Max Planck Documentation

Modules Overview

All components of a module are listed below in the correct order.

Module01 – Rather

1. TitleScreen
2. (*n)
   - QuestionScreen
   - AnswerScreen
   - UserVotesScreen
   - ScoreScreen

Module02 – Manipula

1. TitleScreen
2. QuestionScreen (*n)
3. FinalScreen

Module03 – Predictor

1. TitleScreen
2. ChartScreen (*n)
3. FinalScreen

Module04 – Fake News

original: Rock 'n poll made with d3js version 3

1. TitleScreen
2. PollScreen (*n)

Module05 – Of What

1. TitleScreen
2. QuestionScreen (*n)
3. FinalScreen

Module06 – Compound

1. no repayment
   - TitleScreen
   - ConfigScreen
   - TimelineScreen (*n)
2. complete repayment
   - PreferenceScreen
   - TimescaleScreen (*n)

3. choose your rate
    - SettingScreen
    - TimebarScreen (*n)

Frontend Tool Chain

Introduction

That's what this is for: everything you need to get started with 'modern' frontend development with ES2015, rollup, sass and linters to keep your code sane and consistent. This is a highly opinionated, but easily extensible collection of tools for a nice workflow to create modern web applications.

Requirements

Make sure that you have a current version of node and npm installed on your system. Node is required for the whole build process and gulp.

- Install node — preferably with nvm or download an installer package
- Install npm — if npm is not installed via node (which it should be), run `curl -L https://npmjs.org/install.sh | sh` in your terminal or have a look here

After having installed node and npm, open a fresh terminal and check if everything is installed correctly by typing: `node -v && npm -v`, this should result in sth. like:

```
 v8.9.0
5.1.0
```

This package is tested with current node LTS (8.x) and should work without any issues.

Installation

Open up a terminal and type: `npm install`.

This will take a while, npm will install gulp (our build system of choice) globally and all dependencies for this starter kit. If something fails, try removing the node_modules-folder and run `npm install` again. On some systems you have to this with sudo, so try `sudo npm install` if it still fails.

IMPORTANT NOTE

Create a new virtual host pointing to the public-folder of this project, and start up your LAMP or nginx-

Stack. Creating a virtual host is the correct way to use this package.

Configuration

After installing, you should edit the file `config.js` and configure it to your liking, normally you just have to change two to three options:

- **proxy**: this should match your vhost on your local machine, and the vhost should point to the public-folder of this project
- **scheme**: if you want to use https instead of http, change this value
- **root**: this is the folder where you should point your webserver to
- **dest**: this is where you want all compiled and minified CSS and JS-Files to go (defaults to *./public/assets*, e.g. if you plan on using this with WordPress, the __dest__-variable could look like *./public/wp-content/themes/yourtheme/*)

There are a lot of other options in config.js, just have look at them and read the comments and change everything to your liking. You can change all paths, browser-support for autoprefixer, deactivate linters etc. pp.

Usage

You now should be all set up, to use it, just type `gulp` or `npm run` in your terminal and start coding. The default task starts the watch-task on all your files and you can code away. It also initializes the Browsersync proxy for fast and easy live-previews and remote debugging.

You should do all your work and save all files in the *./src/*-folder and the subfolders contained therein.

Your websites root-folder is the folder *./public/* — the plain HTML goes here.

If you need more information please have a look at the separate structure documentation which contains specific information regarding to preact and d3js.

All done? If you want to build a minified and production-ready version of your project, type `gulp build` or `npm run build`. This task checks for errors and creates minified versions of your CSS and JS, copies images etc.

For more documentation on tasks, conventions, building and other issues, please refer to the separate documention in the doc-folder, you will find a lot of information for the different aspects of this project:

- Conventions

- Workflow
- Gulp & Tasks Overview
- Sass
- JavaScript
- Images & Sprites
- Linting

Troubleshooting

- Make sure you have up-to-date and working installations node and npm on your machine.
- Remove the *node_modules*-folder, run `npm cache clean && npm install` and try again...
- Try to install gulp globally as sudo: `sudo npm install -g gulp`

This is not thoroughly tested on Windows and any linuxes except for ubuntu/debian.

Browser Support

IE<=10 is explicitly not supported, and the default styles and JavaScript don't add any support or fallbacks for this browser. Regarding everything else: it's up to you. The default assumption is: support the last 2 versions of the most used browsers, and the current version of browsers with a smaller marketshare. Have a look at *config.js* for autoprefixer-browser-support.

Known Bugs

When adding new files (JavaScript, Sass or images/fonts) you maybe have to restart the default gulp-task `gulp` (type `Ctrl+C ↑ ↲` in the terminal running gulp), so that the watcher recognizes the new files. This will hopefully be fixed with gulp4.

Structure

Introduction

Preact

Web: preact

> Fast 3kB alternative to React with the same modern API.

d3js

Web: d3js

> D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

Offline Version

An offline version exist for all modules which does connect to the API.

- module01
- module02
- module03
- module05

Basic HTML structure

```
<body>
  <section class="wrapper  wrapper__xx">
```

```
      <header></header>
      <main></main>
      <footer class="footer"></footer>
    </section>
  </body>
```

Wording

- **{Some}Screen.jsx**: file contains complete Sceen including header, main and footer
- **{Some}Item.jsx**: single part („*partial/snippet*“) of a screen
- **{Some}Helper.js**: global helper function

Folder structure

- **components/**: `.jsx`-files for each module as well as index / initial screen
    - **Module0{x}/**: module specific files
    - **shared/**: files which are used by multiple module
- **config/**: config data for each module (api urls, number of questions, ..)
- **content/**: offline content as well as static text content
- **helper/**: `jsx` helper files which could be used across all modules
- **modules/**: plain `js` modules, all d3js files will be found here
- **services/**: api connection related files
- *+vendor/**: 3rd-party content

Build Script

- `gulp build`: build final files
- `gulp`: enter watch modus (all modules)
- `gulp --file module02`: enter watch modus for module 02
- `gulp --file module02 --file module03`: enter watch modus for specific modules

Conventions

Coding Style & Linting

If your editor of choice supports editorconfig, you should already be fine, this project provides an .editorconfig-file that sets all conventions.

If not: 2 Spaces for *everything*, UTF-8 everything, Unix-Style linebreaks, no trailing spaces except for Markdown and text-files.

The JavaScript-, HTML- and Scss-linters will be enabled by default with the settings from above, so if you try to move away from those conventions: you will be warned :) The JavaScript- and Scss-linters will try to enforce a specific coding style regarding whitespace, naming stuff, usage of variables etc. to make your code consistent, so please try to stick to it.

More on configuring and using the linters

Filenames

Files should always be in lowercase, use dashes or double-dashes where applicable. No underscores, except as a starting character for sass-partials.

Watching and building

After having everything up and running (and configured to your liking) the main two gulp-tasks you will be using are:

- `gulp` — the default watch task for development
- `gulp build` — the 'make everything production-ready' task

For more information on the gulp tasks avaiable by default, have a look at Gulp & Tasks Overview.

Where should I put my stuff?

All your Sass, JavaScript, Images and SVGs that you want to use in your stylesheets and all fonts should go into their respective folders in *./src/*. Templates that are gonna be precompiled or used by gulp-tasks go in *./srcs/tmpl/*.

All those files are gonna be parsed/compiled/minified/concatenated/copied to the __dest__-Folder in public by the gulp tasks. The idea here is: keep all your sources in the src-folder, let gulp do the heavy lifting.

Everything else, that is HTML, CMS, PHP or whatever should be public (favicons, editorial images, documents etc.), should reside in the folder *./public*.

Tasks

Overview of all gulp tasks included in this package.

Composite Tasks

There are two main tasks, combining multiple other tasks. Those tasks are defined in *gulpfile.babel.js* - and those two ones are the ones you are gonna use the most:

gulp default (or 'gulp', 'gulp watch', 'npm start', 'npm run watch', 'npm run default')

This is the task you start when developing:

- Start a Browsersync server with your configured proxy
  It's live-reloading, remote-debugging for multiple devices on steroids. By default it starts the proxy on your local IP at port 3000, just have a look at the terminal-output after starting `gulp`. Use this address for developing, copy and paste it in your browser(s) of choice and start on any device on the same network.
- Start watchers for Sass, JavaScript, Images, Fonts and recompile/copy/recreate if something changes
- Activate linting for Sass and JavaScript — if there are any errors you will be notified in the terminal, but the task keeps on running and does not break.

gulp build (or 'gulp production, 'npm run build', 'npm run production')

The main task for creating a production-ready version of your sources:

- First: lint HTML for specified pages, lint all JavaScript and Sass. If anything contains errors, the build-task breaks. This way you are obliged to correct your errors before trying to create a production-ready version.
- Generate the SVG-Sprite and the corresponding Sass-file for the sprite
- And at last: create development + production versions for Sass and JavaScript, copy all fonts and losslessly copy and minify images, create a serviceworker-script and the shared config and copy everything to your dest-folder.

Single Tasks

Every other task (or task-group) can be found in the *./tasks/*-folder. Every file in there will be loaded automatically, this way you can just create a new file in *./tasks/* to add your own tasks for gulp.

Pre-configured tasks are (in alphabetical order):

clean

This task will remove all temporary files created and every file that will be compiled/copied.

copy

There are multiple tasks avaiable for copying files around:

- copy:fonts — copies all woff and woff2 files from src to dest
- copy:images — copies all image files from src to dest
- copy:vendorjs — copies any vendor javascript-files to dest (e.g. large already minified libs)
- copy:serviceworker — copies the serviceworker script and sets new versions/hashes for the chache
- copy:shared — copies the shared.json file to your root-folder
- copy:loadcss — copies the loadCSS script to your dest/js-folderand minifies it

critical

This taks will create a critical.min.css file in your dest/css folder. This CSS file will only contain the critical 'above-the-fold'-rules from your homepage. The idea here is: include this file directly (inline) in your html on the homepage, and use the loadCSS function to load the full main.min.css on the first visit, and do so only on the homepage. For an example see index.php in the public-folder.

js:development and js:production

Those tasks will produce the main.js and main.min.js files, using rollup as a bundler. The non-minified version produced by the js:development-task contains a sourcemap for easy debugging, the one generated by js:production will be minified with uglifyjs.

lint

There are three different types of tasks avaiable for linting your html, sass and js-files, and for each on of those tasks, there is a :development and a :production-flavor — the development-tasks don't break your build (useful while watching files), whereas the :production-ones will throw an error and kill the build-process:

- lint:js:development — lints all your js files, except those in src/js/vendor, does not break
- lint:js:production — lints all your js files, except those in src/js/vendor, will break build if there are any errors
- lint:sass:development — lints all your scss files, except those in src/scss/vendor, does not break
- lint:sass:production — lints all your scss files, except those in src/scss/vendor, will break build if there are any errors
- lint:html:development — this task will try to download all pages specified in config.pages, and then lint those for errors, does not break
- lint:html:production — see above, but will break
- lint:development — this task combines the three development-linting tasks just described
- lint:production — this task combines all three linting tasks and will break, this one is used in gulp build first

sass:development and sass:production

The task sass:development is used in the default gulp watch task. This task will recompile sass and add a sourcemap for easy debugging. This task creates the unminified, fully expanded *main.css*-file and tells Browsersync to reload, if anything changes.

sass:production creates the minified *main.min.css* file, without any sourcemaps.

Both tasks use the gulp-autoprefixer plugin to automatically add vendor-prefixes after sass-compilation.

savehtml

This task downloads all pages specified in config.pages and saves them as temporary files to your public root-folder. This task is used in critical (for getting the HTML of your homepage) and in the lint:html-tasks (for checking your html).

serve

Starts a browsersync-instance and is used in the default gulp-task.

shared

This task creates a shared.json-file from the specified scss-files from config.shared. Only basic sass-maps and variables are supported. This is just a simple regexp-task, converting scss-files (with only variables in it) to one json-file, which then can be used with JS.

sprites

This task uses the gulp-plugin gulp-svg-sprite to create a simple sprite from all SVG-files, that you copy to the *./src/img/sprites/*-folder. This tasks generates a sprites.svg-file (the actual sprite) and a scss-file in *./src/scss/vendor/_sprites.scss*.

The generated sprite is very basic (and at the core just a pimped png-sprite — you still have to create every icon with every color) but can easily be used with the generated mixin. You can change the file *./src/tmpl/sprites.scss* to influence the scss-output.

This is a work in progress, and when we can finally ditch IE9-support, I will revisit this (or use sth. like grumpicon instead…)

Sass

Sass in this project will be compiled with the plugin gulp-sass, using node-sass. This means we don't have to install the rubygem for sass anymore. This also means: have a look at the compatibility table if you want to use any edge-features from ruby-sass.

Everyime I say Sass - I mean the language and Sass with the scss-syntax. I prefer the scss-syntax, because it is easier for people who already know CSS. Curly braces FTW!

Sass Guidelines

Hugo Giraudel wrote an awesome piece on everything you need to know about Sass, it's called Sass Guidelines and you should really have a look at it. I agree with this guideline in almost all points, but I try to keep something more simple, and some things more strict, the linter will let you know :)

Structure

There is one main-file, where everything is included (mixins, modules, configurations etc.) in the order you specify. By default it's *./src/scss/main.scss*. This will be compiled into one single CSS-file, which you can then use on your website. I still prefer this approach over multiple files that need to be loaded on different pages (as long as it's a small website), because once the main stylesheet is loaded, it is cached. The only issue here: be careful when supporting IE9 or lower, because may run into the selector limit issue

All other files are structure in the following folders:

- base — styling for all basic elements, aka HTML-Elements. Defaults for inputs, tables, images, headings etc.
- config — global variables, maps, breakpoints colors etc. Some things get their own map and helper-function (in tools)
- modules — the main folder. This is where all the modules go, and almost everything is a module. If you want to, you can split this up in different folders and create sth. as proposed in the Sass Guidelines with a folder *layouts*, *themes*, *pages* etc. for stuff like your grids, headers, wrappers etc. and custom page styles — but I like to keep it simple: everything is a module, except:
- tools — all custom mixins and functions go here. There are a lot of helper functions out-of-the-box
- vendor — other 3rd-party styles, e.g. the generated sprite-map etc.

Conventions

The general rule here would be the same as in JavaScript:

One module, one file, one declaration

BEM

This project uses the oldschool BEM-Style (Block, Element, Modifier) to describe your modules, and with Sass this can look like:

```
.your-module {
  display: inline-block;

  &--with-fancy-hair { // a modifier
    color: $color-special;
  }

  &__sth-inside { // an element inside of .your-module and styled in relation to it
    float: left;
  }
}
```

Whereas the one-module-one-declaration means: one module should only describe one selector or variants of it, but it would be absolutely OK to write the above example as, and throw it in one file:

```
.your-module {
  display: inline-block;
}

.your-module--with-fancy-hair { // a modifier
  color: $color-special;
}

.your-module__sth-inside { // an element inside of .your-module and styled in relation to it
  float: left;
}
```

… especially when things get longer.

Sass Linting

The pre-bundled sass-lint task comes with pretty strict settings, regarding naming of things, sort-order of properties and various other possible problems. Please just try to stick to it, to keep things consistent.

Vendor prefixes

Just don't write any, unless you are absolutely sure and want to target that specific browser. Vendor prefixes are added as needed by autoprefixer. To change supported browsers have a look at *config.js* in the root folder and change browsersupport-array accordingly.

Defaults

By default, this project uses REM as units for fonts, assumes a mobile-first strategy, IE10+ usage of CSS3-features and tries to set sane margin/padding defaults by defining a base-unit.

Have a look at *./src/scss/config/_sizes.scss* - this is where the `$base-unit` will be defined and then used throughout the project, especially the mixins for positioning and spacing. You can then use them like this:

```
.some-element {
  @include spacing(b 2);
  // ==> margin-bottom: $base-unit * 2;

  @include spacing-inner(l 1/2, r 1/2);
  // ==> padding-left: $base-unit / 2; padding-right: $base-unit / 2;

  @include absolute(t 1, l 3, r 50px);
  // ==> position: absolute; top: $base-unit; left: $base-unit * 3; right: 50px;
}
```

By using rem, you can scale your website easily — just set the font-size on the html/body elements for different breakpoints if you want to.

I highly recommend scanning through everything in config and tools, a detailed documentation on all mixins and how to use them will follow.

Used 3rd-party tools

This project by default includes normalize.css and some resets and helpers I wrote myself or combined from other projects. Those are all installed via npm and are included in main.scss by default.

Structure

Install every module/library/script/tool/helper — whatever you want with npm. This project uses rollup with some plugins to make all those awesome tools usable for your browser/frontend. Then just use the new ES2015-modules import-syntax and you are all set. Have a look at the included modules for fontloading and using breakpoints in your JavaScript.

For JavaScript it's the same as with Sass: there is one main-file, that will be loaded as the entry-point for all your JavaScript, include all your other modules from there.

A very basic sample is provided in the folder *./src/js/*.

Your own modules

The recommended way would be to create your own modules, document them and throw them on npm. If this is not feasible, just create a file in *./src/js/modules/* with the name of your module, for example *somemodule.js*

To keep things consistent throughout the project I would recommend one simple rule:

One module, one file, one function

Meaning: every module should get its own file (e.g. in */src/js/modules/somemodule.js*), the module should have the same filename as the exported function, and there should be only ever one function, that said module exports.

With ES6 this could look like this:

```
// use ES2015 modules to import other stuff you might need
import stuff from 'othermodule';

// those two vars will be used in your module, but won't be exported
const wontBeExported = () => { ... };
const wontBeExportedEither = 12;

// that's the function this module will export
export default () => {
  // do stuff...
  return 'sth'; // if needed
};
```

If you have to export multiple, similar functions that should live in one module, you can do so with the new ES6-modules, but I would advise to stick to the pattern above, and then just do sth. like:

```
export default () => {
  ...

  // using new ES6 Enhanced Object Literals to export multiple functions
  return {
    funcOne () {
      console.log('fuck yeah');
    },
    funcTwo () {
      console.log('this is nice');
    }
  };

};
```

This way you can throw similar functions into one module, and use it like this:

```
import someModule from './modules/somemodule';

someModule().funcOne();
someModule().funcTwo();
```

The idea is: don't make me think. And this way I just have to remember that I will always get **one** function if I import sth.

Keep your modules as short, concise and reusable as possible and refactor all the time :)

Recommended ES6-Features

Since ES6 is here to stay: use it. You still can write 'normal' JavaScript, but I really recommend to check some of the new features out. Though there are features that are not well supported, I recommend only those, that can be easily transpiled with babel.js and don't require a polyfill:

- arrow functions
- let and const
- ES6 modules (import + export)
- Default + Rest + Spread
- destructuring
- literal object creation
- template strings
- classes (if you really need those over basic $.extend or _.assign...)

For more information refer to the Learn ES2015 page on babeljs.io. If you want to use 'advanced' features like WeakMap etc., you have to use the polyfill from babel.

If you want to use promises this package includes es6-promise to include a fallback for IE<=11.

Plugin, library polyfill Recommendations

For general "somebody wants a simple website with carousels and stuff" I hereby recommend some plugins to save some time:

- jQuery, because after all: you might need jQuery ;)
- svgxuse: a polyfill to make the SVG-sprites work in IE (included)
- Carousel/Slider: slick, the last carousel you'll ever need
- Lightbox: Magnific Popup or the older colorbox
- Native Video: video.js or mediaelement
- Though I still recommend jQuery, I don't recommend jQuery UI. It feels outdated, and there are a lot of smaller, better plugins for stuff like datepickers for example: pickadate
- For all your functional needs and the occasional debounce/throttle: lodash
- In case you use responsive images, use picturefill

For an insane list of awesome stuff, have a look at those two lists:
https://github.com/sorrycc/awesome-javascript and https://github.com/sindresorhus/awesome.

Images

All images should go into *./src/img/*, be it SVG, PNG, GIF or PNG. You can use those images directly in your Sass as background-images with sth. like `background-image: url('../img/imagename.png');` - if you don't change this project's folder structure.

(That is for images you want to use in your templates and stylesheets directly. I often differentiate between two types of images: the ones you use and create for your stylesheets (those should go into *./src/img/*), and the ones editors can add as content — those images are not part of this build-chain.)

SVGs you want to create a sprite of should be copied in *./src/img/sprites* and be given a meaningful name. The sprite can be included directly in HTML like this (be sure to change the path to your assets/dest path):

```
 <svg class="icon  icon--arrow-left">
   <use xlink:href="assets/img/sprites.svg#icon--arrow-left"/>
</svg>
```

Default styling for all icons is provided via *./src/scss/modules/_icons.scss*, and you can then style your icons via CSS, size them, change the fill color etc. Including SVG via fragment-identifiers is still not supported in IE and buggy in Safari, so be sure to include the svgxuse polyfill, that comes pre-installed with this package. For more information on how to use the SVG sprites, refer to this article: Cloudfour: Our SVG Icon process.

Feel free to change the sprites gulp task, if you need multiple sprites.

Recommendations

Always try to use the right image-format for the thing you want to do, and compare their sizes to one another. Sometimes a JPG is better than a PNG regarding image size and quality; and sometimes it is the other way around. For everything that looks like a vectorgraphic: use SVG :)

Try software like imageoptim and imagealpha to further reduce the size of your images — they can sometimes achieve better results than the (no longer included) gulp-imagemin-plugin.

Linting

Repeat after me: "linting your code keeps you sane". This project provides three different linters: one for HTML, one for Sass (.scss-files to be specific) and one for JavaScript.

All those linters can be configured in their set of rules, and they can be globally deactivated in *config.js*, section *lint*.

HTML: htmlhint

The linter used for your HTML-Code is htmlhint. This project comes with a pre-configured set of rules to enforce nice HTML-styling, those rules and their current setting can be found in the file *.htmlhintrc*, description for the rules can be found on github.

If you use a CMS or any other tool that creates your HTML-Output, it sometimes makes more sense to just deactivate linting for HTML or change some of the default rules. But please always check your code for errors with tools such as the W3C Validator to keep yourself sane. Valid HTML (at least regarding nesting and quotes) can save you a lot of headaches when styling your website.

JavaScript: eslint

The linter used for JavaScript is eslint, and this project comes with a pre-configured (kinda restrictive) set of rules, to enforce a consistent and sane code style. You can find all current rules and the settings for them in the file *.eslintrc*.

The predefined *.eslintrc.yml*-file sets all rules you can find in the eslint documentatoin, in the same order for easy reference.

Eslint is very customizable, you can write your own plugins, but for starters the default built-in rules are more than enough.

ES6 / ES2015

Since we use ES6 in this project, eslint is configured to warn you on some things, that can be written better with ES6 (such as arrow-functions or let vs. var). If you don't want to use any ES6-features at all, you can just set all those rules to zero, to deactivate them.

Sass: sass-lint

For linting our Sass we use sass-lint. Configuration for sass-lint can be found in *.sass-lint.yaml*, and is pre-configured to be quite strict. All avaiable rules for sass-lint can be found here. By default anything in the folder *./src/scss/vendor* will **not** be linted, so you can add third-party stylesheets easily.

As always: try to keep to the strict defaults and only in edge-cases change the default rules and allow exceptions (such as `!important`).

Tools

Recommended Tools & Helpers

Those tools are not part of this package, but I really would recommend that you have a look at them :)

- frontend-md — Generate simple documentation for your frontend code
- npm-check-updates — find and save the latest versions of dependencies, regardless of any version constraints in your package.json
- caniuse-cmd — a caniuse.com command line tool, check features in your terminal
- stylestats — evaluate your generated stylesheets for size, number of selectors etc.
- css-colorguard — checks your CSS for too similar colors
- notes — a small tool for checking all your code for TODO, FIXME, OPTIMIZE and NOTE comments

You can install them all via npm, give them a spin!