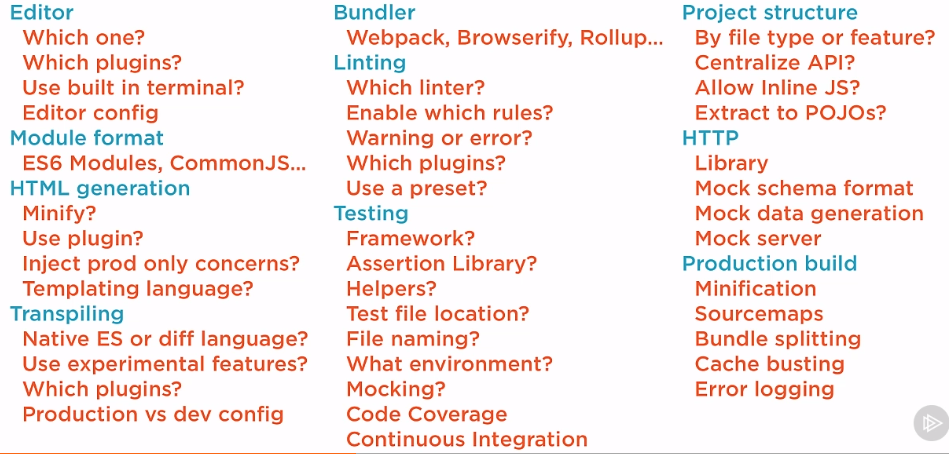
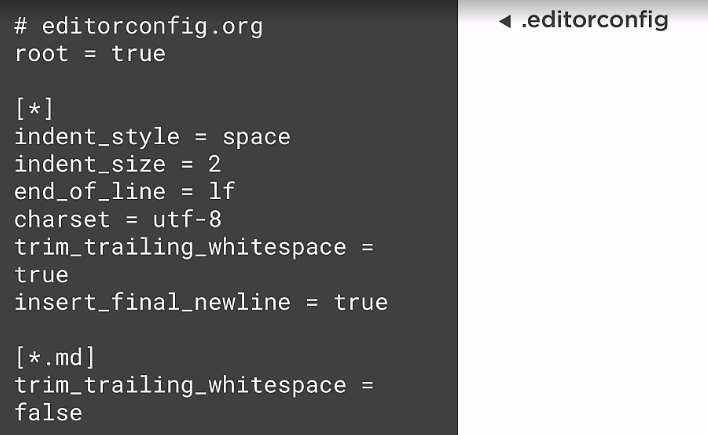
**INTRO**

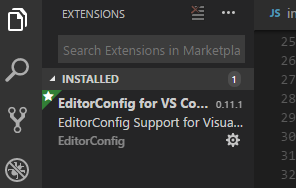




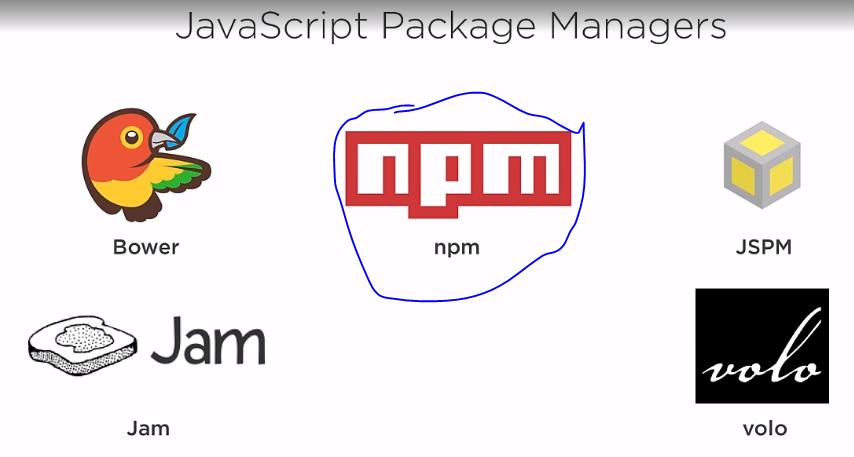
**EDITOR AND CONFIGURATION**

In the root:





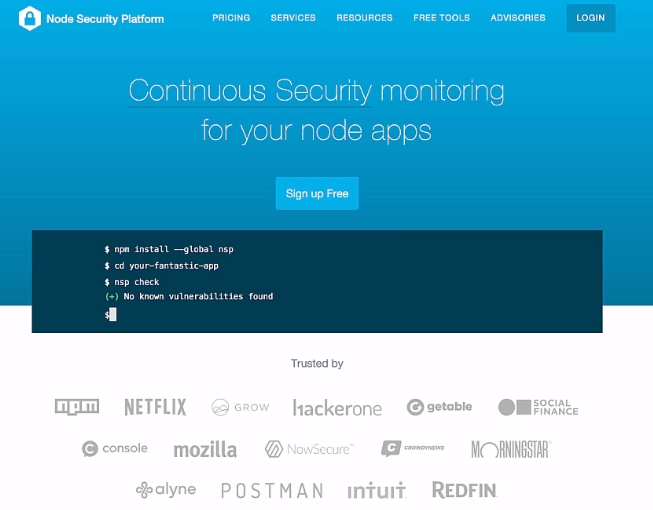
**PACKAGE MANAGEMENT**

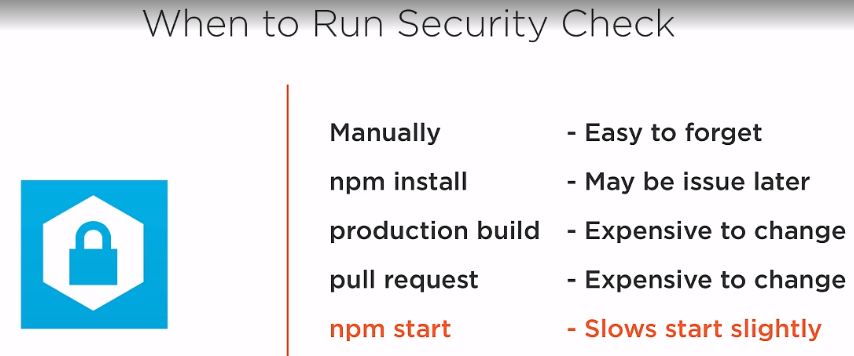


Package.json stores a list of npm package we are using as well npm script we will setup later.



Npm install 🡪 mette tutti questi pacchetti in una cartella chiamata nodemodules del progetto





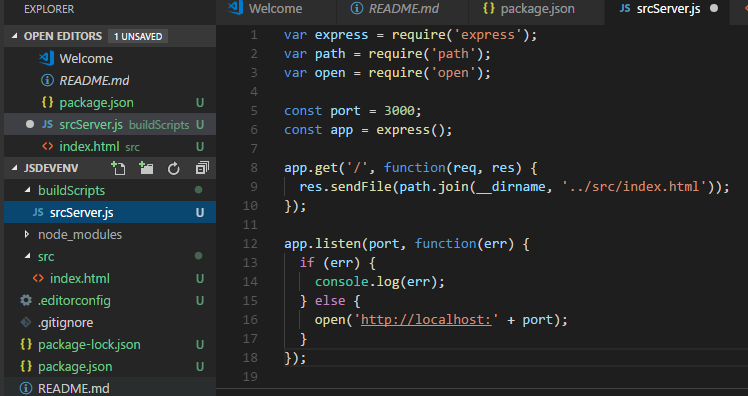
In general, the rule of thumb is:

1. If you’re installing something that you want to use *in* your program, using require('whatever'), then install it locally, at the root of your project.
2. If you’re installing something that you want to use in your *shell*, on the command line or something, install it globally, so that its binaries end up in your PATH environment variable.

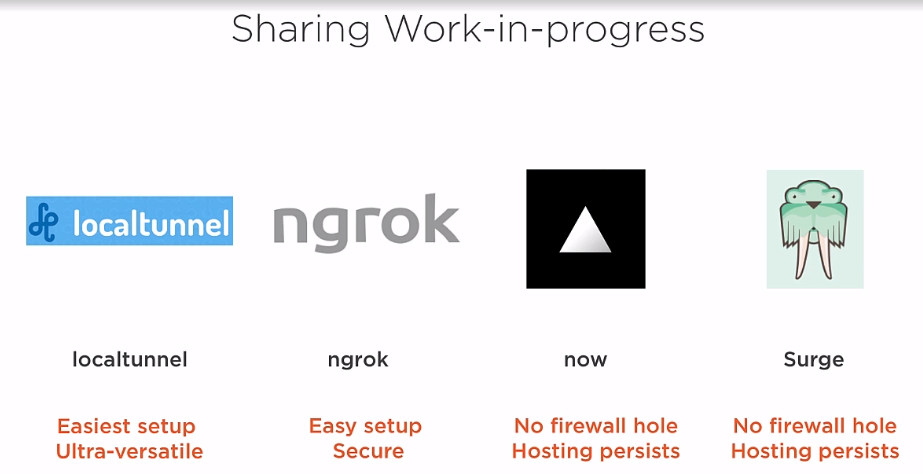
npm install –g nsp



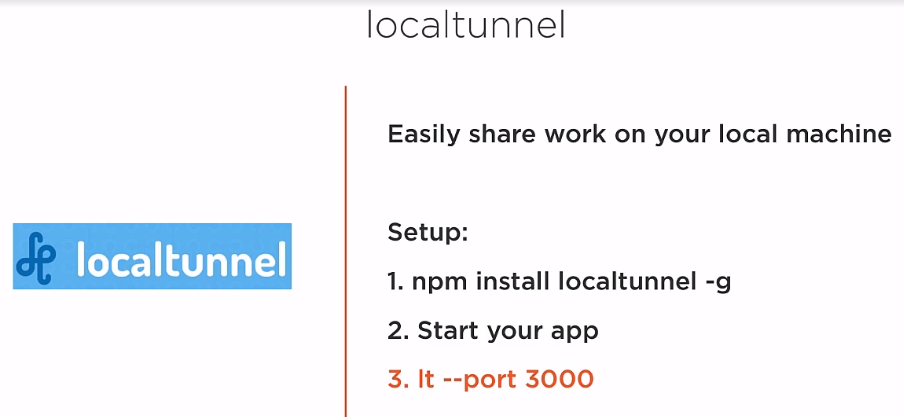
**DEVELOPMENT WEB SERVER**



node .\buildScripts\srcServer.js



Localtunnel offers an elegant way to expose your local host via a public url. It punches a hole in your firewall so that your local machine can operate as a web server.



npm install localtunnel –g

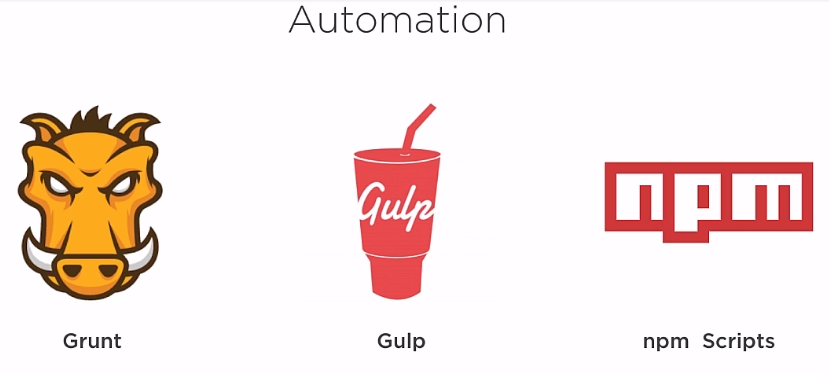
node .\buildScripts\srcServer.js

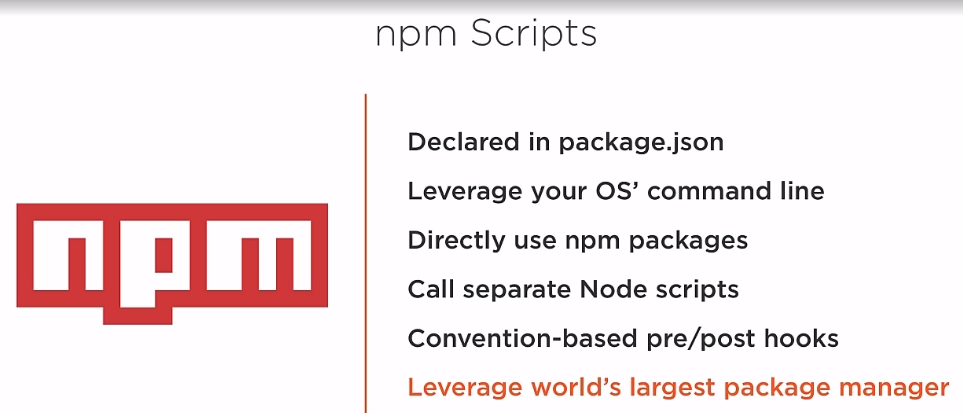
apro un altro terminale

lt –port 3000 (elle ti meno meno)

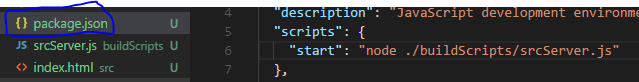
lt –port 3000 –subdomain KiraDiShira

**AUTOMATION**





By convention “start” is the command who launch my app.



npm start

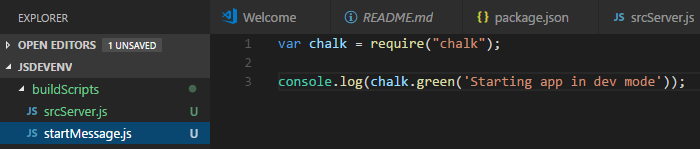


Per lanciare tutti gli altri script: npm run nomeScript

Esiste la convenzione tra gli script:



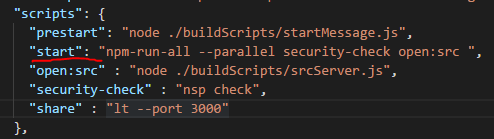
Quando lancio ad esempio lo start, se esiste un prestart verrà eseguito prima, poststart dopo

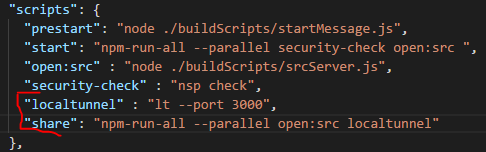




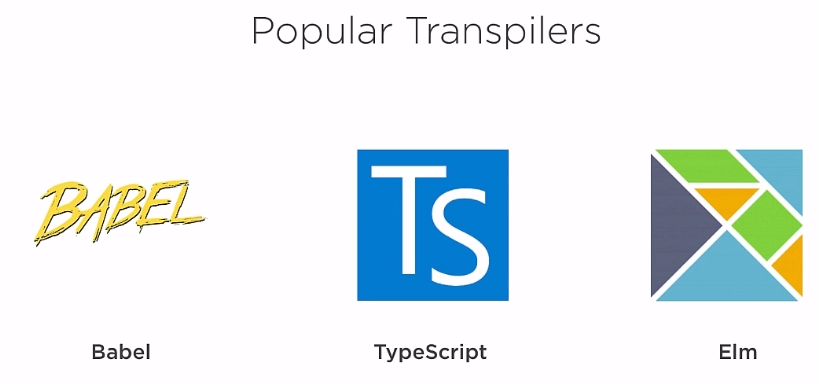
So if the nodemodule is in that folder I don’t need to install it globally.

Run concurrent tasks:



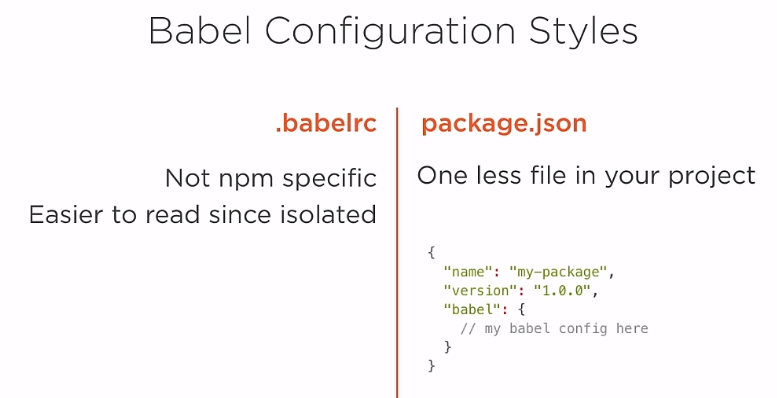


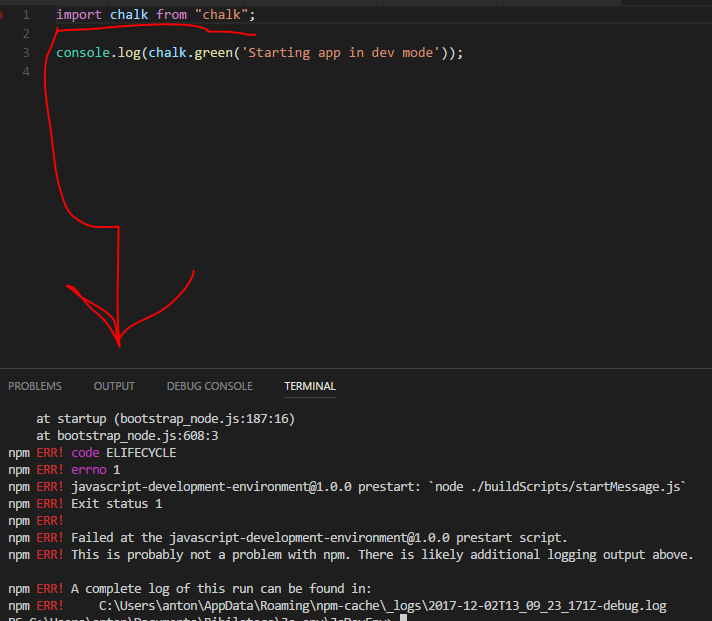
**TRANSPILING**



Babel has a mission: transpile the latest verion of js to es5.

TS has a superset of js



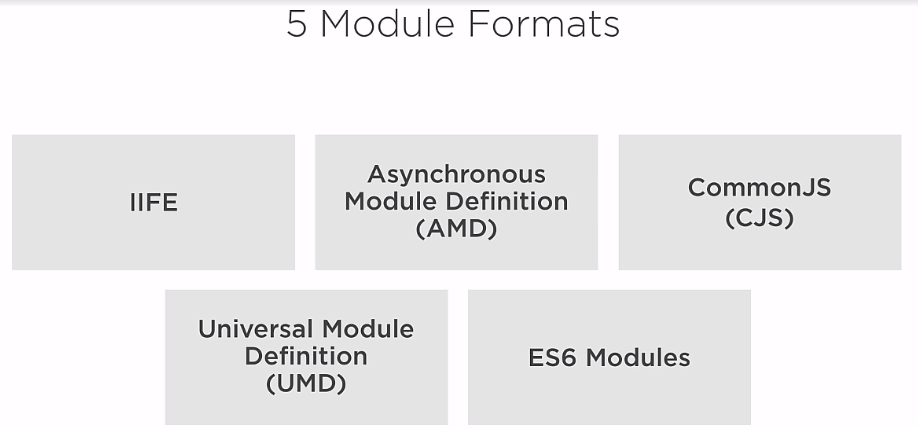


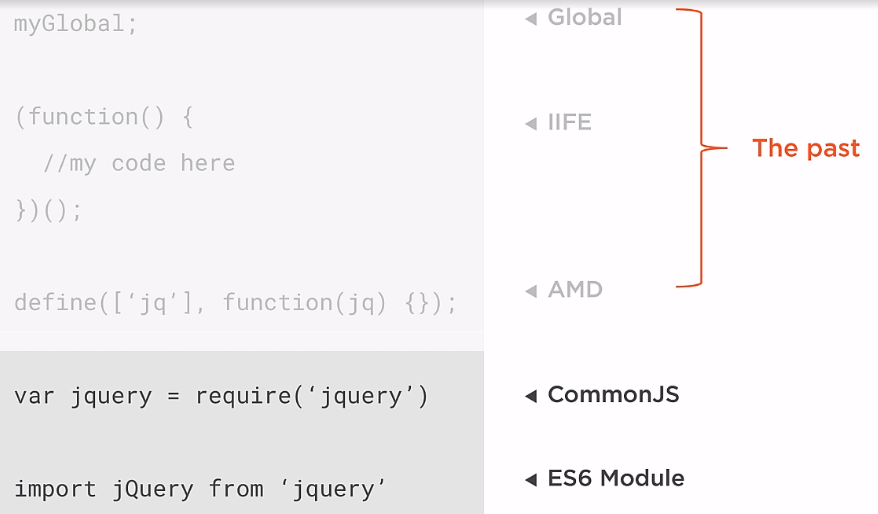
Perchè è versione nuova di js che node non capisce. Quindi posso usare babel:

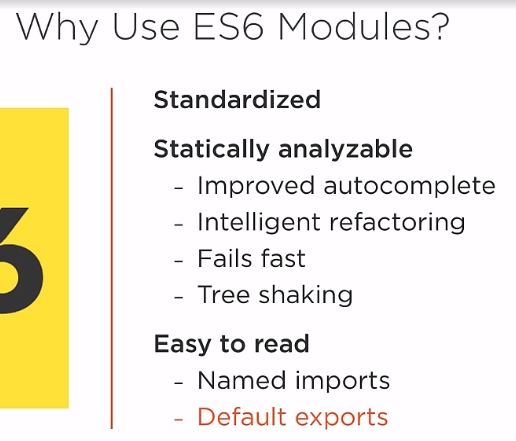


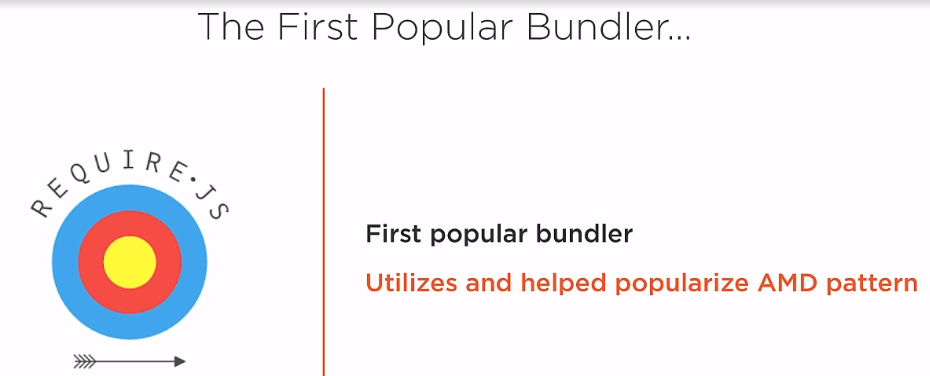
**BUNDLING**

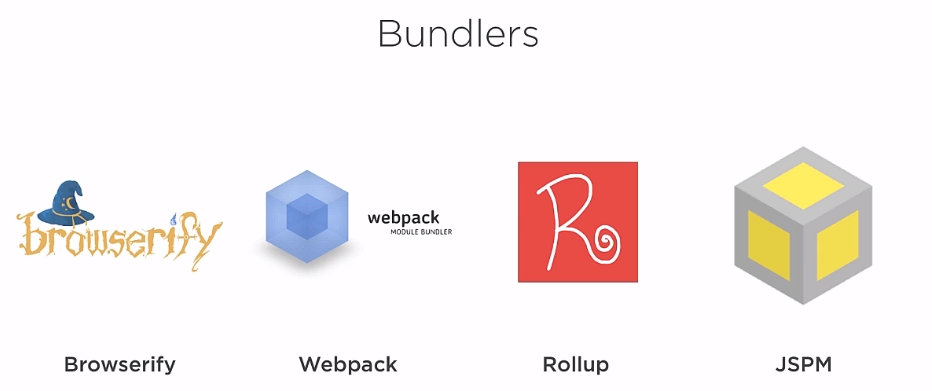
NPM packages use the commonJS pattern. node can handle this just fine, but browsers don't understand it. So you need to bundle NPM packages into a format that the browser can consume. But bundlers aren't just for apps that run in the browser. You may use a bundler to package any JavaScript into a single file. Or strategically into separate files, for different portions of your app. Imagine you've created an app with five separate pages. A powerful bundler can intelligently create separate bundles of JavaScript for each page. That way, the user only has to download the relevant JavaScript for the first page on initial load. This saves bandwidth and speeds page loads. Finally, remember that bundlers aren't just for the Web. You may want to use bundlers if you're coding in node as well, since node's require is slow. But bundling your code for node, you can compile away the require calls, which can often improve performance. So in this module, let's begin by considering the various module formats available in JavaScript, including AMD, CommonJS, UMD, and ES6 modules. Then we'll discuss the various bundlers to consider, like Webpack, Browserify, Rollup, and JSPM. And we'll close out this short module by implicating ES6 modules and bundling them up with our NPM packages via my suggested bundler, Webpack.



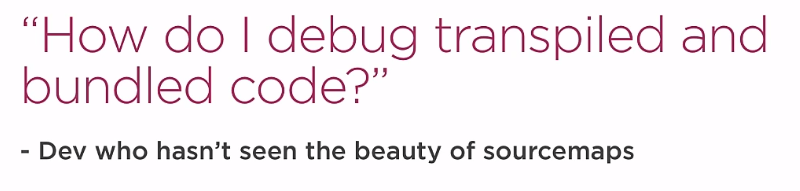


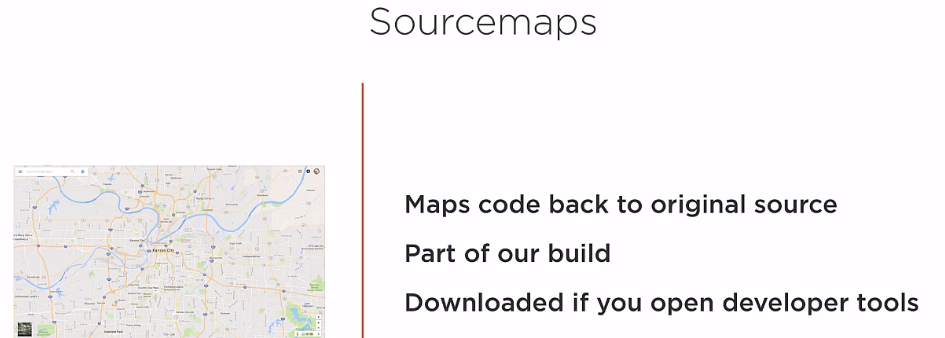


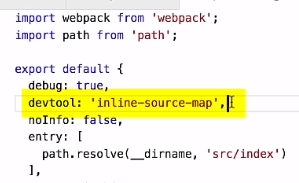












Se ne possono impostare diversi valutando il compromesso tra qualità e velocità.

