Contents

[Javascript starter kit: 2](#_Toc506579743)

[Javascrit editor – selection 2](#_Toc506579744)

[Package manager & its repository 2](#_Toc506579745)

[Install node for npm: 2](#_Toc506579746)

[security checks of npm 2](#_Toc506579747)

[Select dev webserver for the project: 3](#_Toc506579748)

[Configure express server in the project: 3](#_Toc506579749)

[Sharing working-in-progress to the client: 3](#_Toc506579750)

[Automation – integrate code – build – deploy 3](#_Toc506579751)

[Run – concurrent tasks 4](#_Toc506579752)

[Transpiling 4](#_Toc506579753)

[Configure Babel: 4](#_Toc506579754)

[Bundling the java script – make minified js file 5](#_Toc506579755)

[for debugging the bundled or minified js 5](#_Toc506579756)

[Linting tool 6](#_Toc506579757)

[Testing 6](#_Toc506579758)

[Continuous Integration 7](#_Toc506579759)

[HTTP calls 8](#_Toc506579760)

[Mock the HTTP requests 9](#_Toc506579761)

[Project Structure Tips 9](#_Toc506579762)

[Production Build 10](#_Toc506579763)

[Production Deploy 10](#_Toc506579764)

[Hosting in Heroku 10](#_Toc506579765)

[Surge UI deployment 10](#_Toc506579766)

[Update approaches update the projects 10](#_Toc506579767)

[Other ways of development projects 11](#_Toc506579768)

# Javascript starter kit:

Here start create a starter kit for a java script technology. This is just a starting point. Boiler plate code for the js. Node js is looked here as starter kit.

# Javascrit editor – selection

* Good editor should have ES2015+ support – auto complete, parse imports, refractoring
* Framework intelligence – **debugging** , different framework like react, ang js support, plugin support
* Inbuilt command line – check the application status in one place

Options:

* Brackets , webstorm, VScode, Atom – **famous**
* visual studio , eclipse, netbeans – this are famous server code editors. Can use as js editors.

It is nothing wrong to select different editors for front end and backend. So selecting **VScode** for its light weight front end capabilities

Recommendation:

VScode – many plugins available.

Use - .editorconfig for own editor intendation. Vscode require plugin for editor config

# Package manager & its repository

Every language has shared library, structured code . JS has number of package managers in recent times only.

* Bower , **npm** - **famous**  **b**ower once was very popular. But npm replaces it and it provides all the bower + other features. Now npm is common package manager repository which is very huge. Similar to maven in java.
* Other players – jspm , jam, volo

## Install node for npm:

Node js has inbuilt js running engine. It also has npm package manager by default. Use min version 4 for faster module dwn.

* Create package.json --- it is similar to pom.xml – have dependency libraries or packages used in project
* **npm install** in cmd line – dwn all the dependencies into the project based on package.json
* **npm init –yes** will create empty sample package.json
* here we downloaded package.json from [**http://bit.ly/jsdevpackagejson**](http://bit.ly/jsdevpackagejson)

## security checks of npm

npm can be published by anyone. So security checks are important. Two ways. **Retire.js** / **Node security platform .**

**npm start** will be good option which can be included before the development environment start. This will be little bit slow but dev environment it is ok. We can do manually also. However we might forget the security check of npm packages. Add automatic tasks of security check.

Manually check the package.json dependencies are security free.

Cmd> **npm install –g nps //** globally install nps and check the security of packages

Cmd**> nsp check**

# Select dev webserver for the project:

Famous dev options – **http-server**, live-server , highly configurable **production able** **express, koa, hapi** , hot loading budo/ webpack, browsersync(special feature dedicated ip allocation) and can be integrate with other servers.

**Choose Express – because it can be used in production also and powerful**

## Configure express server in the project:

1. in Package.json – make dependency for express
2. for building we create separate folder – **buildScripts** -> srcServer.js – configure the server
3. run in cmd -> node buildScripts/srcServer.js



## Sharing working-in-progress to the client:

Options – localtunnel, ngrok, surge, now

Share your localhost in public url so client can see the working in progress modules.

We select **localtunnel. Ngrok**  is secure so it is also can be used. **Now** – can upload to the cloud at 1 command. Surge only support static files but it is very simple.

Steps for localtunnel:

1. npm install localtunnel –g // into the project
2. start your app
3. lt –port 3000 // exposing to the public as local as webserver

# Automation – integrate code – build – deploy

Options – grunt -1st introduced, gulp- faster than grunt. It is code based than configuration. Like grunt plugin system , **npm scripts – we choose this as it provides other features also**

**Npm scripts –** leverage OS command line. Can run other scripts also. Simplicity unlike gulp as it needs plugin. Simpler debugging. Better docs. Easy learn then gulp.

1. declare in package.json

"scripts": {

"start": "node buildScripts/srcServer.js"

},

1. cmd> npm run start or simply npm start( run is not required for frequently using command)

we can also add many tasks we do manually which automate the stuffs.

## Run – concurrent tasks

Dependencies - "npm-run-all": "3.1.1",

Package.json --- this runs multiple tasks when starting the task

"scripts": {

"prestart":"node buildScripts/startMessage.js",

"start": "npm-run-all --parallel security-check open:src ",

"open:src":"node buildScripts/srcServer.js",

"security-check":"nsp check"

},

# Transpiling

History : ES1 1997 -> ES2 1998 -> ES3 1999 ->ES5 2009 ->ES6/ES2015 -> ES7/ES2016 -> ES8/ES2017

Transpilers - tools that read source code written in one programming language, and produce the equivalent code in another language.  Languages you write that transpile to JavaScript are often called **compile-to-JS** languages.

Options – hundreds of tools available. Popular are **Babel , TypeScript, Elm**

both babel & typescript are so good.

**Here Babel is chosen** for support of lot of features. Typescript also good. It is superset of JS actually which actually superset of EC6/7. Its their choice.

## Configure Babel:

2 ways - .babelrc / package.json -- we choose .babelrc approach to isolate this configuration

1. add dependencies to package.json
2. create .babelrc

{

"preset":[

"es2015"

]

}

1. we can now write js code with ES6 standards.

Eg: startMessage.js

~~var chalk = require('chalk');~~ import chalk from 'chalk';

but this is not understandable by node run. So change the node to babel-node in start script

"scripts": {

"prestart":"babel-node buildScripts/startMessage.js",

"babelbuild": "babel ./babelScripts/ -d ./buildScripts/",

}

"devDependencies": {

"babel-cli": "^6.26.0",

"babel-preset-es2015": "^6.24.1",

# Bundling the java script – make minified js file

Manifest into the single code. How you are writing the format of code. Java scripts developed in the node may not understand by the browser engines. Good practice to module the format. Maintenance of code to be easy.

Options – global , IIFE (function({///})), AMD (define([‘jq’],function(jq)))--- they are now considered past

Recommended standard options:

CommonJS – var jquery = require(‘jquery’); -- if using node.js can use the same

ES6Module **–** import jQuery from ‘jquery’

We choose ES6 Module. – standarlize based on official ES6 code, analyzable, intelligent refractor etc, logical way of writing.

Require.js – AMD is required.

ES6 Moduler **: options**

Browserify – 1st came simple and npm packageing js for web, webpack- bundles not only js. Css,images also, rollup- hot reloading , tree shaking (remove unwanted code so bandwith resuces) still new in market, jspM – uses systemjs to modules. Independent package managers.

**Here use webpack** – which is popular as of now and cover requirement. Bundle splitting dev/prod

**Configure webpack:**

webpack.config.dev – write config file with entry js and output bundle.js

srcServer.js – combine the webser with the **minified files**

we can inject the css styles also into them

## for debugging the bundled or minified js

generate sourcemaps. Which generated general files from sourcemaps.

Only generated if we open debugger. Otherwise wont generated.

webpack.config.dev

devtool: 'inline-source-map',

index.js – add debugger line

debugger;

# Linting tool

Enforce the consistency in the programs. Standards maintain. It avoid mistakes. Find errors in the program itself.

JSlint, JSHint, **ESLint --- it is more powerful and choose this**

Config files?: json,yml,…

We config inside the package.json

Which rules?: select from outofbox

Warning or errors?: errors will break the prm.

Plugins?: eslint-plugin-angular, eslint-plugin-react,eslint-plugin-node

Preset?: es recommended or new scratch or combined

Eslint-loader or eslint-watcher --- will be used

If you are using experimental babel feature – then use babel-eslint

Demo:

.eslintrc.json or inside package.json

**Bit.ly/jsdeveslint**

Add in the script – eslint watch plugin

“eslint”:”esw webpack\* src ” // recommended rules check in these folders

# Testing

Unfortunately, there are not much proper testing frameworks and continuous integration

**Framework** – **mocha** , jasmine, tape, Qunit for jquery, ava, jest from facebook,

**chai** assertion library

**Running in browser** :needs more config. Some common browser – karma, testem

Can also use headless browser – PhontomJS no visible interface

Can also use inmemory DOM – JSDOM ->fast

**Organize the testfiles**-> test folder in the source

**\*\*\*\*\*\*\*We select** – mocha – testing framework

Chai – assertion library

JSDOM – helper library – inmemory DOM options are useful

Where to run – in node itself instead of browser

Where to palce – testfiles inside the source code

When to run – whenever save the code

**Package.json**

Scripts”:{

\*\*\*\*\*\*

“test”: mocha --reporter progress buildscripts\testSetup.js \”src/\*\*\*.test.js\””

}// mocha will run the test classes located in src folder. Initial setup by testsetup.js. its progress is displayed in reporter.

**Command line**

* Npm run test // to run the script

**Src\index.test.js //** writing test file for index.js

Import {expect } from ‘chai’; // framework for assertion

Describe (‘our first test’,()=>{

It(‘shuld pass’,()=>{

Expect ((true).to.equal(true));

})

}

**Testsetup.js** // setup file to configure mocha

Require(‘babel-register’); // register to transpile before test. Mocha cant understand.

Require.extensions(‘.css’) = function(){}; // disable webpack that mocha don’t understand

**Note:**

**If index.html is to test** – jsdom is imported in index.test.js and write method

**Add scripts segment** - “test:watch”:”run run test -- --watch” // automate the testing during startup

# Continuous Integration

Few advantages of CI server -Ensure **code run cross platform**. Useful when multiple person involved in a particular modular. Can cover code coverage,tests,etc. automate the deployement. Can pass in local but can fail in CI server. ---- helps to identify the mistakes fast.

Options: travi, appveyer –windows supported, Jenkins ->all are popular others- circleci , semaphore, shapci

**Travis CI:**

<https://travis-ci.org/> /// can login with gitup account

enable the repository in the trais-ci page

**in project: add the following file**

.travis.yml

language: node\_js

node\_js: -"8"

when commit happens, it install the node 8 and run in the linux internal console of tervis.

**Note :** appveyer CI server also useful for windows based machines.

# HTTP calls

Libraries for making http calls. Will check the mocking the mocking http calls without browser.

In general Http calls – handled by

1. Node – use http or request library inbuilt in node
2. Browser – xmlhttprequest library if js runs in browser or jquery or framework ike angular or fetch api native support
3. Node & browser – isomorphic-fetch can handle http in both node/browser or xhr library, **axios** library(promise based), super agent

**Selecting fetch library** to handle http calls

srcServer.js //hard code the json value

app.get(‘/’,function(req,res){

res.json([{},{}])

})

Api folder -> userApi.js // make a folder and put it inside to centralize the http calls

Import ‘whatwg-fetch’;

Export function getusers(){

Return get(‘users’); // call our method get for the url

}

Function get(url){ // to get the http api

Return fetch(url).then(onsuccess,onerror); // todo onsuccess/ onerror methods

}

URl to call the api=> ./api/userApi

**Note:** fetch is native library. Some browsers wont know the fetch. To make unsupported browser to know fetch, use polyfill in index.html. chrome & all wont require but older ones need.

Index.html

<script src=<https://cdn.polyfill.io/......>

## Mock the HTTP requests

Useful for unit testing. Rapid prototyping. Work offline without connecting DB or connecting api.

Options – NOCK, static json, create dev server -> api-mock or json server or json schema faker

Json server serves static json everytime which saves the changes u made from pgm. Ie deleted will reflect in json server file also.

Here use – 1. json schema faker 2. Generate random data –randexp.js or faker.js(refer the site) 3. Serve data via API(json server)

1. Mockdataschema.js /// bit.ly/ps-mock-data-schema get the sample from here

Export const schema ={

}

1. Generatemockdata.js // to generate mock data each time using the schema

Import jsf from ‘json-schema-faker’; // random generate lib

Import {schema} from ‘’\mockDataSchema’;

Import fs from ‘fs’; // node inbuild library

Const json = JSON.stringify(jsf(schema));

fs.writeFile(“./src/api/db.json”,json,function(err)){

}

1. Mock api server creation

Package.json

Script{

“start-mockapi”:”json-server –watch src/api/db/json --port 3001”

It creates api service for many json objects – here localhost:3001/users & localhost:3001/home etc

This can be consumed in the app itself from Userapi.js

# Project Structure Tips

Include demo App in the project – useful for newer team members

1. Try to not use in html <script> why? -> you lose some power in js file. Transpiling with ES6, linting, resuse code, mocking
2. Don’t use dynamic generated js code snippets in net
3. Organise by filetype Vs organize by feature

File type organize => /component, /model, /views //commonly used.

Feature organize => /authors , /courses // but big projects it can be considered

1. Extract logic to POJOs --- similar to java pojo classes. For easy testing

# Production Build

1. **Minification**: to save bandwidth for application. Faster loading the pages

Removes whitespaces, newlines, shortern variable and fn names

Some framework – remove dead code or tree shaking.

Here used – webpack configured

Also compression is used to further reduce the size

1. Use **sourcemap** for debugging the minified js version also.

Here – dev source map is configured in webpack config. In prod, don’t use dev sourcemap

1. Dist server / srv dev server – separate them. In dev we mocked also for testing
2. Production build npm – clean the dist, recreate the dir – configure properly by webpack

Package.json: // include them for prod

Clean-dist:”rimraf ./dist && mkdir dist” // clean & recreate the folder

Pre-build:”npm-run-all clean-dist test lint”

Build:”babel-node build.js”

Post-build”:babel-node distserver.js”

1. **Dynamic generation of HTML –** html-webpack-plugin / manipulate via node / hardcode in html

conditional web pack plug in – useful dynamic add the scripts in html page

1. **Bundle splitting –** In prod, all the js should not load in 1 page itself. Necessary page in a page is required
2. **Cache Busting –** save js in browser itself cache so that repeated requested avoided
3. **Error logging –** trackjs, sentry, new relic etc

# Production Deploy

Till now we deploy in localhost. Hereafter in server.

1. Separating UI and API backend should be separated. Serve UI can also be served by content delivery
2. Hosting in cloud – aws, google cloud, azure, heruko, firebase. Only static(gethubpages, surge)

## Hosting in Heroku

* Visit the page. Signup . fork the github ready project –githun/couryhouse/js-dev-env-demo-api

Here only api code is deployed. UI will be separately deployed. Heroku based github files in already available. Slightly modified are given in above mentioned url

* Heroku command line – login and push the changes to github. It generates a new domain to your website.

## Surge UI deployment

Package.json

Deploy: “surge ./dist”

## Update approaches update the projects

1. Yeoman scaffold
2. github
3. npm – update the npm packages

# Other ways of development projects

For react js => Andrewfarmer.com/starter-project/

For angular => github.com/gianarb/awesome-projects

Just google – js framework boiler plate or starter plate or starter project or seed