← back to Meteor Patterns

Multiple builds for a project

Build and deploy different apps and services from the same source

Pieter Soudan — https://gist.github.com/Sewdn/855f6e4f7d889be012ee

TL;DR

Meteor is great at sharing code between different builds for different platforms. You can use the same source for your browser builds, server builds, your builds for iOS, Android, ... But how to organize your project to be able to orchestrate your builds for different apps and services from the same source? This post elaborates on the reasons why you need these different builds and how you could accomplish this with Meteor easily.

Use cases: Why would you build different apps?

1. DIFFERENT APPS FOR DIFFERENT ROLES

Say you have an app with completely different end user experiences depending on their role. It is common practice to have the user logged in, check her authorization (role) and then setup different routes or load different templates to serve that type of user's needs. While doing so, all types of users load the same build and the app decides what portions of the build to use and what not to use.

This results in builds that can be way too large for a user with a role that can actually do very few things with your app. A common example that illustrates this perfectly is a landing page. An unauthenticated user only needs to be shown the landing page, she must be able to login and register a new account, only. Worse, this approach may lead to security vulnerabilities. Because all of the logic is deployed and catered to all users (whatever their role), you must be damn sure access rights are checked all the way through, in every single aspect of your app.

Wouldn't it then make more sense to never load any code at all for users who'll never need it, or shouldn't use it? Sure it would, and deploying different apps at different subdomains (yourapp.io,

app.yourapp.io, admin.yourapp.io, etc.) while making sure users are redirected to using the app which fits their current role and access rights, may well be the way to go. But then you'll soon run into maintenance headaches and will struggle with code organization.

2. DIFFERENT BUILDS FOR DIFFERENT MICROSERVICES

Say you have a very computation heavy process somewhere in your backend, which you know will have scaling issues as soon as the user base is growing. You then want to be able to isolate this process from the rest of your app, by deploying it as a independent (micro)service and by scaling that service seperately, along with the user base.

Sure it's possible to setup this process as a different project, and have your Meteor app communicate with the service as it would do with any other third-party service. (Or, better still, over Meteor's built-in DDP since both the mother app and the microservice speak Meteor). But say you would now want to go this way and deploy a dozen different components within your backend — the burden of building and maintaining different services would grow into a very tedious process, again. Development itself would be getting harder too: you'd be forced to setup a complicated local development environment, running all of the microservices involved on different ports, and to configurate communication between them.

Wouldn't it then make more sense to develop all those services within

a single app during development, while at build time separate all the many different microservices again into stand-alone production apps, each scalable in their own right?

Organize your code in app-packages

So we want to organize our code such that we may build (deploy and scale) different apps/services from a single development source, very similar as to how one would go about when developing an app for several native platforms, only this time around targetting the Web as the platform, to which you'll be releasing an app-network of services talking with each other.

Key to the solution is to architect your project as a collection of app-specific packages. You'll then be able to lift-out all of the app-level Meteor code and organize it in packages. Not as in those regular packages you're used to distribute on Atmosphere, but as in local, app-specific packages. Indeed, per package you would add a directory in your packages folder, add a package.js file to it, include the package in your app-dependencies (.meteor/packages) and you're good to go.

example project directory structure:

```
/project
```

```
/.meteor
  packages
/packages
  /myproject:core
  /myproject:app-ui
  /myproject:app-profile
  /myproject:app-projects
```

example package.js - /packages/myproject:app-ui/package.js

```
Package.describe({
  name: 'myproject:app-ui',
  version: '0.0.1',
  summary: 'My Projects User Interface',
});

Package.onUse(function(api) {
  api.versionsFrom('1.1.0.3');

  api.use([
    'myproject:core',
    'myproject:app-profile',
    'templating',
    'kadira:flow-router',
    'kadira:blaze-layout'
  ], 'client');

  api.addFiles([
    'head.html',
    'layout.html',
    'components/logo.html',
```

```
    'routing.js'
  ], 'client');
});
```

With all of your code neatly organized in packages, you now have much more control over build order and your dependencies are more clearly stated, too. Also, you'll benefit from naturally writing more modular code, since you start thinking about encapsulating business logic in separate packages, with a clear dependency chain in mind.

SETUP UMBRELLA PACKAGES FOR BUILDS

You should also setup an *umbrella package* for every app you want to build in your project. An umbrella package is a package only implying other dependant packages, but not adding any logic (and thus not adding any file to the build, or using any package).

example umbrella package.js - /packages/myproject:app/package.js

```
Package.describe({
  name: 'myproject:app',
  version: '0.0.1',
  summary: 'My Projects User facing app umbrella package',
});

Package.onUse(function(api) {
  api.versionsFrom('1.1.0.3');
```

```
    api.imply([
      'myproject:core',
      'myproject:app-profile',
      'myproject:app-projects',
      'myproject:app-ui'
    ], ['client', 'server']);

});
```

Basically you're replacing your .meteor/packages file by a package. You're moving dependency management to an app umbrella package. Doing so, there is no logic or dependency control left at app-level, but more importantly you can start creating apps that inherit from other apps by implying other app umbrella packages. This way you can setup your apps as a cascading model of functionality: admin-app > moderator-app > user-app > guest-app > unathenticated-app.

example packages file - /.meteor/packages

```
meteor-platform
myproject:app
```

Organize your project in different meteor projects

for each build

Ok, we now have organized all of our app code for reuse, but we didn't reuse it for now, because we haven't setup other apps yet.

Instead of creating a meteor project at the root of your project repository, create a meteor project in a sub-folder for every build you need (landingpage, app, microServiceA, ...).

Setup a packages directory at root level of your project, and symlink to this directory from every single app-directory.

Doing so, every app has its app-specific .meteor directory, containing all needed files to orchestrate the meteor build process (.id, packages, platforms, ...). They can run independently from other apps in your project, but still share all of your business logic, picking what is needed for each build, using your umbrella packages.

This results in the following directory structure:

```
/project
  /app
    /.meteor
    /packages => ../packages
  /landing
    /.meteor
    /packages => ../packages
  /services
```

```
    /.meteor
    /packages => ../packages
  /packages
    /myproject:core
    /myproject:app
    /myproject:app-profile
    /myproject:app-projects
    /myproject:app-ui
    /myprojects:services
    /myprojects:landing
```

Configuring your apps for local development and production builds

When you have all your apps and packages in place, you should optimize on local development. You don't need to mimick the deployment for production on your local development machine. Microservices are needed to be able to scale your app at the next bottleneck, but this is only the case for production builds. During development you just want to work with one node process (for each of your end user apps) including all of your microservices. I'm not saying you shouldn't test the microservices build, but it's overhead to do this while developing. This is only needed during testing (using continuous integration techniques).

SETTINGS.JSON FILES FOR EVERY APP

You can control this by providing each app with its own set of settings files for local development. In these settings files you can control what background processes must run in your services. During local development, you mostly enable all of your background processes for all services, while you disable them for production builds in your user-facing app. You'd rather configure separate service-apps enabling only one background process per build, so you can scale them separately.

Make sure to add a folder to your apps containing your settings files:

```
/project
  /package.json
  /app
    /.meteor
    /.deploy
      local.json
    /packages => ../packages
  /landing
    /.meteor
    /.deploy
      local.json
    /packages => ../packages
  /services
    /.meteor
    /.deploy
      local.json
    /packages => ../packages
```

```
    /packages
      ...
```

You now have your settings files for local development and can start your apps in development mode:

```
$ cd app
$ meteor --settings .deploy/local.json --port 3000
$ cd landing
$ meteor --settings .deploy/local.json --port 3001
```

BUILD & DEPLOY FOR PRODUCTION

I presume you are familiar with MUP. MUP is a node script for building and deploying your builds on your infrastructure of choice. MDG is working on Galaxy and recently released a preview in private beta, but I believe (and others with me) MUP will still be valuable for lots of developers even when Galaxy is released to the public.

SETUP MUP FOR ALL YOUR APPS/SERVICES

Organize your mup settings in the .deploy directories at app-level for every environment you want to deploy to:

```
$ mkdir -p app/.deploy/production
```

```
$ cd app/.deploy/production
$ mup init
```

You can then easily build and deploy all of your apps and services by configuring mup.json and defining other settings for production deployments.

```
/project
  /package.json
  /app
    /.deploy
      local.json
      /production
        mup.json
        settings.json
      /staging
        mup.json
        settings.json
    /packages => ../packages
  /landing
    /.deploy
      local.json
      /production
        mup.json
        settings.json
  /services
    /.deploy
      local.json
      /production
        mup.json
```

settings.json

setting up npm scripts

To make this even more developer friendly, you can set up npm scripts, to setup your own CLI to run, build and deploy your apps. Generate a package.json file at the root level of your project and setup your scripts like the following example.

package.json

```
{
  "name": "myproject",
  "scripts": {
    "start": "cd app && MONGO_URL=mongodb://localhost:27017/myproj
    "landing": "cd landing && MONGO_URL=mongodb://localhost:27017/
    "services": "cd services && MONGO_URL=mongodb://localhost:2701
    "deploy": "npm run deploy-app & npm run deploy-landing & npm r
    "deploy-app": "cd app/.deploy/production && mup deploy",
    "deploy-landing": "cd landing/.deploy/production && mup deploy
    "deploy-services": "cd services/.deploy/production && mup depl
  }
}
```

You can now run your servers and execute build with these simple commands:

```
$ cd project
$ npm start
$ npm run landing
$ npm run services
$ npm run deploy
```

Conclusion

This Meteor pattern for organizing builds for apps and microservices is working very well for us. You'll gain more control over your builds, get better oversight over the dependencies within your project, write more maintainable and modular code, and you'll be able to scale your processes (microservices) separately, each according to their need. We think it's a pattern every large Meteor project should adapt, decidedly as soon as it grows beyond the prototyping phase.

Did you find this article useful?

THEN SHARE THIS RESOURCE WITH YOUR PEERS.

Share with Twitter

OR FOLLOW ME ON TWITTER FOR OTHER UPDATES

Follow @Sewdn

ABOUT

Meteorpatterns.com is a resource covering interesting design patterns and insights for meteor development

Made by Red&Ivory

JOIN IN!

This platform is setup as a frontend for external content on github. Contact me if you would like to publish on this platform.

pieter.soudan@redandivory.be
@sewdn