# Starter Kit

## What Belongs in My Starter Kit

* Package Management
* Bundling
* Minification
* Sourcemaps
* Transpiling
* Dynamic HTML Generation
* Centralized HTTP requests
* Mock API framework
* Component libraries
* Development Webserver
* Linting
* Automated testing
* Continuous Integration
* Automated build
* Automated deployment
* Working example app (to convey how everything comes together including directory structure and file-naming practices).

# Editors and Configuration

## What to Look For in a JavaScript Editor

* It should show strong support for the latest JS features (ES 2017), but shouldn’t be too large.
  + To index JS files to provide **autocompletion**.
  + To **parse ES6 imports**
  + To **report unused imports**, typos and import statements.
  + To offer **automated refactoring** tools (rename, extract…)
* Built-in support for popular frameworks and libraries (Node, React, Angular…) [WebStorm, VSCode]
* Built-in terminal

## JS Editor Recommendations

* Atom
  + Free
  + Plug-in for EditorConfig
* WebStorm
  + 30-day free trial
* Brackets
  + Free
  + Plug-in for EditorConfig
* VSCode
  + Free
  + Fast
  + Git integration
  + Built-in terminal
  + Node debugging
  + Plenty of plug-ins (including EditorConfig)

## Automated Consistency via EditorConfig

Create a file named .editorconfig in the root of your project. Specify how your editor handles common settings (tabs/spaces, indent size, line feeds, char sets, trailing white space…)

# Package Management

## Package Managers

* Bower
  + Doesn’t require a build step.
* Jam
* JSPM
  + Bundles code.
* **npm**
* volo

## Security Scanning

* Node Security Platform
  + nsp check reports the number of vulnerabilities found.
* retire.js

### When to Run Security Check

* **Manually** is easy to forget.
* **npm install** is insufficient.
* At **production build** is too late and is expensive to change.
* At **pull request** is too late and is expensive to change.
* **npm start** means slow starts, requires a network connection, but notifies quickly when issue exists.

# Development Web Server

Except for Express, NONE of these web servers are for production.

* **budo**
  + Integrates with Browserify
  + Includes hot reloading
* **Browsersync**
  + Dedicated IP for sharing work on LAN
  + All interactions remain in sync!
  + Great for cross-device testing
  + Integrates with Webpack, Express…
* **Express**
  + Comprehensive
  + Highly configurable
  + Production grade
  + Can run it everywhere
  + For Node.js mostly
* **hapi**
  + Compelling configuration model
  + For Node.js mostly
* **http-server**
  + Ultra-simple
  + Single command serves current directory
* **koa**
  + Strong embrace of ES6 generators
  + For Node.js mostly
* **live-server**
  + Lightweight
  + Support live-reloading
* **Webpack dev server**
  + Built in to Webpack
  + Serves from memory
  + Includes hot reloading

## Sharing Work-in-progress

* **localtunnel**
  + Easily share work on your local machine
  + Easiest setup
  + Ultra-versatile
* **ngrok**
  + Secure tunnel to local machine
  + Easy setup
  + Secure
* **now**
  + Quickly deploy Node.js to the cloud
  + No firewall hole
  + Hosting persists
* **Surge**
  + Quickly host **static** files to public URL
  + No firewall hole
  + Hosting persists

# Automation

* **Grunt**
  + Configuration over code
  + Writes intermediary files between steps
  + Large plugin ecosystem
* **Gulp**
  + In-memory streams (pipes)
  + Fast
  + Code over configuration
  + Large plugin ecosystem
* **npm Scripts**
  + Declared in package.json
  + Leverage your OS’ command line
  + Directly use npm packages
  + Leverage world’s largest package manager

## Recommended scripts:

* start: to serve our site
* prestart/poststart: scripts to run before and after start
* security-check: for nsp check
* share: to run our development web server

Parallel scripts help to run multiple scripts at the same time.

# Transpiling

* **Babel**
  + Modern, standards-based JS, today
  + Write standardized JS
  + Leverage full JS Ecosystem
  + Use experimental features earlier
  + No type definitions, annotations required
  + ES6 imports are statically analyzable
  + Test, Lint, Babel, Great libs, IDE = safety
* **TypeScript**
  + Superset of JavaScript (type annotations)
  + Enhanced autocompletion
  + Safer refactoring
  + Clearer intent
  + Enhanced readability
  + Additional non-standard features
* **Elm**
  + Compiles down to JS
  + Clean Syntax
  + Immutable data structures
  + Friendly errors
  + All errors are compile-time errors
  + Interops with JS

## Babel Configuration

* **.babelrc**
  + Not npm specific
  + Easier to read since isolated
* **package.json**
  + One less file in our project

To avoid transpiling features unnecessarily, we can select one of the following plugins for Babel.

* babel-preset-es2015-node (version detection, Node-specific)
* babel-preset-latest-minimal (feature detection)

## Transpiling Build Scripts

* **ES5**
  + No waiting for transpile = faster
  + No transpiler dependency
* **Transpiled**
  + Enjoy the latest features
  + Consistent coding style
  + Use the same linting rules everywhere
  + Can eventually remove transpiler

# Bundling

npm packages use CommonJS. Node can handle it, but browsers don’t understand it. We need to bundle npm packages into a consumable format for browsers. Bundlers serve this purpose, but they can also…

* …package all our .js files into a single one, or into several ones.
* …improve Node performance.

## Module formats

Past

* Universal Module Definition (**UMD**)
* Immediately-Invoked Function Expressions (**IIFE**)
* Asynchronous Module Definition (**AMD**)

Modern

* CommonJS (**CJS**)
* ES6 module format
  + Standardized
  + Statically analyzable
    - Improved autocompletion
    - Intelligent refactoring
    - Fails fast
    - Tree shaking
  + Easy to read
    - Named imports
    - Default exports

## Bundlers

* **Browserify**
  + Bundle npm packages for the web
  + **Simple**
* **JSPM**
  + SystemJS, a universal module loader
  + Can **load modules at runtime**
  + Has its own **package manager**
  + Can install from npm, git
  + Uses Rollup
* **RequireJS**
  + Utilizes AMD pattern
* **Rollup**
  + **Tree shaking**
  + **Faster loading production code**
  + No hot reloading and code splitting
* **Webpack**
  + Bundles more than just JS
  + Import CSS, images…
  + Built in hot-reloading web server
  + **Comprehensive**

# HTTP Calls

Virtually every modern JavaScript application makes Hypertext Transfer Protocol (HTTP) calls.

## Approaches

### Node

* http
  + Built-in package
  + Low level library
  + Basic functionality for making HTTP requests
* **request**
  + Higher level library
  + Simple calls
  + Streamlined API

### Browser

* XMLHttpRequest (XHR)
  + Old
* jQuery
  + Logical way to handle HTTP calls
  + Avoids pulling in extra dependencies
* Framework-based
  + Many full-featured frameworks include their own HTTP service.
* **Fetch**
  + Streamlined API
  + Not supported by every browser
    - Browser polyfill (github.com/github/fetch)
    - Isomorphic polyfill (github.com/matthew-andrews/isomorphic-fetch
    - Polyfill.io sends a polyfill only to browsers that need it.

### Node & Browser

* isomorphic-fetch
  + npm package
  + Fetch-like API
* xhr
  + npm package
* SuperAgent
  + Plugin ecosystem
* Axios
  + Clean, promise based API

## Centralize API Calls

We must ensure API calls are handled in a single spot. It gives us…

* …one place to configure all our calls.
* …one place to handle preloader logic
* …one place to handle errors
* …a single seam for mocking

## Mock HTTP

Benefits:

* Quick and reliable unit testing
* Instant response
* Keep working when services are down
* Rapid prototyping
* Avoid inter-team bottlenecks
* Work offline

Ways:

* Nock
  + For unit testing
* Static JSON
* Create development webserver
  + api-mock
  + JSON server
  + JSON Schema faker
  + Browsersync
  + Express

## Planning for Mocking HTTP

1. Declare our schema
   1. JSON Schema Faker
2. Generate Random Data
   1. faker.js
      1. github.com/Marak/faker.js/wiki
      2. marak.github.io/faker.js/index.html
   2. chance.js
   3. randexp.js
3. Serve Data via API
   1. JSON Server

## Mocking Libraries

JSON Schema (json-schema.org)

# Project Structure

## Demo App

Because many people learn best by example, a working example is useful for the team.

A demo app serves as an example of…

* …directory structure and file naming.
* …framework usage.
* …testing scenarios, test naming conventions, test file placement, mocking strategies.
* …mock API.
* …automated deployment.

Demo apps codifies decisions, as they reflect our coding standards, we update as we learn new techniques and patterns we share.

Demo apps offer an interactive example of the starter kit working in a realistic scenario.

## Tip 1: JavaScript belongs in .js files

Never write JavaScript between <script> tags.

Doing so stops us from…

* …testing the code
* …linting the code
* …reusing the code
* …transpiling the code
* …importing explicit dependencies for our code

Never use server-side language to generate JavaScript. That code…

* …won’t have color.
* …won’t have autocompletion support.
* …will have runtime failures.
* …won’t have testing.
* …won’t have bundling.
* …won’t have minification.

Instead dynamically generate JSON instead.

## Tip 2: Consider organizing by feature

When working with small projects with MVC frameworks, organize by file type to place together files that serve the same purpose (components, data, models, views...).

For larger projects, organize by feature (authors, courses…).

## Tip 3: Extract logic into “POJOs”

Write as many logic in Plain Old JavaScript Objects (POJOs) as possible. POJOs do not have framework specific concerns, so it’s easy to migrate to other frameworks. Check React Slingshot starter kit on GitHub serves as example.