and has managed migrations from PHP to Spark to Blaze and to React; and all major Continuous Integration platforms. For additional help, please see:

Nightwatch API Documentation
Nightwatch.js Google Group

## Examples

### Configuration

The main reason that Nightwatch is so powerful, is because of it's excellent configuration file. Unlike most other testing frameworks, Nightwatch is fully configurable and customizable to different environments and technology stacks.

**.meteor/nightwatch.json**

The following configuration file is for Meteor v1.3 and later, and supports two environments... a `default` environment which launches the chromedriver browser, and a `phantom` environment which runs the tests in a headless environment.

```
{
  "nightwatch": {
    "version": "0.9.8"
  },
  "src_folders": [
    "./tests/nightwatch/walkthroughs"
  ],
  "custom_commands_path": [
    "./tests/nightwatch/commands"
  ],
  "custom_assertions_path": [
    "./tests/nightwatch/assertions"
  ],
  "output_folder": "./tests/nightwatch/reports",
  "page_objects_path": "./tests/nightwatch/pages",
  "globals_path": "./tests/nightwatch/globals.json",
  "selenium": {
    "start_process": true,
    "server_path": "./node_modules/starrynight/node_modules/selenium-server-standalone-
jar/jar/selenium-server-standalone-2.45.0.jar",
    "log_path": "tests/nightwatch/logs",
    "host": "127.0.0.1",
    "port": 4444,
    "cli_args": {
      "webdriver.chrome.driver":
"./node_modules/starrynight/node_modules/chromedriver/bin/chromedriver"
    }
  },
  "test_settings": {
    "default": {
      "launch_url": "http://localhost:5000",
      "selenium_host": "127.0.0.1",
      "selenium_port": 4444,
      "pathname": "/wd/hub",
      "silent": true,
      "disable_colors": false,
      "firefox_profile": false,
      "ie_driver": "",
      "screenshots": {
        "enabled": false,
        "path": "./tests/nightwatch/screenshots"
      },
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true,
        "loggingPrefs": {
          "browser": "ALL"
        }
      },
      "exclude": "./tests/nightwatch/unittests/*",
      "persist_globals": true,
      "detailed_output": false
    },
    "phantom": {
      "desiredCapabilities": {
        "browserName": "phantomjs",
        "javascriptEnabled": true,
        "databaseEnabled": false,
        "locationContextEnabled": false,
        "applicationCacheEnabled": false,
        "browserConnectionEnabled": false,
        "webStorageEnabled": false,
        "acceptSslCerts": true,
        "rotatable": false,
        "nativeEvents": false,
        "phantomjs.binary.path": "./node_modules/starrynight/node_modules/phantomjs-
prebuilt/bin/phantomjs"
      }
    },
    "unittests": {
      "selenium": {
        "start_process": false,
        "start_session": false
      },
      "filter": "./tests/nightwatch/unittests/*",
      "exclude": ""
    }
  }
}
```

## Installation & Usage

To get Nightwatch working, you'll need a local copy of **selenium** which is a command-and-control server which manages automated browser instances. You'll also need a web browser which selenium can control, such as **chromedriver** or **phantomjs**.

Add the following devDependencies to your `package.json`:

```
{
  "devDependencies": {
    "nightwatch": "0.9.8",
    "selenium-server-standalone-jar": "2.45.0",
    "chromedriver": "2.19.0",
    "phantomjs-prebuilt": "2.1.12"
  }
}
```

Then install all the depndencies.

```
cd myapp
meteor npm install
```

You should then be able to run Nightwatch with the following commands:

```
nightwatch -c .meteor/nightwatch.json
nightwatch -c .meteor/nightwatch.json --env phantom
```

If you haven't written any tests, or set up your folder structure yet, you may get some errors.

## Setting up launch scripts

In the root of your application should be a `package.json` file, where you can define scripts and devDependencies.

```
{
  "name": "myapp",
  "version": "1.0.0",
  "scripts": {
    "start": "meteor --settings settings-development.json",
    "nightwatch": "nightwatch -c .meteor/nightwatch.json",
    "phantom": "nightwatch -c .meteor/nightwatch.json --env phantom",
  }
}
```

You will then be able to launch nightwatch with the following commands:

```
meteor npm run-script nightwatch
meteor npm run-script phantom
```

In this example, it would almost be easier to simply run `nightwatch -c .meteor/nightwatch.json`. However, with more complex commands, with complex environment variables, options, and settings, this becomes a very useful way to specify devops scripts for a team.

## Folder Structure

A basic Nightwatch installation for Meteor will have the following directories and files installed.

```
/myapp
/myapp/.meteor/nightwatch.json
/client/main.html
/client/main.js
/client/main.css
/tests
/tests/nightwatch
/tests/nightwatch/assertions
/tests/nightwatch/commands
/tests/nightwatch/data
/tests/nightwatch/logs
/tests/nightwatch/pages
/tests/nightwatch/reports
/tests/nightwatch/screenshots
/tests/nightwatch/walkthroughs
/tests/nightwatch/walkthroughs/critical_path.js
/tests/nightwatch/globals.json
```

## Data Driven Testing

Nightwatch accepts a second `globals.json` configuration file which injects data into the test runner itself, very similar to how `Meteor.settings` makes data from the command line available throughout the app.

### globals.json

```
{
  "default" : {
    "url" : "http://localhost:3000",
    "user": {
      "name": "Jane Doe",
      "username" : "janedoe",
      "password" : "janedoe123",
      "email" : "janedoe@test.org",
      "userId": null
    }
  },
  "circle" : {
    "url" : "http://localhost:3000",
    "user": {
      "name": "Jane Doe",
      "username" : "janedoe",
      "password" : "janedoe123",
      "email" : "janedoe@test.org"
      "userId": null
    }
  },
  "galaxy" : {
    "url" : "http://myapp.meteorapp.com",
    "user": {
      "name": "Jane Doe",
      "username" : "janedoe",
      "password" : "janedoe123",
      "email" : "janedoe@test.org"
      "userId": null
    }
  }
}
```

You can then write your tests that aren't hardcoded with specific users, passwords, search inputs, etc.

```
module.exports = {
  "Login App" : function (client) {
    client
      .url(client.globals.url)
      .login(client.globals.user.email, client.globals.user.password)
      .end();
  }
};
```

Read Nightwatch - Configuration & Setup online: https://riptutorial.com/meteor/topic/5901/nightwatch---configuration---setup

# Chapter 36: Node/NPM

## Examples

### Meteor Tested/Supported Node Version

To determine the latest tested/supported version of Node that can be used with your installed version of Meteor, dump the node version directly from the build tool's bundled node instance.

```
meteor node -v
```

Read Node/NPM online: https://riptutorial.com/meteor/topic/4599/node-npm

# Chapter 37: Offline Apps

## Remarks

Further Appcache Research

http://www.html5rocks.com/en/tutorials/indexeddb/todo/
http://grinninggecko.com/2011/04/22/increasing-chromes-offline-application-cache-storage-limit/
http://www.html5rocks.com/en/tutorials/offline/quota-research/
https://developers.google.com/chrome/apps/docs/developers_guide?csw=1#installing
https://developers.google.com/chrome/apps/docs/developers_guide?csw=1#manifest

## Examples

### Meteor.status()

The first thing to do when taking your Meteor app offline is to create some visual indication of whether the local client app is connected to the server or not. There are lots of ways to do this, but the simplest way is to probably do something like this:

```
Template.registerHelper('getOnlineStatus', function(){
  return Meteor.status().status;
});

Template.registerHelper('getOnlineColor', function(){
  if(Meteor.status().status === "connected"){
    return "green";
  }else{
    return "orange";
  }
});
```

```
  <div id="onlineStatus" class="{{getOnlineColor}}">
    {{getOnlineStatus}}
  </div>
```

```
.green{
  color: green;
}
.orange{
  color: orange;
}
```

### Enable Appcache

One of the easier steps is adding the appcache. Appcache will allow your application content to load even when there is no internet access. You won't be able to get any data from your mongo servers, but the static content and assets will be available offline.

```
meteor add appcache
```

## Enable GroundDB

Finally, we want to get some of our dynamic data to be stored offline.

```
meteor add ground:db
```

```
Lists = new Meteor.Collection("lists");
GroundDB(Lists);

Todos = new Meteor.Collection("todos")
GroundDB(Todos);
```

## Things to Be Careful Of

- The appcache will cause some confusion in your development workflow, because it hides Meteor's auto-updating features. When you turn off the server component of your app, the client portion in your browser will continue working. This is a good thing! But, you don't get the immediate feedback that your app has been turned off, or that there have been updates.
- Try using Chrome's Incognito Mode when developing your app, because it doesn't use appcache.
- GroundDB doesn't work particularly well with IronRouter.

Read Offline Apps online: https://riptutorial.com/meteor/topic/3375/offline-apps

# Chapter 38: Performance Tuning

## Remarks

It should be noted that Meteor is simply Javascript and Node.js. Yes, it's a very specific implementation of those two technologies, and has it's own unique ecosystem, and leverages isomorphic APIs and a JSON datastore to achieve some truly amazing results. But, at the end of the day, Meteor is a web technology, and it's written in Javascript. So all of your typical javascript performance techniques still apply. Start there.

25 Javascript Performance Techniques
Performance Optimizations for High Speed Javascript
Optimizing JavaScript Code
Performance Tips for JavaScript in V8
10 Javascript Performance Boosting Tip
Write Efficient JavaScript
Improving the Performance of your Meteor JS projects

## Examples

### Designing and Deploying Production Ready Software

Remember, all the best practices of typical web architecture still apply. For an excellent overview on the subject, please refer to Michael Nygard's excellent book Release It! Design and Deploy Production-Ready Software. Writing your app in Meteor doesn't absolve you of auditing third party libraries, writing circuit breakers, wrapping calls in timeouts, monitoring your resource pools, and all the rest. If you want your application to perform well, you need to make sure you're using stability patterns, and avoiding anti-patterns.

**Stability Patterns**

- Timeouts
- Circuit Breakers
- Bulkheads
- Handshaking
- Stability Anti-Patterns

**Integration Points**

- Third Party Libraries
- Scaling Effects
- Unbalanced Capabilities
- Capacity Anti-Patterns

**Resource Pool Contention**

- AJAX Overkill
- Overstaying Sessions
- Excessive White Space
- Data Eutropification

If these concepts are unfamiliar and don't seem like second-nature to you, it means that you haven't brought bigger production systems online. Buy a copy of the book. It will be time and money well spent.

Read Performance Tuning online: https://riptutorial.com/meteor/topic/3363/performance-tuning

# Chapter 39: Publishing A Release Track

## Remarks

Publishing a Release Track is actually pretty simple if you understand a) that the publish-release command requires a .json file as a parameter, and b) what that file looks like. That's definitely the biggest hurdle in getting started, because it's pretty much not documented anywhere.

Just keep in mind that every package in the release has to be published and on Atmosphere. The .meteor/versions file of an app is a particularly good place for finding all the necessary packages and versions that should go into the release.

After that, it's a matter of figuring out what you're willing to support, what you want to include, etc. Here is a partial Venn Diagram of what the Clinical Release is currently working on; and should give you a general idea of how we're going about the decision making process of what gets included.

For more discussion, see the topic on the Meteor Forums:
https://forums.meteor.com/t/custom-meteor-release/13736/6

## Examples

### Basic Usage

The idea is that a distro maintainer wants to run something like the following command:

```
meteor publish-release clinical.meteor.rc6.json
```

Which will then allow users of the distro to run this:

```
meteor run --release clinical:METEOR@1.1.3-rc6
```

### Release Manifest

A release manifest is similar to an NPM `package.json` file, in that it's primary concern is specify a list of Atmosphere packages, and providing a bit of metadata about that list of packages. The basic format looks like this:

```
{
  "track":"distroname:METEOR",
  "version":"x.y.z",
  "recommended": false,
  "tool": "distroname:meteor-tool@x.y.z",
  "description": "Description of the Distro",
  "packages": {
    "accounts-base":"1.2.0",
    "accounts-password":"1.1.1",
    ...
  }
}
```

## Customizing the Meteor Tool

If you need to extend the meteor tool or the command line, you'll need to create and publish your own meteor-tool package. Ronen's documentation is the best out there for this process:

http://practicalmeteor.com/using-meteor-publish-release-to-extend-the-meteor-command-line-tool/1

It's easy to get a meteor helloworld command working, but after that, I felt it was easier to just create a separate node app to test out commands. Which is how StarryNight came about. It's something of a staging ground and scratchpad for commands before trying to put them into a version of the meteor-tool.

## Extracting a Release Manifest from .meteor/versions

StarryNight contains a small utility that parses an application's `.meteor/versions` file, and converts it into a Release Manifest.

```
npm install -g starrynight
cd myapp
starrynight generate-release-json
```

If you don't wish to use StarryNight, simply copy the contents of your `.meteor/versions` file into the `packages` field of your manifest file. Be sure to convert to JSON syntax and add colons and quotes.

## Displaying the Release Manifest for a Specific Release

```
meteor show --ejson METEOR@1.2.1
```

## Publishing a Release From Checkout

```
meteor publish-release --from-checkout
```

## Fetching the Latest Commits for Each Package in a Release

When building a custom release track, it's common to keep packages in the `/packages` directory as git submodules. The following command allows you to fetch all of the latest commits for the submodules in your `/packages` directory at the same time.

```
git submodule foreach git pull origin master
```

Read Publishing A Release Track online: https://riptutorial.com/meteor/topic/4201/publishing-a-release-track

# Chapter 40: Publishing Data

## Remarks

Within Meteor's data subsystem, a server publication and its corresponding client subscriptions are the main mechanisms of reactive, live data transport where the underlying data is constantly synchronized between the server and the client.

## Examples

### Basic Subscription and Publication

First, remove `autopublish`. `autopublish` automatically publishes the entire database to the client-side, and so the effects of publications and subscriptions cannot be seen.

To remove `autopublish`:

```
$ meteor remove autopublish
```

Then you can create publications. Below is a full example.

```
import { Mongo } from 'meteor/mongo';
import { Meteor } from 'meteor/meteor';

const Todos = new Mongo.Collection('todos');

const TODOS = [
  { title: 'Create documentation' },
  { title: 'Submit to Stack Overflow' }
];

if (Meteor.isServer) {
  Meteor.startup(function () {
    TODOS.forEach(todo => {
      Todos.upsert(
        { title: todo.title },
        { $setOnInsert: todo }
      );
    });
  });

  // first parameter is a name.
  Meteor.publish('todos', function () {
    return Todos.find();
  });
}

if (Meteor.isClient) {
  // subscribe by name to the publication.
  Meteor.startup(function () {
    Meteor.subscribe('todos');
  })
}
```

## Global publications

A global publication does not possess a name and does not require a subscription from the connected client and therefore it is available to the connected client as soon as the client connects to the server.

To achieve this, one simply names the publication as `null` like so

```
Meteor.publish(null, function() {
  return SomeCollection.find();
})
```

## Named publications

A named publication is one that possesses a name and needs to be explicitly subscribed to from the client.

Consider this server side code:

```
Meteor.publish('somePublication', function() {
  return SomeCollection.find()
})
```

The client needs to request it by:

```
Meteor.subscribe('somePublication')
```

## Template scoped subscriptions

Meteor's default templating system Spacebars and its underlying rendering subsystem Blaze integrate seemlessly with publication lifecycle methods such that a simple piece of template code can subscribe to its own data, stop and clean up its own traces during the template tear down.

In order to tap into this, one needs to subscribe on the template instance, rather than the `Meteor` symbol like so:

First set up the template

```
<template name="myTemplate">
  We will use some data from a publication here
</template>
```

Then tap into the corresponding lifecycle callback

```
Template.myTemplate.onCreated(function() {
  const templateInstance = this;
  templateInstance.subscribe('somePublication')
})
```

Now when this template gets destroyed, the publication will also be stopped automatically.

Note: The data that is subscribed to will be available to all templates.

### Publish into an ephemeral client-side named collection.

For if you have to fine-tune what is published.

```
import { Mongo } from 'meteor/mongo';
import { Meteor } from 'meteor/meteor';
import { Random } from 'meteor/random';

if (Meteor.isClient) {
  // established this collection on the client only.
  // a name is required (first parameter) and this is not persisted on the server.
  const Messages = new Mongo.Collection('messages');
  Meteor.startup(function () {
    Meteor.subscribe('messages');
    Messages.find().observe({
      added: function (message) {
        console.log('Received a new message at ' + message.timestamp);
      }
    });
  })
}

if (Meteor.isServer) {
  // this will add a new message every 5 seconds.
  Meteor.publish('messages', function () {
    const interval = Meteor.setInterval(() => {
      this.added('messages', Random.id(), {
        message: '5 seconds have passed',
        timestamp: new Date()
      })
    }, 5000);
    this.added('messages', Random.id(), {
      message: 'First message',
      timestamp: new Date()
    });
    this.onStop(() => Meteor.clearInterval(interval));
  });
}
```

### Creating and responding to an error on a publication.

On the server, you can create a publication like this. `this.userId` is the id of the user who is currently logged in. If no user is logged in, you might want to throw an error and respond to it.

```
import Secrets from '/imports/collections/Secrets';

Meteor.publish('protected_data', function () {
  if (!this.userId) {
    this.error(new Meteor.Error(403, "Not Logged In."));
    this.ready();
  } else {
    return Secrets.find();
  }
});
```

On the client, you can respond with the following.

```
Meteor.subscribe('protected_data', {
  onError(err) {
    if (err.error === 403) {
      alert("Looks like you're not logged in");
    }
  },
});
```

File /imports/collections/Secrets creates reference to the secrets collection as below:

```
const Secrets = new Mongo.Collection('secrets');
```

## Reactively re-subscribing to a publication

A template autorun may be used to (re)subscribe to a publication. It establishes a reactive context which is re-executed whenever any *reactive data it depends on changes*. In addition, an autorun always runs once (the first time it is executed).

Template autoruns are normally put in an `onCreated` method.

```
Template.myTemplate.onCreated(function() {
  this.parameter = new ReactiveVar();
  this.autorun(() => {
    this.subscribe('myPublication', this.parameter.get());
  });
});
```

This will run once (the first time) and set up a subscription. It will then re-run whenever the `parameter` reactive variable changes.

## Wait in the Blaze view while published data is being fetched

Template JS code

```
Template.templateName.onCreated(function(){
    this.subscribe('subsription1');
    this.subscribe('subscription2');
});
```

Template HTML code

```
<template name="templateName">
    {{#if Template.subscriptionsReady }}
        //your actual view with data. it can be plain HTML or another template
    {{else}}
        //you can use any loader or a simple header
        <h2> Please wait ... </h2>
    {{/if}}
</template>
```

## Validating User Account On Publish

Sometimes it's a good idea to further secure your publishes by requiring a user login. Here is how you achieve this via Meteor.

```
import { Recipes } from '../imports/api/recipes.js';
import { Meteor } from 'meteor/meteor';

Meteor.publish('recipes', function() {
  if(this.userId) {
    return Recipe.find({});
  } else {
    this.ready();  // or: return [];
  }
});
```

## Publish multiple cursors

Multiple database cursors can be published from the same publication method by returning an array of cursors.

The "children" cursors will be treated as joins and will not be reactive.

```
Meteor.publish('USER_THREAD', function(postId) {
  let userId   = this.userId;

  let comments = Comments.find({ userId, postId });
  let replies  = Replies.find({ userId, postId });

  return [comments, replies];
});
```

## Simulate delay in publications

In real world, connection and server delays could occur, to simulate delays in development environment `Meteor._sleepForMs(ms);` could be used

```
Meteor.publish('USER_DATA', function() {
    Meteor._sleepForMs(3000); // Simulate 3 seconds delay
    return Meteor.users.find({});
});
```

## Merging Publications

Publications can be merged on the client, resulting in differently shaped documents within a single cursor. The following example represents how a user directory might publish a minimal amount of public data for users of an app, and provide a more detailed profile for the logged in user.

```
// client/subscriptions.js
Meteor.subscribe('usersDirectory');
Meteor.subscribe('userProfile', Meteor.userId());

// server/publications.js
// Publish users directory and user profile

Meteor.publish("usersDirectory", function (userId) {
    return Meteor.users.find({}, {fields: {
        '_id': true,
        'username': true,
        'emails': true,
        'emails[0].address': true,

        // available to everybody
        'profile': true,
        'profile.name': true,
        'profile.avatar': true,
        'profile.role': true
    }});
});
Meteor.publish('userProfile', function (userId) {
    return Meteor.users.find({_id: this.userId}, {fields: {
        '_id': true,
        'username': true,
        'emails': true,
        'emails[0].address': true,

        'profile': true,
        'profile.name': true,
        'profile.avatar': true,
        'profile.role': true,

        // privately accessible items, only availble to the user logged in
        'profile.visibility': true,
        'profile.socialsecurity': true,
        'profile.age': true,
        'profile.dateofbirth': true,
        'profile.zip': true,
        'profile.workphone': true,
        'profile.homephone': true,
        'profile.mobilephone': true,
        'profile.applicantType': true
    }});
});
```

Read Publishing Data online: https://riptutorial.com/meteor/topic/1323/publishing-data

# Chapter 41: Reactive (Vars & Dictionaries)

## Examples

### Reactive Query

Example code :

In main.html

```
<template name="test">
     <input type="checkbox" id="checkbox1" name="name" value="data">Check Me
      {{showData}}
</template>
```

In Main.js

```
 var check_status='';
 //Reactive Var Initialization
 Template.main.onCreated(function (){
       check_status=new ReactiveVar({});

 });

 Template.main.helpers({
       showData : function(){
           return Collection.find(check_status.get());
       }
 });

 Template.main.events({
      "change #checkbox1" : function(){
             check_status.set({field: 'data'});
       }
 });
```

Explanation:

When we initialize the reactive var `check_status` we set the value equal to `{}`. In the helper, at the time of rendering, the same data is passed to the query `Collection.find(check_status.get())` which is as good as *show all* data.

As soon as you click on the checkbox, the event described in the `Template.main.events` is triggered which sets the value of `check_status` to `{field: data}`. Since, this is a *reactive var*, the `showData` template is re run and this time the query is `Collection.find({field: data})`, so only fields, where `field` matched `'data'` is returned.

You will need to add the `reactive var` package(`meteor add reactive-var`) before using this commands.

Read Reactive (Vars & Dictionaries) online: https://riptutorial.com/meteor/topic/6535/reactive--

vars---dictionaries-

# Chapter 42: Replica Sets and Sharding

## Remarks

For those not familiar, a Replica Set is defined as a redundant configuration of three servers. A Sharded Database is defined as a horizintally scalled database, where each Shard is defined as a Replica Set. Therefore, a sharded Mongo cluster involves a minimum of 11 servers for a 2 shard cluster, and increases by three servers for each additional shard. So, a sharded cluster always has 11, 14, 17, 20, 23, etc server instances. That is, there's 2 shards of 3 servers each, 3 more config controllers, and 2 routers. 11 servers total for a 2 shard cluster.

## Examples

### Replica Set Quickstart

Build yourself **three** servers using whatever physical or virtual hardware you wish. (This tutorial assumes you're using Ubuntu as your operating system.) Then repeat the following instructions three times... once for each server.

```
# add the names of each server to the host file of each server
sudo nano /etc/hosts
  10.123.10.101 mongo-a
  10.123.10.102 mongo-b
  10.123.10.103 mongo-c

# install mongodb on the server
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee
/etc/apt/sources.list.d/mongodb.list
sudo apt-get update
sudo apt-get install mongodb-10gen


# create the /data/ directories
sudo mkdir /data
sudo mkdir /data/logs
sudo mkdir /data/db

# make sure the mongodb user and group have access to our custom directories
sudo chown -R mongodb:mongodb /data

# edit the mongo upstart file in /etc/init/mongodb.conf
sudo nano /etc/init/mongodb.conf
  start on started mountall
  stop on shutdown
  respawn
  respawn limit 99 5
  setuid mongodb
  setgid mongodb
  script
    exec /usr/bin/mongod --config /etc/mongodb.conf >> /data/logs/mongo-a.log 2>&1
  end script
```

```
# edit mongodb configuration file
sudo nano /etc/mongodb.conf
    dbpath=/data/db
    logpath=/data/logs/mongod.log
    logappend=true
    port=27017
    noauth=true
    replSet=meteor
    fork=true

# add a mongo log-rotation file
sudo nano /etc/logrotate.d/mongod
  /data/logs/*.log {
    daily
    rotate 30
    compress
    dateext
    missingok
    notifempty
    sharedscripts
    copytruncate
    postrotate
        /bin/kill -SIGUSR1 `cat /data/db/mongod.lock 2> /dev/null` 2> /dev/null || true
    endscript
  }

# make sure mongod service is started and running
sudo service mongodb start
sudo reboot
```

## Replica Set Configuration

Then go into the mongo shell and initiate the replica set, like so:

```
meteor mongo

  > rs.initiate()
  PRIMARY> rs.add("mongo-a")
  PRIMARY> rs.add("mongo-b")
  PRIMARY> rs.add("mongo-c")
  PRIMARY> rs.setReadPref('secondaryPreferred')
```

Read Replica Sets and Sharding online: https://riptutorial.com/meteor/topic/4332/replica-sets-and-sharding

# Chapter 43: Retrieving data from a Meteor.call

## Examples

### The basics of Meteor.call

```
Meteor.call(name, [arg1, arg2...], [asyncCallback])
```

(1) name String
(2) Name of method to invoke
(3) arg1, arg2... EJSON-able Object [Optional]
(4) asyncCallback Function [Optional]

On one hand, you can do : (via **Session variable**, or via **ReactiveVar**)

```
    var syncCall = Meteor.call("mymethod") // Sync call
```

It mean if you do something like this, server side you will do :

```
    Meteor.methods({
        mymethod: function() {
            let asyncToSync =  Meteor.wrapAsync(asynchronousCall);
            // do something with the result;
            return  asyncToSync;
        }
    });
```

On the other hand, sometimes you will want to keep it via the result of the callback ?

**Client side :**

```
Meteor.call("mymethod", argumentObjectorString, function (error, result) {
    if (error) Session.set("result", error);
    else Session.set("result",result);
}
Session.get("result") -> will contain the result or the error;

//Session variable come with a tracker that trigger whenever a new value is set to the session
variable. \ same behavior using ReactiveVar
```

**Server side**

```
Meteor.methods({
    mymethod: function(ObjectorString) {
        if (true) {
            return true;
        } else {
            throw new Meteor.Error("TitleOfError", "ReasonAndMessageOfError"); // This will
and up in the error parameter of the Meteor.call
        }
    }
});
```

The purpose here is to show that Meteor propose various way to communicate between the Client and the Server.

### Using Session variable

## Server side

```
Meteor.methods({
  getData() {
    return 'Hello, world!';
  }
});
```

## Client side

```
<template name="someData">
  {{#if someData}}
    <p>{{someData}}</p>
  {{else}}
    <p>Loading...</p>
  {{/if}}
</template>
```

```
Template.someData.onCreated(function() {
  Meteor.call('getData', function(err, res) {
    Session.set('someData', res);
  });
});

Template.someData.helpers({
  someData: function() {
    return Session.get('someData');
  }
});
```

### Using ReactiveVar

## Server side

```
Meteor.methods({
  getData() {
    return 'Hello, world!';
  }
});
```

## Client side

```
<template name="someData">
  {{#if someData}}
    <p>{{someData}}</p>
  {{else}}
    <p>Loading...</p>
  {{/if}}
</template>
```

```
Template.someData.onCreated(function() {

  this.someData = new ReactiveVar();

  Meteor.call('getData', (err, res) => {
    this.someData.set(res);
  });
});

Template.someData.helpers({
  someData: function() {
    return Template.instance().someData.get();
  }
});
```

`reactive-var` package required. To add it run `meteor add reactive-var`.

Read Retrieving data from a Meteor.call online: https://riptutorial.com/meteor/topic/3068/retrieving-data-from-a-meteor-call

# Chapter 44: Routing

## Examples

### Routing with Iron Router

#### Install Iron Router

From the terminal:

```
meteor add iron:router
```

#### Basic configuration

```
Router.configure({
    //Any template in your routes will render to the {{> yield}} you put inside your layout
template
    layoutTemplate: 'layout',
    loadingTemplate: 'loading'
});
```

#### Render without data

```
//this is equal to home page
Router.route('/', function (){
    this.render('home')
});

Router.route('/some-route', function () {
    this.render('template-name');
});
```

#### Render with data and parameters

```
Router.route('/items/:_id', function () {
    this.render('itemPage', {
        data: function() {
            return Items.findOne({_id: this.params._id})
        }
    });
});
```

#### Render to a secondary yield

```
Router.route('/one-route/route', function() {
    //template 'oneTemplate' has {{> yield 'secondary'}} in HTML
    this.render('oneTemplate');

    //this yields to the secondary place
    this.render('anotherTemplate', {
        to: 'secondary'
    });

    //note that you can write a route  for '/one-route'
    //then another for '/one-route/route' which will function exactly like above.
});
```

#### Subscribe and wait for data before rendering template

```
Router.route('/waiting-first', {
    waitOn: function() {
        //subscribes to a publication
        //shows loading template until subscription is ready
        return Meteor.subscribe('somePublication')
    },

    action: function() {
        //render like above examples
    }
});
```

#### Subscribe to multiple publications and wait for data before rendering template

```
Router.route('/waiting-first', {
    waitOn: function() {
        //subscribes to a publication
        //shows loading template until subscription is ready
        return [Meteor.subscribe('somePublication1'),Meteor.subscribe('somePublication2')];
    },

    action: function() {
        //render like above examples
    }
});
```

Guide for Iron Router: http://iron-meteor.github.io/iron-router/

### With FlowRouter

FlowRouter is more modular compared to Iron Router.

## Install FlowRouter

```
meteor add kadira:flow-router
```

## Rendering a template

In particular, you must manually add a layout rendering package to link with your rendering engine:

- Blaze Layout for Blaze: `meteor add kadira:blaze-layout`

- React Layout for React: `meteor add kadira:react-layout`

Then you can render through dynamic templating (in the case of Blaze):

```
<template name="mainLayout">
  {{> Template.dynamic template=area}}
</template>
```

```
FlowRouter.route('/blog/:postId', {
  action: function (params) {
    BlazeLayout.render("mainLayout", {
      area: "blog"
    });
  }
});
```

## Rendering a template with parameters and/or query

The parameters are specified on the route, like with Iron Router:

```
FlowRouter.route("/blog/:catId/:postId", {
  name: "blogPostRoute",
  action: function (params) {
    //...
  }
})
```

But the parameters are not passed as data context to the child template. Instead, the child template must read them:

```
// url: /blog/travel/france?showcomments=yes
var catId = FlowRouter.getParam("catId"); // returns "travel"
var postId = FlowRouter.getParam("postId"); // returns "france"

var color = FlowRouter.getQueryParam("showcomments"); // returns "yes"
```

Read Routing online: https://riptutorial.com/meteor/topic/5119/routing

# Chapter 45: Use Private Meteor Packages on Codeship

## Remarks

Note that we did not discuss how to use & develop your local packages. There are several ways, I suggest to use the `PACKAGE_DIRS` environment variable described by David Weldon on his website.

## Examples

### Install MGP

We make use of Dispatches great Meteor Github Packages (mgp) package:

```
npm install --save mgp
```

Then, add the following command to your `package.json` scripts:

```
"mgp": "mgp"
```

Create a file named `git-packages.json` in your project root. Add a config for every (private) Meteor Github package that your project depends on:

```
{
  "my:yet-another-private-package": {
    "git": "git@github.com:my/private-packages.git",
    "branch": "dev"
  }
}
```

More information about how to configure your private packages can be found on the projects Github repo.

### Configure Codeship to Install Private Github Packages

Append the following command to the Codeship setup commands:

```
meteor npm run mgp
```

Now, we need to give Codeship access to these private repositories. There is a Codeship documentation article describing this process in detail but here are the steps that you have to take for Github:

- Create a new Github account. A so called Machine user.
- Remove the deploy key from your repo under test. Here:

https://github.com/YOUR_USERNAME/REPO_UNDER_TEST/settings/keys

- Grab the SSH public key from your codeship projects settings. Somewhere here: https://codeship.com/projects/PROJECT_NUMBER/configure
- Add this SSH public key to your machine user's SSH keys: https://github.com/settings/keys
- Give this machine user access to all your referenced repositories

It should be similar for BitBucket and others.

Read Use Private Meteor Packages on Codeship online: https://riptutorial.com/meteor/topic/6742/use-private-meteor-packages-on-codeship

# Chapter 46: Using Meteor with a Proxy Server

## Examples

### Using the `HTTP[S]_PROXY` env var

This page describes how to use the Meteor command-line tool (for example, when downloading packages, deploying your app, etc) behind a proxy server.

Like a lot of other command-line software, the Meteor tool reads the proxy configuration from the `HTTP_PROXY` and `HTTPS_PROXY` environment variables (the lower case variants work, too). Examples of running Meteor behind a proxy:

- on Linux or Mac OS X

```
export HTTP_PROXY=http://user:password@1.2.3.4:5678
export HTTPS_PROXY=http://user:password@1.2.3.4:5678
meteor update
```

- on Windows

```
SET HTTP_PROXY=http://user:password@1.2.3.4:5678
SET HTTPS_PROXY=http://user:password@1.2.3.4:5678
meteor update
```

### Setting Up a Proxy Tier

- Deploy Meteor App to Ubuntu with Nginx Proxy
- How to Create an SSL Certificate on Nginx for Ubuntu 14
- How to Deploy a Meteor JS App on Ubuntu with Nginx
- How to Install an SSL Certificate from a Commercial Certificate Authority
- NameCheap SSL Certificates

Read Using Meteor with a Proxy Server online: https://riptutorial.com/meteor/topic/517/using-meteor-with-a-proxy-server

# Chapter 47: Using Polymer with Meteor

## Examples

### Using differential:vulcanize

In the root of your project, make sure Bower is installed (`npm install -g bower`) and run `bower init`. This will create a `bower.json` file in your project's directory.

Create a new file called `.bowerrc` to your root directory. It should contain the following:

```
{
  "directory": "public/bower_components"
}
```

This lets Bower know that it should save components in the `bower_components` folder in your app's public directory.

Now add the Polymer components you wish to use with your app.

In your app's root directory bower-install each component you want to use.

```
bower install --save PolymerElements/paper-button#^1.0.0 PolymerElements/paper-checkbox#^1.0.0
```

Add Vulcanize to your project

```
Meteor add differential:vulcanize
```

Create a new file called config.vulcanize in the root of your project. It should contain the following:

```
{
    "polyfill": "/bower_components/webcomponentsjs/webcomponents.min.js",
    "useShadowDom": true, // optional, defaults to shady dom (polymer default)
    "imports": [
        "/bower_components/paper-button/paper-button.html",
        "/bower_components/paper-checkbox/paper-checkbox.html"
    ]
}
```

`"imports"` should list each component you will use in your app.

You can now use components you have imported in your Blaze templates just as you would any other element:

```
<template name="example">
    <div>
        this is a material design button: <paper-button></paper-button>
        this is a material design checkbox: <paper-checkbox></paper-checkbox>
    </div>
```

```
</template>
```

Read Using Polymer with Meteor online: https://riptutorial.com/meteor/topic/4598/using-polymer-with-meteor

# Chapter 48: Wrapping asynchronous methods into a Fiber for synchronous execution.

## Syntax

1. Meteor.wrapAsync(func, [context])

## Parameters

| Parameters | Details |
|---|---|
| func: Function | An asynchronous/synchronous function to be wrapped in a Fiber that takes a callback w/ parameters `(error, result)`. |
| context: Any (optional) | A data context in which the function gets executed upon. |

## Remarks

An asynchronously wrapped function may still be ran asynchronously if a callback with parameters `(error, result) => {}` is given as a parameter to the wrapped function.

The incorporation of `Meteor.wrapAsync` allows for code ridden with callbacks to be simplified given that callbacks can now be neglected in compensation for making the call block its present `Fiber`.

To understand how Fibers work, read here: https://www.npmjs.com/package/fibers.

## Examples

### Synchronously executing asynchronous NPM methods w/ callbacks.

This example wraps the asynchronous method `oauth2.client.getToken(callback)` from the package NPM package `simple-oauth2`into a Fiber so that the method may be called synchronously.

```
const oauth2 = require('simple-oauth2')(credentials);

const credentials = {
    clientID: '#####',
    clientSecret: '#####',
    site: "API Endpoint Here."
};
```

```
Meteor.startup(() => {
    let token = Meteor.wrapAsync(oauth2.client.getToken)({});
    if (token) {
        let headers = {
            'Content-Type': "application/json",
            'Authorization': `Bearer ${token.access_token}`
        }

        // Make use of requested OAuth2 Token Here (Meteor HTTP.get).
    }
});
```

Read Wrapping asynchronous methods into a Fiber for synchronous execution. online:
https://riptutorial.com/meteor/topic/2530/wrapping-asynchronous-methods-into-a-fiber-for-synchronous-execution-

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with meteor | Ankit, Christian Fritz, Community, Gal Dreiman, ghybs, grahan, hwillson, João Rodrigues, levon, Matthias Eckhart, mav, mertyildiran, Ray, reoh, robfallows, Tom Coleman, Zoltan Olah |
| 2 | Acceptance Testing (with Nightwatch) | AbigailW |
| 3 | Accessing Meteor build machines from Windows | Tom Coleman |
| 4 | Assets | Matthias Eckhart |
| 5 | Background tasks | Filipe Névola |
| 6 | Basic Codeship Setup for Automated Testing | schmidsi |
| 7 | Beginner guide to Installing Meteor 1.4 on AWS EC2 | AGdev |
| 8 | Blaze Templating | Dan Cramer, distalx, ghybs, jordanwillis, khem poudel, RamenChef, robfallows, Thomas Gerot |
| 9 | Blaze User Interface Recipes (Bootstrap; No jQuery) | AbigailW, Anis D |
| 10 | Continuous Deployment to Galaxy from Codeship | schmidsi |
| 11 | Continuous Integration & Device Clouds (with Nightwatch) | 4444, AbigailW |
| 12 | Debugging | AbigailW, distalx |

|  |  |  |
|---|---|---|
| 13 | Deployment with Upstart | AbigailW, ghybs |
| 14 | Development Tools | AbigailW, Ankit, Ankit Balyan, Fermuch, Ilya Lyamkin |
| 15 | Directory Structure | AbigailW, anomepani, ghybs, Michael Balmes, Nick Carson, Phe0nix, reoh, Thomas Gerot |
| 16 | Electrify - Compiling Meteor as a Locally Installable App | AbigailW, JuanGesino, Nick Bull, RamenChef |
| 17 | Environment Detection | AbigailW, ghybs |
| 18 | Environment Variables | AbigailW, hcvst |
| 19 | ES2015 modules (Import & Export) | reoh |
| 20 | ESLint | saimeunt |
| 21 | File Uploading | AbigailW |
| 22 | Full Installation - Mac OSX | AbigailW, RamenChef |
| 23 | Horizontal Scaling | AbigailW |
| 24 | Integration of 3rd Party APIs | AbigailW |
| 25 | Logging | AbigailW |
| 26 | Meteor + React | AbigailW, aedm, corvid, ghybs, RamenChef, Teagan Atwater, zliw |
| 27 | Meteor + React + ReactRouter | rafahoro |
| 28 | Meteor User Accounts | Barry Michael Doyle, KrisVos130 |
| 29 | Mobile Apps | AbigailW, Anis D, Antti Haapala, ghybs |
| 30 | Mongo Collections | AbigailW |
| 31 | Mongo Database Management | AbigailW, distalx, RamenChef, TechplexEngineer |

| | | |
|---|---|---|
| 32 | Mongo Schema Migrations | AbigailW |
| 33 | MongoDB | distalx, Dranithix, hwillson, Matthias Eckhart, robfallows, Thomas Gerot |
| 34 | MongoDB Aggregation | AbigailW, levon |
| 35 | Nightwatch - Configuration & Setup | AbigailW |
| 36 | Node/NPM | hwillson |
| 37 | Offline Apps | AbigailW |
| 38 | Performance Tuning | AbigailW, RamenChef, reoh |
| 39 | Publishing A Release Track | AbigailW |
| 40 | Publishing Data | Abdelrahman Elkady, AbigailW, Chris Pena, corvid, Dair, dangsonbk, Eliezer Steinbock, Faysal Ahmed, ghybs, j6m8, Maciek, RamenChef, Ramil Muratov, robfallows, Serkan Durusoy |
| 41 | Reactive (Vars & Dictionaries) | Ankit |
| 42 | Replica Sets and Sharding | AbigailW, Anis D |
| 43 | Retrieving data from a Meteor.call | Ramil Muratov, Rolljee, Sacha |
| 44 | Routing | Ankit, ghybs, Luna, Michael Balmes |
| 45 | Use Private Meteor Packages on Codeship | schmidsi |
| 46 | Using Meteor with a Proxy Server | AbigailW, Serkan Durusoy, Tom Coleman |
| 47 | Using Polymer with Meteor | Thaum Rystra |
| 48 | Wrapping asynchronous | Dranithix |

| | |
|---|---|
| methods into a Fiber for synchronous execution. | |