**Webpack Hot Reloading and React**

**– You already use HR ?**

If you have a project using webpack, React, you probably have followed some tutorials and blogs to add the **Hot Reloading** piece.

Here, we are going to explain what’s going on **under the hood**, and how the HR projects / plugins you installed are working together (webpack HR middlewares, react-transform..).

We are going to use the latest React HR project in date [**react-transform-hmr**](https://github.com/gaearon/react-transform-hmr) and not the older one [**react-hot-loader**](https://github.com/gaearon/react-hot-loader) which has been sentenced to the maximum penalty : [the-death-of-react-hot-loader](https://medium.com/@dan_abramov/the-death-of-react-hot-loader-765fa791d7c4). I guess it’s still working properly but you know that the Javascript community is always on the bleeding edge side!

**– You don’t already use HR ?**

If the HR part does not sound familiar to you, that’s fine because I’m going to explain step-by-step why it’s mandatory to use HR nowadays, and what is the way to install and use it into your project.

Let’s just start by resuming why everybody is now talking about HR (beyond Javascript).

Why should we enjoy Hot Reloading ?

In one word : **productivity**.  
In another word : **DX** (Developer eXperience).

Some more explanations :

– you broke your F5 key years ago, you can’t refresh anymore  
– you don’t want to develop blindly, then refresh, then try again. It’s a backend thing. We have a UI, it’s **alive**  
– you don’t want to lose your **application state** (by doing a full refresh) if you just fixed a typo or a color  
– that can display your compilation and runtime **errors** directly **in the browser** in the exact spot in the UI where it’s used  
– you can do HR with Javascript, CSS, anything. We’ll just focus on React components here.  
– no caveat

As soon as you put it in place, it will work **forever** without any modifications, for any of the future files you’re going to add. It’s just some pipes to plug together to make it work and last forever.

2015-12-28: actually there is only one caveat (!) : the React HR plugin we are going to use does not handle the [stateless functional components](https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html#stateless-functional-components) created with the simpler React v0.14 syntax : **const App = (props) => { <div>Hello {props.name}</div> };**.  
Refer to [@dan\_abramov](https://twitter.com/@dan_abramov) [comment](https://github.com/gaearon/babel-plugin-react-transform/issues/57#issuecomment-167675819).

Now, let’s tackle the packages we need to install to use it in a nodejs project.

What packages to install to use HR ?

Let’s suppose you already have a base project using :

– **webpack** to compile your javascript bundle(s)  
– **React** for your frontend ofc  
– a nodejs server that runs a **http server** such as **expressjs** to serve your static content (html, js, css..) while you’re developing

You now want to experiment HR.

webpack to the rescue

**webpack** is actually the main actor dealing with the HR, it already exposes an API to process some part of the HR pipeline.

We just need to add some wrapper around to use its API, and some more logic to manage the specific React components state : **you don’t want to lose the current state** of your components when you change something in a js file (a style, some constant, a prop, add a React component inside an existing one etc.).

We need to install 4 packages : 2 for webpack, 2 for React.

[webpack-dev-middleware](https://github.com/webpack/webpack-dev-middleware)  
– a classic expressjs [middleware](http://expressjs.com/en/guide/using-middleware.html), where requests are passed on  
– it automatically **watches the sources** for changes to recompile the javascript bundle server side if some javascript source have changed  
– it always serves the bundle up to date

[webpack-hot-middleware](https://github.com/glenjamin/webpack-hot-middleware)  
– a classic expressjs [middleware](http://expressjs.com/en/guide/using-middleware.html), where requests are passed on  
– it automatically **subscribes to the bundle recompilation events** (such as “start”, “done”), to notify the frontend that something has changed and it need to update itself  
– it uses [SSE](http://www.html5rocks.com/en/tutorials/eventsource/basic) to communicate with the frontend

Specific packages for React HR

[babel-plugin-react-transform](https://github.com/gaearon/babel-plugin-react-transform)  
– it can add any code around React component methods during the Babel compilation ES6+JSX to ES5. You should already have configured your babel loader in webpack, such as :

|  |
| --- |
| module: {     loaders: [{       test: /\.js$/,       loader: 'babel',       include: path.join(\_\_dirname, 'src'),     }]   }, |

[react-transform-hmr](https://github.com/gaearon/react-transform-hmr)  
– it is used by **babel-plugin-react-transform** to add specific code around the React components to properly handle HR and their current state

That gives us :

|  |
| --- |
| $ npm install --save-dev webpack-dev-middleware  $ npm install --save-dev webpack-hot-middleware  $ npm install --save-dev babel-plugin-react-transform@beta  $ npm install --save-dev react-transform-hmr  // or for the copy/paste, all together :  $ npm i -D webpack-dev-middleware webpack-hot-middleware babel-plugin-react-transform@beta react-transform-hmr |

Now that we have installed the necessary packages, it’s time to configure them.

Configure Babel

We need to configure Babel (we are going to use **.babelrc**) to use **babel-plugin-react-transform**and **react-transform-hmr** to add the HR code around them.

The most basic configuration is :

|  |
| --- |
| {    "presets": ["react", "es2015"],    "env": {      "development": {        "plugins": [          ["react-transform", {            "transforms": [{              "transform": "react-transform-hmr",              "imports": ["react"],              "locals": ["module"]            }]          }]        ]      }    }  } |

Basically :  
– that adds the transform **babel-plugin-react-transform** for the **development** NODE\_ENV only  
– this transform retrieve all the React components it can find in the source code  
– it passes them down to each of its processors defined in **"transforms"** to let them add their custom code. (in our case, **react-transform-hmr** will add the HR code)

For the record, **babel-plugin-react-transform** handles as many **"transforms"** as we want. They are just going to be called right after each other. For instance :

|  |
| --- |
| {    "presets": ["react", "es2015"],    "env": {      "development": {        "plugins": [          ["react-transform", {            "transforms": [{              "transform": "react-transform-hmr",              "imports": ["react"],              "locals": ["module"]            }, {              "transform": "react-transform-catch-errors",              "imports": ["react", "redbox-react"]            }]          }]        ]      }    }  } |

The React component code will pass through **react-transform-hmr**, then through **react-transform-catch-errors**.  
Each of them will add its code around each components.

FYI, the latter is used to catch errors that are thrown in the **render()** method of the React components. Then it’s using its **"imports"** property to redirect the error to a visual React component. Here [redbox-react](https://github.com/KeywordBrain/redbox-react" \t "_blank) displays a big red screen of the death with the stacktrace for instance. But it could be anything else.

Basically, **react-transform-catch-errors** just adds **try { render() } catch (e) { ... }** around the original **render()** method of your components, and in the **catch**, it’s returning the React component you gave in **"imports"**. Makes sense right ?

Now, our React code is ready to handle HR.  
We now have to make the server communicate to the browser that the code has changed and that it needs to update.

Handle server/client communication to send/receive HR updates

Bundle recompilation on the fly

First, we need to make the server aware that the source code has changed to recompile the bundle, and then notify the browser.  
That’s the role of **webpack-dev-middleware** and **webpack-hot-middleware**.

– **webpack-dev-middleware** will automatically start to watch the source code for changes and recompile the bundle  
– **webpack-hot-middleware** will be notified a new bundle is compiled and will notify the browser

We just need to plug them into **expressjs** as middlewares to start them :

|  |
| --- |
| var express = require('express');  var webpack = require('webpack');  var path = require('path');    var webpackDevMiddleware = require("webpack-dev-middleware");  var webpackHotMiddleware = require("webpack-hot-middleware");  var webpackConfig = require('../webpack.config');    var app = express();    var compiler = webpack(webpackConfig);  app.use(webpackDevMiddleware(compiler));  app.use(webpackHotMiddleware(compiler));  app.use(express.static('src'));    app.listen(3000); |

But how is the browser going to handle the updates ? It’s where webpack itself rises.

Browser live update

We need to add some code client-side to deal with it, otherwise, it’s not possible. HR is not browser native or anything.

Therefore, to inject some more code, we will use the webpack bundle **entry** point in**webpack.config.js**, to add another entry.  
A bundle could have **several** entry points.  
It is just there to say to webpack : “hey, resolve the import dependency (for make the bundle) tree starting from those files!”.

|  |
| --- |
| entry: [    'webpack-hot-middleware/client',    path.join(\_\_dirname, 'src', 'App.js'),  ], |

The entrypoint **webpack-hot-middleware/client** simply refers to the file**node\_modules/webpack-hot-middleware/client.js**.  
It is the file that contains the code that will be used in the browser to handle the **SSE**communication with the server (to intercept the update notifications).

Then we need to add a specific webpack internal plugin **HotModuleReplacementPlugin** to expose the generic webpack HR API in the browser :

|  |
| --- |
| plugins: [    new webpack.optimize.OccurenceOrderPlugin(), // recommanded by webpack    new webpack.HotModuleReplacementPlugin(),    new webpack.NoErrorsPlugin() // recommanded by webpack  ] |

This API will be used by the code injected from **webpack-hot-middleware/client** when this one will retrieve some “update” events through its SSE. (specifically, it will call **module.hot.apply(..)**from **HotModuleReplacementPlugin**).  
You follow ?

Nothing more to do, you’re good to go !

See it in action

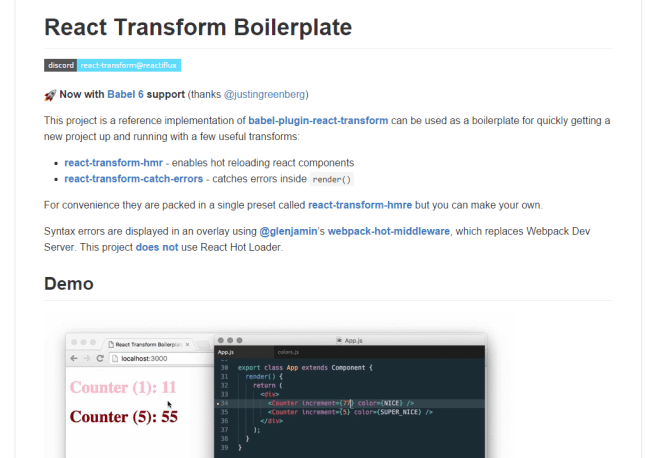
Process :  
– start your nodejs server  
– go to your page using the bundle on your browser  
– go to your code editor and **change some javascript** bits  
– see the **live update** !  
– if not, check your **console** in the browser to see some nice error messages

Behind the scene :  
– the server automatically recompile the bundle if some javascript code is updated and notify the browser to update itself via SSE  
– the bundle used in the browser contains some SSE code to intercept those notifications, some generic HR code to **“patch”** the javascript, and some custom HR code for the React components to **not lose their current state**

A boilerplate on github

You can checkout this boilerplate from @dan\_abramov <https://github.com/gaearon/react-transform-boilerplate> to give a try.  
It’s very simple and does exactly what we just talked about.  
You will see some more options used but nothing fancy.

Its **.babelrc** is using [**react-hmre**](https://github.com/danmartinez101/babel-preset-react-hmre) instead of our two transforms packages (**babel-plugin-react-transform@beta** and **react-transform-hmr**), but it is exactly the same actually. **react-hmre**simply encapsulates them.



More bits to learn

Let’s explain some more in-depth some aspects of the code used.

webpack-dev-middleware is optional, but…

Without **webpack-dev-middleware**, you just need to launch the webpack **watch** yourself :

|  |
| --- |
| // app.use(webpackDevMiddleware(compiler));  // replace the dev-middleware with a simple watch() (args are mandatory)  compiler.watch({}, function(){});  app.use(webpackHotMiddleware(compiler)); |

Because **webpackHotMiddleware** subscribes to the bundle compilation events (no matter what started it), it will work.

But you’ll **suffer** from some consequences : a bunch of **.js** and **.json** files in your project will appears each time a compilation occurs.

They contain the delta sent to the client to update itself (**webpack** only sends the updated**chunks**, not the whole bundle each time). The advantage of using **webpack-dev-middleware** is that you won’t see those files. They will be handled **in-memory** by itself. That’s why you need to install this particular package too.

babel-plugin-react-transform and react-transform-hmr

Without the code added by **react-transform-hmr**, **webpack** would not be able to hot update the React components, you would get this in the browser console :

|  |
| --- |
| [HMR] bundle rebuilding  [HMR] bundle rebuilt in 160ms  [HMR] Checking for updates on the server...  [HMR] The following modules couldn't be hot updated: (Full reload needed)  [HMR]  - ./src/App.js |

What’s inside the SSE ?

1. The browser initializes a **SSE** request : (thanks to the code **webpack-hot-middleware/client.js**injected in the bundle) on this specific url : **GET localhost:3000/\_\_webpack\_hmr** (it never returns). It’s handled by the server that knows it’s SSE. (by the **webpack-hot-middleware**expressjs middleware)

Then if a Javascript file is edited, because **webpack-dev-middleware** started a watch on the sources, **webpack-hot-middleware** is notified (because it subscribed to the recompilation events) and notify the frontend via SSE with a new module map (used by webpack), such as :

|  |
| --- |
| data: {"action":"building"}    // few ms later ...    data: {"action":"built","time":260,"hash":"6b625811aa23ea1ec259","warnings":[],"errors":[],"modules":{"0":"multi main","1":"./~/fbjs/lib/invariant.js","2":"./~/react/lib/Object.assign.js","3":"./~/fbjs/lib/warning.js","4":"./~/fbjs/lib/ExecutionEnvironment.js","5":"./~/react/lib/ReactMount.js","6":"./~/react/lib/ReactElement.js", ... |

Then the frontend asks for those 2 files:  
(I guess the hash used in the GET comes from the SSE data ? I didn’t check)

– **GET localhost:3000/0.0119cbdcd4c2cf8d27c2.hot-update.js**

|  |
| --- |
| {"h":"6b625811aa23ea1ec259","c":[0]} |

– **GET localhost:3000/0119cbdcd4c2cf8d27c2.hot-update.json**

|  |
| --- |
| webpackHotUpdate(0,{    /\*\*\*/ 97:  /\*\*\*/ function(module, exports, \_\_webpack\_require\_\_) {      /\* WEBPACK VAR INJECTION \*/(function(module) {'use strict';      ... |

Those particular urls are served by **webpack-dev-middleware** which is keeping those files (**\*.hot-update.js[on]**) in memory and serves them as any classic static file. They are requested by the code injected by **webpack.HotModuleReplacementPlugin()** that handles the responses and hot-updates the javascript with the new code.

Conclusion

Before I dug into, I was finding that the HR was a bit complicated to understand how it works, who are the actors, how does it communicate, etc. I was never sure what package was used for what exactly, and when. I hope those explanations were clear enough and that now, you know as much as me ! (or more!) Don’t hesitate to add more details or correct me if I’m wrong.

Finally, it’s not that complex but it just need a lot of “pipes” to plug everything together. But it all makes sense.