Hermanni Piirainen

Optimizing web development workflow

Metropolia University of Applied Sciences

Bachelor of Engineering

Media Technology

Bachelor's Thesis

1.11.2016

Tekijä Otsikko Sivumäärä Aika	Hermanni Piirainen Verkkokehityksen työnkulun optimointi 39 sivua + 3 liitettä 1.11.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Toimitusjohtaja Tapio Nurminen Lehtori Ilkka Kylmäniemi

Insinöörityön tavoitteena oli tutkia moderneja JavaScript-pohjaisia työkaluja ja tekniikoita verkkokehityksen eri vaiheiden optimoinnin näkökulmasta. Työssä selvitettiin myös Node.js-ajoympäristön toimintaperiaatteita. Node.js toimii lähes kaikkien työssä esiteltyjen työkalujen pohjana. Työn tarkoituksena oli vertailla perinteistä ja modernia verkkokehityksen työnkulkua aina projektin aloittamisesta lopputuotteen siirtämiseen tuotantopalvelimelle sekä sitä, mihin ongelmiin uudet työkalut yrittävät löytää ratkaisuja.

Perinteisessä työnkulussa suuri osa työvaiheista sisältää manuaalisia, toistuvia tehtäviä, joiden nopeampaan ja tehokkaampaan suorittamiseen sellaiset työkalut kuin npm ja Gulp pyrkivät. Insinöörityössä tutkittiin ja vertailtiin työkaluja, joiden avulla verkkokehitystä voidaan optimoida ja automatisoida. Vertailtaessa työkalut jaettiin kolmeen osa-alueeseen: verkkosovellusten riippuvuuksien hallinnan työkaluihin, tehtävänsuorittajiin ja projektin aloittamista tukeviin työkaluihin.

Insinöörityön lopputuotteena valittiin SPA-arkkitehtuuria (single-page application) noudattavalle verkkosovellukselle sopivat kehityksen eri vaiheita tukevat työkalut. Valitut työkalut olivat kehitysriippuvuuksien hallintatyökalu npm, front-end-riippuvuuksien hallintatyökalu Bower ja tehtäviensuorittaja Gulp. Työkalujen käyttöä varten luotiin myös kokoonpano, jonka avulla toistuvien tehtävien manuaalisesta suorittamisesta päästiin eroon. Kokoonpanon avulla valitut työkalut ja niiden toiminnot saatiin lisäksi integroitua: esimerkiksi Bowerin avulla hallinnoitavat front-end-riippuvuudet saatiin koottua yhteen tiedostoon Gulpilla suoritettavan tehtävän avulla. Gulp-tehtävien avulla saatiin lisäksi aikaan muun muassa Node.js-kehityspalvelin, joka mahdollisti sovelluksen kehittämisen ja testaamisen reaaliaikaisesti monilla eri laitteilla, ja sovelluksen tuotantoversion automaattinen kokoaminen sekä siirto tuotantopalvelimelle muutaman komennon avulla.

Insinöörityössä huomattiin, että työkalujen kehitystahti on erittäin nopea: uusia työkaluja luodaan viikoittain. Kehityksen mukana pysyminen vaatii vaivannäköä, mutta tärkeintä ei ole opetella työkalujen eroja, vaan ymmärtää niiden konsepti ja periaatteet sekä se, mihin ongelmiin ne kehittäjälle tarjoavat apua.

Avainsanat	verkkokehitys, optimointi, Node.js, npm, riippuvuuksien hallinta, tehtävänsuorittaja

Author Title Number of Pages Date	Hermanni Piirainen Optimizing web development workflow 39 pages + 3 appendices 1 November 2016
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Tapio Nurminen, CEO Ilkka Kylmäniemi, Lecturer

The goal of this final year project was to research different JavaScript-based tools and techniques from the perspective of optimizing the different phases of web development. Node.js, a JavaScript runtime environment, and its principles were also examined in this project, as it serves as the basis for most of the tools introduced in the project. The purpose of the project was to compare the traditional and modern web development workflows, from scaffolding a new project to transferring the application to the production server, and to research which problems the modern tools are trying to solve.

In the traditional workflow most phases include manual, repetitive tasks, that tools such as npm and Gulp are trying to run faster and more efficiently. Tools that provide optimization and automation to web development were compared in the project. These tools were divided into three sectors: dependency management tools, task runners and scaffolding tools that help start a new project.

The end result of this project, namely, a collection of tools suitable for developing a single-page application was selected. The selected tools were the development dependency management tool npm, front-end dependency management tool Bower and task runner Gulp. A setup eliminating the need to running different recurring tasks manually was created for these tools. The functionalities of these tools were also integrated, for example the front-end dependencies managed by Bower were collected and concatenated to a single file with a Gulp task. Other Gulp tasks provided among other things a Node.js development server, with which it was possible to develop and test the application with multiple devices in real time, and a way to build and deploy the production version of the application with just a few commands.

During this research it was noticed that the progress of modern development tools is extremely fast, as new tools are being introduced on a weekly basis. It takes effort to try to keep up with the progress, but the most important thing is not to learn the differences of each new tool, but rather to understand the underlying concepts and principles of those tools, and which problems are they trying to solve.

Keywords	web development, optimization, Node.js, npm, dependency management, task runner

Table of contents

1 Introduction

The world of front-end development has experienced major changes during the last years. Gone are the days of manually downloading files, and large, rigid and all-encompassing software tools. One of the principal catalysts for the changes has been Node.js, a JavaScript runtime environment that has enabled creating all sorts of JavaScript-based front-end development tools. With the rising of Node.js and its counterpart, Node Package Manager, the emphasis for modern front-end tools has switched from creating a single, monolithic tool to handle all aspects of development, to small modules or plugins that concentrate on solving little problems, and solving them well [1, 2].

Older, graphical tools have mostly been replaced by Node.js-based command line tools, that provide better and more universal configuration options, and allow developers to reuse and share their configurations for better integration. With these modern tools developers can optimize and automate parts of all phases of the development workflow. New projects can be initialized with scaffolding tools, front-end or development dependencies can be installed and managed with dependency management tools, and development tasks can be automated with task runners, all within seconds or minutes.

With the right tools and a functional setup for the development workflow, the developers can focus on creating better code and solutions for the web application, rather than manually running repetitive tasks that the computer is capable of doing more efficiently. The purpose of this research is to examine Node.js and the different tools and techniques it enables, that help optimize and automate as many aspects of the development workflow as possible. This research aims to find a practical front-end development setup for a customized single-page web application. The goal is to optimize all phases of the workflow, from creating a server for developing to deploying the final version of the application to the production server.

2 Comparing web development workflows

Finding the right workflow is crucial for a successful web project. The process of finding the right setup, though, is not simple. With the multitude of tools available and with new tools being created daily, it can be hard to determine the best tools, especially because the projects vary so vastly, and there is no one-fits-all setup that will work for every project and every developer. [3] The web development workflow is all about trying different tools and coming up with the setup that works for the project at hand.

In a traditional web development workflow for example installing and maintaining third party packages and frameworks would include first searching for a web page that offers a downloadable version of a package, such as jQuery. If the package was updated, the developer would need to download and replace the old version of the package with a new one. This process would have to be done for each package individually, and when packages are downloaded manually, there is no good way of knowing if and when a package has been updated. Downloading and transferring the package files manually is time-consuming, especially when there are dozens of packages to be maintained. The process of checking for updates and downloading them might be so tedious that the updating process is left neglected, and the application will run outdated code. [4] This example describes the problems of the traditional workflow, the main question being "why do a task manually, when a computer can do it faster and better?".

For some developers and projects, an old-fashioned workflow might work well, especially if the project does not involve any modern development tools or languages such as Sass (Syntactically Awesome Style Sheets), a powerful extension for the CSS language. In projects that use for example only plain HTML and CSS, there might not be any repetitive tasks that need to be run with short intervals, like compiling Sass into CSS. But almost all projects can profit from optimizing or automating at least some parts of the development process, whether it be optimizing images for smaller file size or minifying assets such as CSS files for production usage.

Zell Liew divides a normal web development workflow into six processes:

- scaffolding
- developing

- testing
- integrating
- optimizing
- deploying [3].

All of these processes can be optimized with the right tools and techniques. Finding the right tools is necessary but not always easy because of the multitude of similar but still different tools and packages. Many tools with a graphical user interface (GUI), such as CodeKit and Hammer, have been created with which developers can minify files and automate other tasks as well. GUI tools, as they are visually intuitive, may feel easier to adopt, especially for beginners, compared to the text-based command line interface (CLI) tools. Tools with a GUI are usually more rigid and only provide customizing options to a certain extent The GUI tools have limitations that become visible especially when a project involves more than one developer. All of the other developers working on a project might have different operating systems and preferred GUI tools for developing. These tools can have highly different functionalities, and developers can end up in situations where integrating their code with others becomes tricky, or if a tool does not have a functionality that is needed, they might have to change the tool to a completely different one [5].

CLI tools can often defeat these problems, as they are more flexible, highly configurable and most of the time work across all platforms and operating systems. The configuration options for these tools can then be shared with other developers working on the project, which makes integrating the code easier, as everyone working on the project can use the same set of tools and configurations regardless of their operating system. Command-line based tools usually come with a steeper learning curve, so they require some time to get used to them. However, because of their configurability and the ability to share and reuse the configurations with other developers and projects, command-line tools are definitely worth learning.

3 Node.js and npm

Multiple technologies have been created in order to aid developers achieve their goals quicker and easier and to provide universal solutions to the problems the traditional workflow has had. One of the technologies that has enabled these modern tools for workflow optimization is Node.js, an efficient JavaScript runtime environment that runs on top of the Google-developed V8 JavaScript engine [6].

Client-side JavaScript is executed in the user's web browser, very much like the two other client-side scripting languages HTML and CSS. Client-side JavaScript handles processing and responding to user input and making requests to the server. Client-side – generally known as front-end – JavaScript has been popular since its birth [7, p. 2]. Node.js though runs server-side or in the back-end, which means it takes care of responding to client-side requests and serving the correct content to the end-user. The field of server-side scripting has traditionally been controlled by scripting languages created solely for back-end purposes, such as Python and Ruby. Since the launch of Node.js it has become a significant competitor to these more traditional server-side languages.

Node.js has a strong ecosystem around it, meaning it is actively developed and maintained by a large amount of people. A large part of the Node.js ecosystem stems from its modularity. A massive amount of packages, also known as modules, has been written for Node.js. Packages allow the core language to be kept simple while maintaining a comprehensive set of functionalities. Packages are small Node.js programs that add features for the developers to use along with the core language. In its simplest form a Node.js package is only a directory containing files. [8] Currently there are 292 322 packages written for Node.js, and the average growth is 402 new packages per day [9].

Node.js comes shipped with npm, which stands for Node Package Manager. Basically npm is two things: a huge registry which holds information and metadata about all of the Node.js packages people have published, and a command line tool used for searching, installing and updating these packages [8]. All npm packages must include a configuration file called *package.json*. This file includes all of the information about the package, such as version number, keywords, license and repository information. It also specifies all of the files that should be downloaded when a user wants to install this package. [10]

3.1 Event loop

The main advantage of Node.js is its non-blocking I/O (input-output) model. This means that all code is processed asynchronously, whereas in traditional programming the system halts while waiting for a response from the server. Java and other server-side languages have tackled this problem by implementing multithreading to allow multiple processes to run concurrently [7, p. 3]. A thread is an independent process separated from the main program. The main problem with multiple threads is that they share memory, so extra precautions are needed when multiple threads are accessing a single variable. [11, p. 2; 7, p. 3] When simplified, Node.js always has a single thread, but it can still run concurrent processes. Concurrency in Node.js is accomplished by its event loop. Node.js uses an event-driven model, where events and state changes trigger callback functions. Callback functions allow Node.js programs to traverse through code lines without pausing to wait for responses or spawning multiple threads for processing. Whenever an operation returns something, the result can be processed by a callback function. [12]

Several other programming languages also include the concept of an event loop, but all of them incorporate it via external libraries. The event loop in Node.js is part of the language core and the loop does not need to be initiated by a blocking start call like for example in Python's Twisted engine. [6; 13] Node server will simply begin waiting for events as soon as the server is started, without an additional call [14].

Figure 1 demonstrates a normal life cycle of an event in the Node.js event loop. An event, for example a user input, occurs and is passed to Node.js event queue. Each event in the queue enters the event loop, which then delegates the event to a thread pool. Although all of the Node.js application's JavaScript runs on a single thread, a thread pool is still needed for some tasks. When a result is returned from the thread pool, the callback function is passed back to the event queue. The event queue and thread pool in Node.js are managed by a library called libuv, which means that the developer does not need to and cannot access the thread pool. [15] The event-driven and non-blocking model give Node.js a considerable speed advantage when compared to more traditional server-side scripting languages [16].

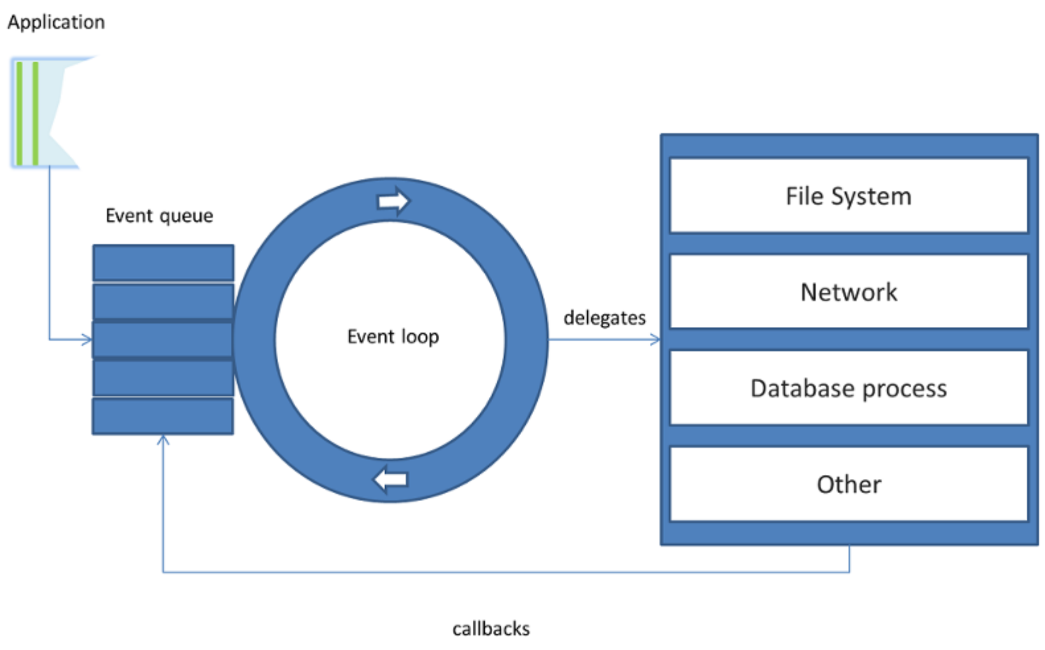
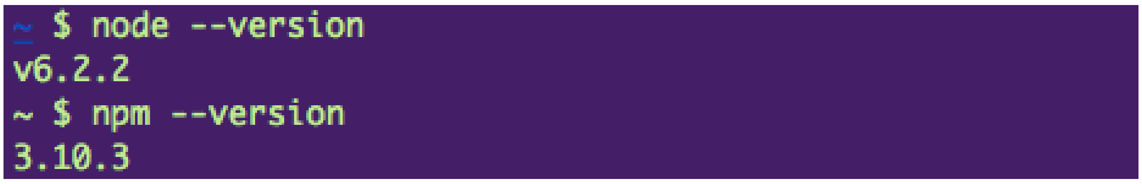


Figure 1. Node.js event loop [17].

3.2 Installation process

Node.js can be installed on a machine either by downloading the Node installer from the official download page https://nodejs.org/en/download/ or by using a package manager from the command line, such as Homebrew for Mac OSX or Yum or APT for Linux operating systems. After the installation process is complete, success of the installation can be verified by executing *node --version* and *npm -v* via the command line. These commands tell Node.js and npm to print out their current versions. The flags -*-version* and -*v* both have the same meaning, with -*-version* being more verbose and easy to understand. An example of a successful installation verification can be seen in figure 2.

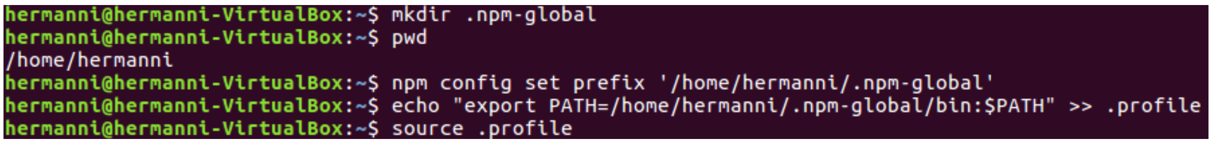


```
~ $ node --version
v6.2.2
~ $ npm --version
3.10.3
```

Figure 2. Verifying a successful installation of Node.js and npm.

Npm is used via the command line. New packages can be installed either globally or locally, depending on the use case of the package. Packages should be installed globally whenever the tools need to be used via the command line, since globally installed packages' executable files can be used from any directory. This is possible, because installing packages globally places the module files and executables into the *PATH* environment variable, for example into directory called */usr/local/.* [18] The location of the executable file of a program can be printed by executing *which <program>*, for example *which node*. In most cases accessing the default directory requires administrative permissions, which means that installing packages globally via npm would require using the *sudo* command. With *sudo* user can execute commands as the superuser [19]. Using *sudo* is not recommended when installing npm packages, a better solution is to configure npm to use a different location for installing packages, a location where no administrative permissions are needed [20].

In figure 3 the path for npm is changed from the default location to the user's home directory. A new directory called *.npm-global* is created on the first line, and on the second line current working directory is printed. *Npm config* command is an npm-specific command that can be used to view or change the npm's configuration variables. The third command in the figure changes the *prefix* variable to point to the newly created directory. This command ensures that all things npm-related will be installed into this directory. If a new global package is installed at this point, the package will be installed into the correct directory. This means that using the packages executable file should be possible from any directory. Trying to use the executable will not work, since the new path variable is not yet part of the system-wide path. This step is shown in figure 3 on the fourth line. This command will add the new path into the existing path represented with *$PATH*, so that when executable files are executed, the system will know to look from the defined directory as well. The last command in figure 3 runs the script file *.profile* into which the new path was added. After these steps when running an executable file of an npm-installed global package from any directory, the system should print out the path that was set as the new npm prefix. [21]

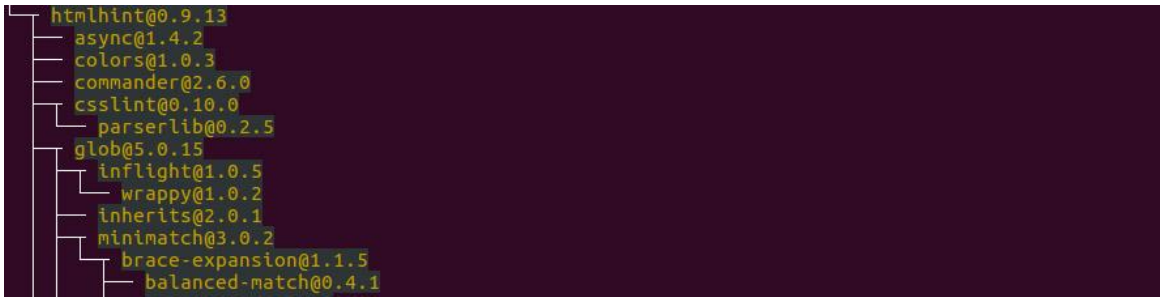


```
hermanni@hermanni-VirtualBox:~$ mkdir .npm-global
hermanni@hermanni-VirtualBox:~$ pwd
/home/hermanni
hermanni@hermanni-VirtualBox:~$ npm config set prefix '/home/hermanni/.npm-global'
hermanni@hermanni-VirtualBox:~$ echo "export PATH=/home/hermanni/.npm-global/bin:$PATH" >> .profile
hermanni@hermanni-VirtualBox:~$ source .profile
```

Figure 3. Changing the path of npm.

3.3 Managing modules with npm

Npm has many built-in commands, the most used of them being *npm install*. Global packages are installed with flag *-g* or *--global*. [22] Running for example *npm install -g htmlhint* will not only install a package called *htmlhint* but it will also install all the packages that *htmlhint* is depending on. Those sub-dependencies can also have their own dependencies and so on, this is called a nested dependency tree. An excerpt of a nested tree can be seen in figure 4. All packages below *htmlhint* are sub-dependencies of that package. Multiple depths of dependencies can be seen in this figure, for example package *balanced-match* is on the fifth level of this dependency tree.



```
└─┬ htmlhint@0.9.13
  ├── async@1.4.2
  ├── colors@1.0.3
  ├── commander@2.6.0
  ├─┬ csslint@0.10.0
  │ └── parserlib@0.2.5
  ├─┬ glob@5.0.15
  │ ├─┬ inflight@1.0.5
  │ │ └── wrappy@1.0.2
  │ ├── inherits@2.0.1
  │ ├─┬ minimatch@3.0.2
  │ │ └─┬ brace-expansion@1.1.5
  │ │   ├── balanced-match@0.4.1
```

Figure 4. An excerpt of the nested dependency tree of a package.

Dependencies in a flat dependency tree model all exist on the same level. It is therefore simpler in its design. A simplistic example of a flat dependency tree can be seen in figure 5. The application in the figure depends on three modules: A, B and C. Dependency C also depends on dependency A. This may cause problems, if the developer decides or has to update the version of dependency A to be able to for example utilize a new functionality that only the newer version of the package provides. Updating dependency A means that dependency C must also be compatible with the newer version of dependency A. If dependency C is incompatible with the newer version, it creates a dependency conflict. Being unable to update modules because of conflicts is called dependency hell. [23]

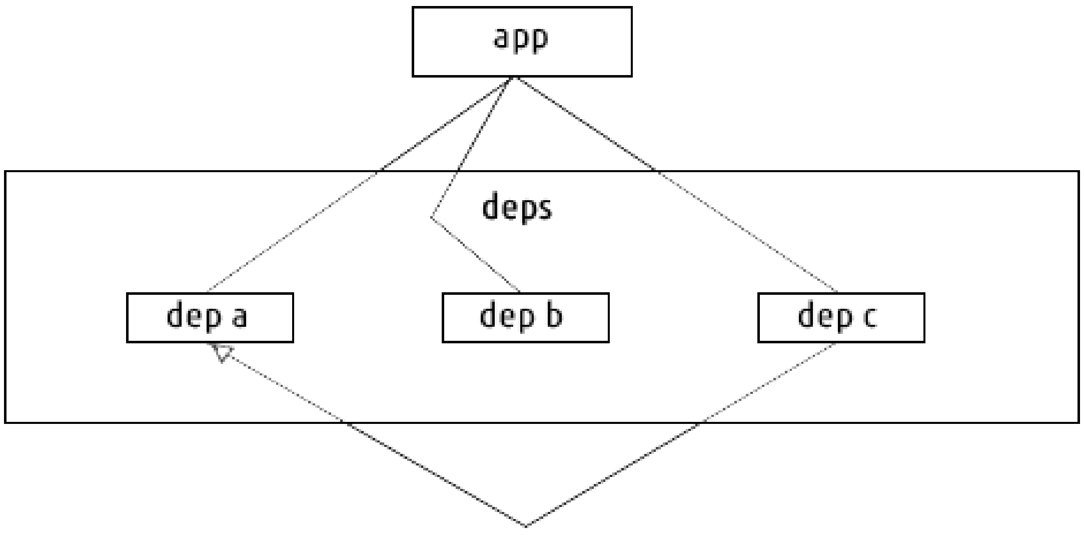


Figure 5. An example of a simple flat dependency tree [23].

Nested dependency tree model solves this problem by allowing each module to install their dependencies separately. If the tree in figure 5 was nested, it would mean that dependency C would install a separate version of dependency A so that the application could then depend on a completely different version of dependency A, and could be updated without conflicts. The design of a nested dependency tree is more complicated the more dependencies the application has, and there might at some point be multiple different versions of the same module to serve the needs of all of the modules. This solution does of course increase the needed disk space, but the fact that modules are usually small in size and not having to resolve conflicts are the things that make the nested dependency tree model superior in most cases. [23]

Modules in npm can be uninstalled with the command *npm uninstall*, which has all the same flags that the *install* command has. The nested dependency tree model allows the developer to only state the name of the main module that is no longer needed, npm then checks if each of the main modules own dependencies can be uninstalled as well. All installed packages can be listed with *npm list*, which prints out all of the installed packages' names and versions. The *list* command can be configured to only show an individual level of the dependency tree with an optional flag --*depth*, for example *npm list --depth=0*. Modules can also be searched from the npm registry via command line with the command *npm search keyword*. Npm will then return a list of matching package titles and descriptions. Additional info about a certain package can be fetched with the command *npm view package*. For example, *npm view htmlhint* will show all of the information about *htmlhint* package that is stored in the npm registry. Field names can be added to the command to only show data from those fields, for example *npm view htmlhint author repository.url*. Packages that are installed and have newer versions available in the registry can be listed with *npm outdated*, and they can be updated with the command *npm update*, for example *npm update htmlhint -g*. Npm also provides a few optimizing commands, *npm prune*, which removes unused packages and *npm dedupe*, which traverses through the dependency tree and tries to flatten the tree by removing duplicate packages. For example, if dependencies A and B are depending on the exact same version of package C, the tree can be simplified by moving C up the tree and uninstalling the duplicate packages from under both A and B. [22]

4 Workflow optimization processes and tools

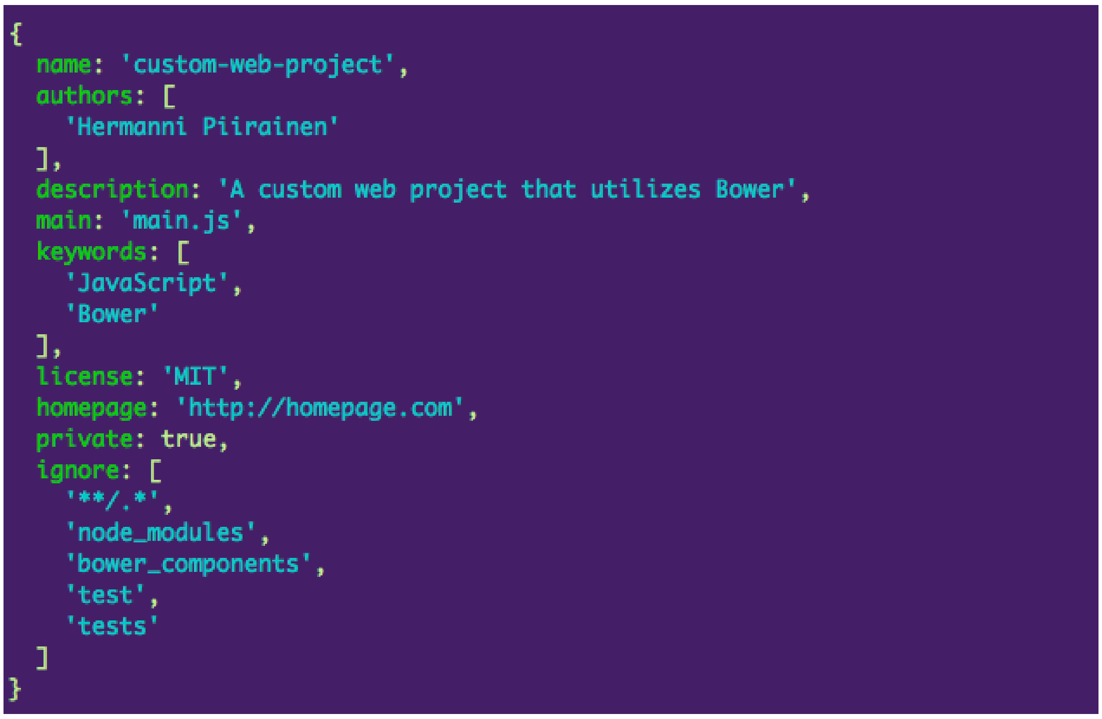
4.1 Dependency management

Most web applications have at least some dependencies such as front-end libraries like jQuery or Angular. Downloading front-end libraries manually for each project can be repetitive and time-consuming, let alone updating them. The traditional way of adding a new front-end dependency would be to first open the library's official website and downloading the source. Then the source needs to be copied to a folder containing all of the assets, and a link to the JavaScript or CSS file needs to be added to the HTML file accordingly. Updating an existing dependency would also add manually checking for updates and removing the old version of the library to the process. [24]

Bower is a tool developed by Twitter in 2012 that specializes in front-end dependency management. It basically downloads and updates libraries and other assets with a simple command-line API, but it also helps integrating project dependencies with other developers. Bower is managed with a project-specific configuration file called *bower.json*, that resembles the *package.json* file that npm is using. When a developer wants to reuse the same front-end dependencies as in another project or share project dependencies with other developers, only *bower.json* needs to be shared. Bower has to be installed globally so that its commands can be run in every directory. [25]

Bower is installed with npm, so in order to use it, Node.js and npm must be installed. Bower also requires the command-line tool *git* to be installed, since it uses Git repositories in GitHub or similar services to download the assets. For Linux, *git* can be installed with Yum or APT. For Mac and Windows users *git* can be installed by installing GitHub for Mac or GitHub for Windows as *git* is included in them. Optionally Windows users can download it from Git website, and Mac users can install Xcode Command Line Tools application, which also includes *git*. After installing requirements, Bower itself can be installed with npm just like any other global package: *npm install bower -g.* Installation can be then verified with *bower --version*. [25; 26]

Each project that uses Bower needs to have the manifest file *bower.json*. This file can be created either manually or automatically by moving to the project root directory and running *bower init*. *Bower init* launches an interactive dialog which allows modifying each field to the project's needs. An example file created with *bower init* is seen in figure 6. Most of the objects in the file are really needed only when the developer wants to register a package to Bower in order for other developers to be able to use it as a dependency in their own projects.



```
{
  name: 'custom-web-project',
  authors: [
    'Hermanni Piirainen'
  ],
  description: 'A custom web project that utilizes Bower',
  main: 'main.js',
  keywords: [
    'JavaScript',
    'Bower'
  ],
  license: 'MIT',
  homepage: 'http://homepage.com',
  private: true,
  ignore: [
    '**/.*',
    'node_modules',
    'bower_components',
    'test',
    'tests'
  ]
}
```

Figure 6. Bower.json after running *bower init*.

Most important part of *bower.json* is the *dependencies* object. All of the application's dependencies will be listed inside that object. Bower commands closely resemble those used by npm. Suitable libraries and assets can be searched with *bower search* command, for example *bower search jquery*. Installing a package is done with *bower install package --save*. The flag *--save* writes the dependency into the *dependencies* object of *bower.json*, which helps identifying and maintaining the installed assets and reusing same project configuration. Bower installs all dependencies into a single root directory, called *bower_components* by default. The name and location of the components directory and other options can be changed with a JSON file called *.bowerrc* [27]. Bower allows users to install libraries simply with a registered package name such as *jquery* or with a Git endpoint or an absolute URL. Sometimes also registering local assets such as customized or bundled script files is necessary. This is also possible with Bower by providing *bower install* a path to the local asset file. Semantic versioning or semver is used by Bower so that developers can have better control on which version of the library they want to install. Semantic versioning consists of three parts, for example 1.2.21 where the major version is 1, minor is 2 and patch is 21. Installing an exact version of an asset can be done with *bower install package#1.5.3*, but Bower also supports semver ranges. Ranges allow developers to state for example that a version greater than 1.5.3 but less than 1.6.0 is needed with the command *bower install package#~1.5.3*. [28]

Some examples of different types of installs with Bower are seen in figure 7. On the first row a registered package *jquery* is installed with a semantic versioning range. This range represented by a tilde character would translate to greater or equal to 1.11.0 but less than the next major (2.0.0) or minor (1.12.0) update. Replacing the tilde with a caret would only prevent updating to next major updates [28]. Second row installs a git package with exact versioning of 1.2.30. Exact versioning will always prevent updating until the versioning is changed. The third command in the figure uses an explicit versioning range. This command will install the latest version inside the specified range. In the last command a local JavaScript file *app.js* is registered as a Bower component with the name "appJS" and therefore it is copied to *bower_components* as well.

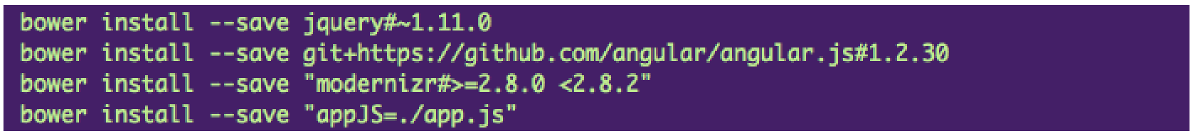
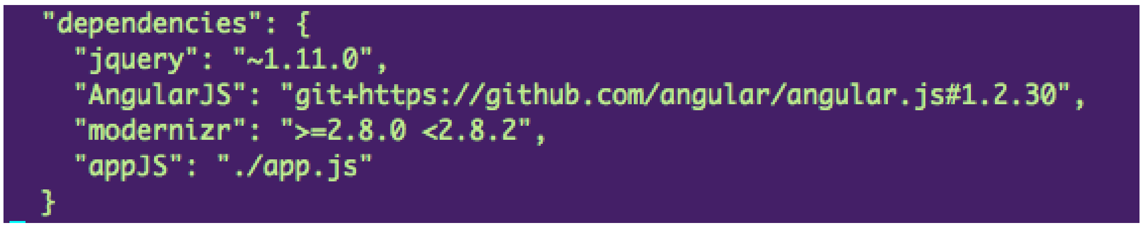

```
bower install --save jquery#~1.11.0
bower install --save git+https://github.com/angular/angular.js#1.2.30
bower install --save "modernizr#>=2.8.0 <2.8.2"
bower install --save "appJS=./app.js"
```

Figure 7. Different ways to install Bower dependencies.

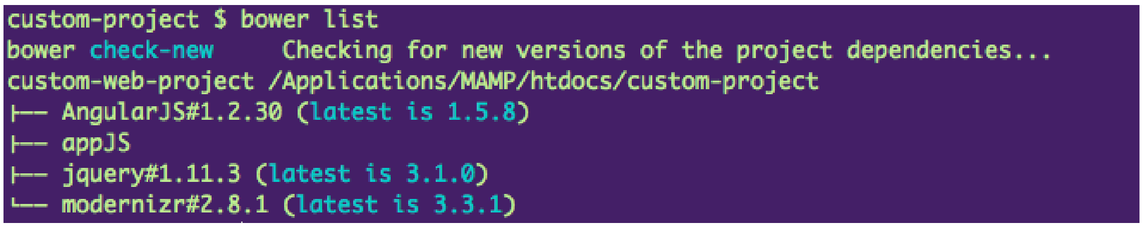
Figure 8 shows the resulting *dependencies* object in *bower.json* configuration file. Each dependency is represented as a key-value pair, where the first part, the key, is the name of the dependency and the second part, the value, is either the version or the combination of URL and version of the dependency. Assets are called by their name in Bower commands, so executing *bower uninstall --save appJS* would remove the local asset *app.js* from both *bower_components* and *bower.json*. When re-using or using a shared *bower.json* file for a new project, all of the dependencies stated in the *dependencies* object can be installed with a single command *bower install*. If a version control system such as git is used in a project, the best practice is to only commit the manifest file to the repository, so the whole *bower_components* directory is omitted in order to keep the repository simpler and smaller in size [29].



```
  "dependencies": {
    "jquery": "~1.11.0",
    "AngularJS": "git+https://github.com/angular/angular.js#1.2.30",
    "modernizr": ">=2.8.0 <2.8.2",
    "appJS": "./app.js"
  }
```

Figure 8.   Bower.json dependencies object resulting from commands in figure 7.

The *dependencies* object only shows the semantic versioning that was set for each package, but command *bower list* shows the explicit versions that were installed with the specified version ranges. Resulting versions of the commands in figure 7 are seen in figure 9. In the figure it can for example be seen that jQuery version 1.11.3 was installed since it is the latest version in the 1.11.x range.



```
custom-project $ bower list
bower check-new     Checking for new versions of the project dependencies...
custom-web-project /Applications/MAMP/htdocs/custom-project
├── AngularJS#1.2.30 (latest is 1.5.8)
├── appJS
├── jquery#1.11.3 (latest is 3.1.0)
└── modernizr#2.8.1 (latest is 3.3.1)
```

Figure 9.   Result of *bower list* command showing the actual installed versions.

Bower resolves all dependencies to a flat dependency tree, which means that only one version of each package can be installed in a single project [25]. The flat dependency tree is usually not a problem for projects with few dependencies considering possible conflicts can be resolved manually. For very large projects with dozens of dependencies where three or more dependencies creating conflicts with each other, the flat tree model might be an insuperable issue. [30]

For most modern web applications npm is usually already involved in the project. In addition to being a dependency manager for tools needed just for development, npm can also serve as a front-end package manager. According to Jaakko Salonen, using Bower in a project that already has npm configured is redundant [30].

Each package registered in npm registry must have a *package.json* file stating its dependencies, and so does each project that is using npm. Basically every project is therefore like a single package that has its dependencies and other information written in its own *package.json* manifest file, with the exception that it will not be published to the npm registry. A new *package.json* file can be created interactively with the command *npm init.* The process of creating the file and the end result is almost identical to the process of creating a *bower.json* file.

Project-specific front-end dependencies are installed locally instead of globally to allow different projects to have different versions of the same packages, and as npm uses a nested dependency tree, a single project could also include several versions of the same dependency. To install npm packages locally, *npm install* must be executed without the *--global* or *-g* flag. This will install the package into a directory in the project root directory called *node_modules*. Although it is possible to just download packages with npm, in almost all cases it is reasonable to also save all installed dependencies to the manifest file. Adding flag *--save* to *install* command will add the dependency into the *dependencies* object in the *package.json* file, again similar to the process of installing a package with Bower. As semantic versioning is nowadays a de-facto standard, version ranges can also be used with npm. There are no big differences whether npm packages are installed globally or locally, the same commands concerning global packages apply to both use cases.

The main differences between the two major dependency managers are the different tree models they utilize and the processes used to wire the dependencies into the project's source code. Both tree models have their pros and cons. Bower's flat tree reduces duplication and is easier to comprehend, but as a downside can sometimes cause conflicts when the project has multiple dependencies. Npm's nested tree allows multiple versions of the same dependency so therefore conflicts are rare, but may also bloat project size.

With both tools, developers can choose their favored style of injecting the dependencies into the source code. One option is to simply use the file paths directly, for example <*script src="bower_components/jquery/jquery.js"*>, but this approach is not optimal since it requires developers to manually add a script tag for each new dependency and check each package's *package.json* file for which files inside a package are to be added to the source code and in which order. With Bower it is recommended to use individual tools that automate injecting correct files in correct order such as Wiredep or to automate the wiring process with task runners such as Gulp or Grunt.

4.2 Task running

Developing, building and deploying a web application includes a lot of different tasks, most of them repetitive. Let's consider a normal workflow when using a CSS pre-processing language like Sass. Common tasks when using Sass might include

- compiling Sass into CSS, that browsers can interpret
- adding browser-specific vendor prefixes
- styling the CSS against certain formatting rules
- minifying the compiled CSS file.

All of the tasks listed above require either using browser-based solutions or installing a separate command line tool for each task. These tools are usually created for specific purposes, for example *cssnano* is a JavaScript-based command line tool that minifies CSS files. Running these tasks without a task runner requires executing the commands for each task separately. Some tools can be easily configured to automatically watch for file changes and automatically run the task whenever a change is made, but setting a tool to watch for file changes requires an ongoing command line process that cannot be interrupted for it to work. The workflow described above may feel like it doesn't take so much time that it would need to be automated. But if for example CoffeeScript, an extension language to JavaScript, is used in the project, CoffeeScript files also need to be compiled into JavaScript and the resulting JavaScript files formatted, validated and minified. When compiling HTML from HTML templating engines such as Handlebars or Jade, or optimizing image assets are added to the workflow, the whole process can get very repetitive and complicated very quickly.

This is where JavaScript-based task runners come in handy. They combine all of the tasks that a developer needs to do over and over again into a single configuration file. Using task runners makes the configurations of different tasks easier to understand, maintain and share. They also alleviate the problem of remembering the correct command syntaxes for each different command line tool. Modern JavaScript task runners have similar functionalities and use cases as older task runners like Make or Rake, but they are designed specifically for automating front-end development tasks.

The most popular JavaScript-based task runners are Grunt and Gulp. Both of them – like all JavaScript task runners – rely on the multitude of small packages and wrappers called plugins. Plugins are created by the tool maintainers as well as the large community of developers using the tools. Normal workflow of adopting a task runner in a project consists of installing the task runner itself, installing plugins that are needed in the project, creating the configuration file and configuring the tasks.

The basics of Grunt and Gulp are the same, but they also differ in a few things. Grunt can be thought of as configuration-based whereas Gulp is more code-driven [31]. They also have differences in the way they handle the files when running tasks. Grunt is file-based as it relies on temporary intermediary files written between each task. This means that each task uses an input-output disk operation. [32] Configuring two tasks that are chained, for example compiling a Sass file into CSS and adding vendor prefixes, requires first compiling the Sass file *main.scss* into a temporary file on disk, for example *tmp.css* and then passing the temporary file to the *autoprefixer* task to compile the final CSS file *main.css*. Gulp is stream-based as it uses Node streams that can be piped together and work in-memory, removing the problem of using temporary files [31]. Writing and reading files to and from the disk is usually more time-consuming than using streams, so Gulp has a speed advantage over Grunt [33]. Zander Martineau's speed comparison tests show that same tasks in Gulp are at least twice as fast as in Grunt [34]. These differences also result in major distinctions in the syntaxes both of these tools use.

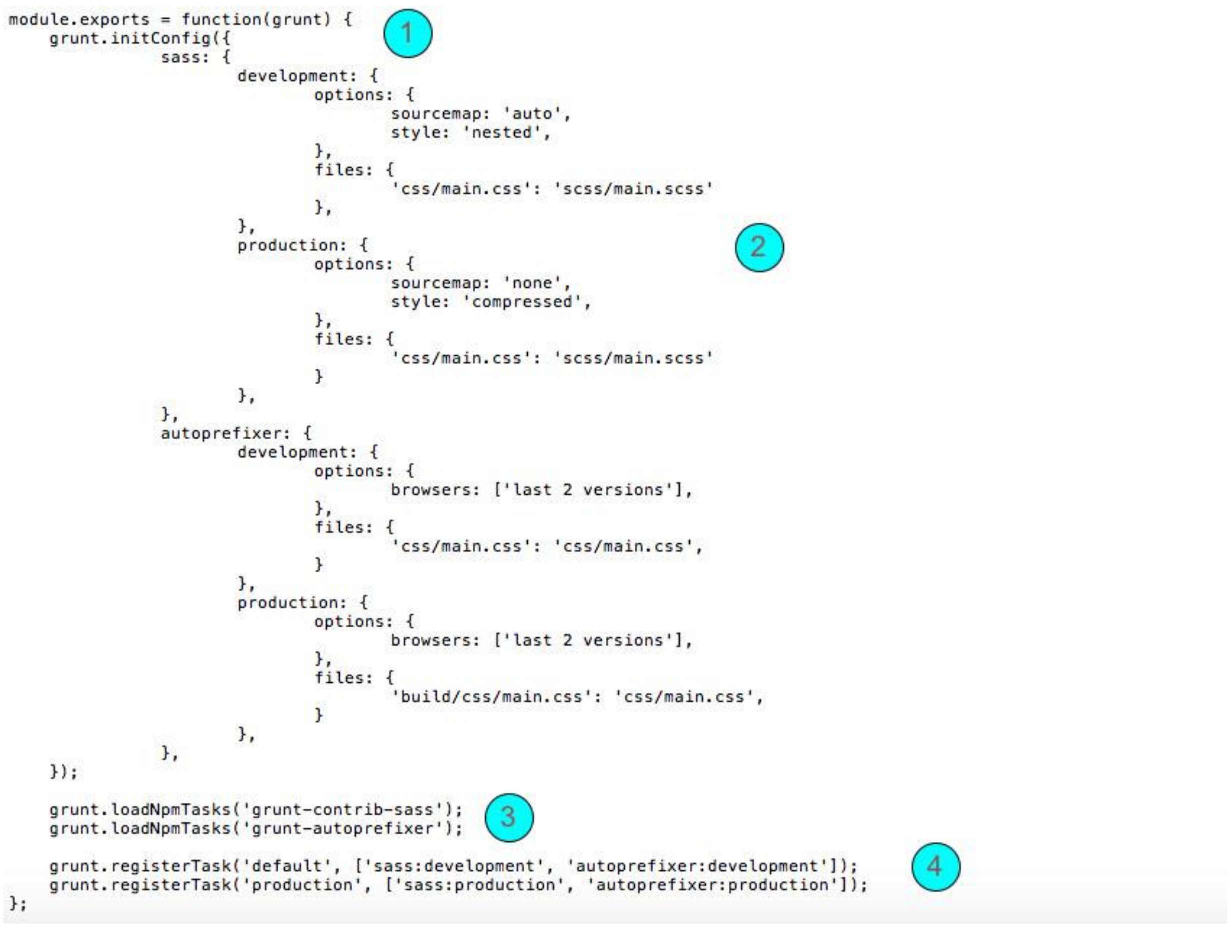
As Grunt and Gulp are both Node.js-based tools and depend on npm's *package.json* file, Node.js and npm are required when using either of these tools. The command line tools for both task runners must be installed globally with *npm install --global grunt-cli* or *gulp-cli*. In addition to installing the global CLI tools, Grunt and Gulp must also be installed as local development dependencies for each project with *npm install --save-dev grunt* or *gulp*. The *--save-dev* flag installs packages as development-only dependencies to differentiate them from front-end dependencies used in the actual application. As Grunt and

Gulp plugins are regular npm packages, they can be installed similarly via npm by looking up the correct package name either from the tools' webpage or by using *npm search* command.

Grunt was first created in 2012, but the first officially stable version 1.0.0 was only released as late as April 2016 [35]. Gulp in turn has been more liberal with their version history; version 1.0.0 was publically released in September 2013 only a few months after the initial release of the project [36]. To date there are 5 889 plugins in the npm repository that are marked as Grunt plugins and 2 637 Gulp plugins [37, 38].

The configuration file used by Grunt is called *Gruntfile.js*, and it has to be created in order to use Grunt. The configuration file consists of four parts: a wrapper function, configurations for individual tasks, plugin loading functions and task definitions. [39] An example configuration file can be seen in figure 10.

The wrapper function (marked with 1 in the example figure) is a simple function that is used to encapsulate all of the Grunt configurations inside a module ready to be used by other Node.js-based modules. Task configurations (2) are placed inside the *grunt.initConfig* function as JSON objects or arrays. In the example file *sass* and *autoprefixer* are names of tasks, more specifically they are the names that are used to point to specific plugins called *grunt-contrib-sass* and *grunt-autoprefixer*. Inside both objects there are two subtasks that are created to allow using different options for development and production versions. Subtasks are optional and can be created freely when needed. The subtasks for the *sass* task consist of two objects, *options* includes all configurable settings that can be set for this task, and *files* which states the output and input files. Available options for each plugin may vary, plugin-specific options can be viewed in the plugin's repository or npm page. Grunt plugins are often just wrappers written for an existing command line tool to make it work in Grunt. For example, *grunt-contrib-sass* command is a wrapper for the *sass* command line tool. All plugins must be loaded (3) inside the *Gruntfile.js* in order to be able to use them. Grunt has a method called *loadNpmTasks* that loads the tasks of the specified plugin. Grunt will not understand the plugins' task names unless this method is called for each plugin. Custom tasks can be created and registered in the bottom of the *Gruntfile.js* (4). This is done by calling the method *registerTask* and passing it two parameters: a name that is assigned to the task and an array of tasks that will be executed when this particular custom task is run. The tasks will be executed in the specified order. In both of the tasks registered in the example file *sass* task must be run before *autoprefixer*, because *autoprefixer* cannot handle *scss* files. Grunt tasks are called from the command line by their names, for example the task called *production* is run with *grunt production*. The *default* task can be run simply with *grunt*.

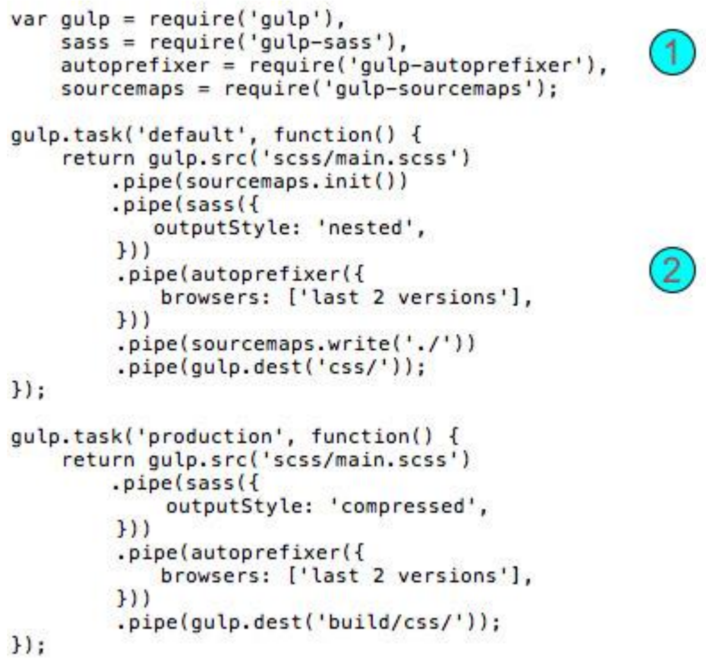


```
module.exports = function(grunt) {
    grunt.initConfig({
            sass: {
                    development: {
                            options: {
                                    sourcemap: 'auto',
                                    style: 'nested',
                            },
                            files: {
                                    'css/main.css': 'scss/main.scss'
                            },
                    },
                    production: {
                            options: {
                                    sourcemap: 'none',
                                    style: 'compressed',
                            },
                            files: {
                                    'css/main.css': 'scss/main.scss'
                            }
                    },
            },
            autoprefixer: {
                    development: {
                            options: {
                                    browsers: ['last 2 versions'],
                            },
                            files: {
                                    'css/main.css': 'css/main.css',
                            }
                    },
                    production: {
                            options: {
                                    browsers: ['last 2 versions'],
                            },
                            files: {
                                    'build/css/main.css': 'css/main.css',
                            }
                    },
            },
    });

    grunt.loadNpmTasks('grunt-contrib-sass');
    grunt.loadNpmTasks('grunt-autoprefixer');

    grunt.registerTask('default', ['sass:development', 'autoprefixer:development']);
    grunt.registerTask('production', ['sass:production', 'autoprefixer:production']);
};
```

Figure 10.   An example *Gruntfile.js.*

The equivalent of *Gruntfile.js* in Gulp is *gulpfile.js*. The Gulp configuration file only consists of two parts: plugin loading and task configurations. An example *gulpfile.js* having identical tasks as the *Gulpfile.js* in figure 10 is demonstrated in figure 11. In the first part (marked with 1 in the figure) all needed plugins are imported with Node.js method *require* by passing the plugin name as a parameter. As all plugins are initially just variables saved inside the file, the naming convention in Gulp is less strict than in Grunt; the plugins can therefore be called anything within the regular JavaScript variable naming rules. In the task configuration part (2) each task is created with *task* method. All tasks must be given a name with which it can be called from the command line. The tasks in the example file also have callback functions which are executed whenever the task is being called.

As mentioned earlier, the utilization of streams is one of the biggest differences between the concepts of Grunt and Gulp. A Node.js stream is a way to continuously read or write data in-memory. Streams are built-in into Node.js as a module called *stream*. [40] Inside the *task* method in the Gulp configuration file there are three main methods used: *src*, *pipe* and *dest*. *Src* and *dest* are methods of *vinyl-fs*, a module created by the Gulp developers. Gulp handles all files as vinyl file objects, as sort of virtual files [41] or objects of metadata representing files [42]. The *src* method accepts strictly-specified file names such as *main.scss*, directories such as *scss/* or glob patterns such as **/**.scss* as a parameter, and returns a stream of vinyl files [43]. This stream can then be passed on to the plugins to work on. Passing the streams is achieved with *pipe*, a method of the *stream* module. For example, in the task *production* the stream containing the *scss/main.scss* file in vinyl format is piped three times, first to the *sass* plugin for compiling into CSS and then on to *autoprefixer* to add the automatic vendor-prefixes. Plugin options can be configured inside the plugin function call. The last pipe in the task includes the *dest* method, which accepts a directory as parameter and writes files from the stream. Each task can have several calls to the *dest* method, if some files need to be written after a certain pipe is done.

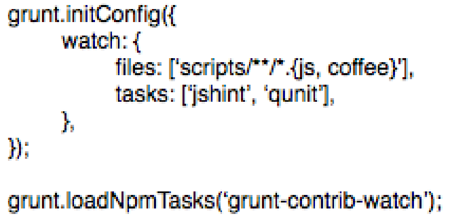


```
var gulp = require('gulp'),
    sass = require('gulp-sass'),
    autoprefixer = require('gulp-autoprefixer'),
    sourcemaps = require('gulp-sourcemaps');

gulp.task('default', function() {
    return gulp.src('scss/main.scss')
        .pipe(sourcemaps.init())
        .pipe(sass({
            outputStyle: 'nested',
        }))
        .pipe(autoprefixer({
            browsers: ['last 2 versions'],
        }))
        .pipe(sourcemaps.write('./'))
        .pipe(gulp.dest('css/'));
});

gulp.task('production', function() {
    return gulp.src('scss/main.scss')
        .pipe(sass({
            outputStyle: 'compressed',
        }))
        .pipe(autoprefixer({
            browsers: ['last 2 versions'],
        }))
        .pipe(gulp.dest('build/css/'));
});
```

Figure 11. An example *gulpfile.js.*

A good convention is to create one task per purpose, for example compiling Sass files into a CSS file. Keeping tasks simple helps keeping the *gulpfile.js* organized and reusable. Custom tasks can be chained into bigger build processes. For example, the task of re-creating the production version of the application could consist of compiling and minifying all the assets, running unit tests and copying the needed files into a build directory. This is achieved by specifying task names inside square brackets like *gulp.task('build', ['styles', 'scripts', 'images', 'test'])*. This syntax means that before running the task called *build*, the tasks inside the square brackets are executed. In this example the *build* task does not contain a callback function, so the task is finished after all the tasks have finished. In case there is a callback function, it is fired after all the tasks specified inside the square brackets have finished. This is particularly useful if running a task requires another task to be run before it. Gulp uses a module called *orchestrator* to execute all tasks with maximum concurrency, so it always tries to run all tasks possible in parallel to maximize the performance [44]. Grunt instead does not run tasks concurrently by default but can also be configured to work like Gulp.

One of the main reasons to use a task runner over multiple individual command line tools is the ability to watch for file changes and run tasks whenever changes occur. Most of the command line tools do have this feature, but using multiple tools with watch commands require one command line session per tool. With task runners all the configurations can be done in one place and file changes can be watched with a single command. Gulp has a *watch* method built-in whereas Grunt requires using a plugin called *grunt-contrib-watch*. The basics of file watching in both tools are the same; a glob pattern of file paths and a list of tasks to be executed on file changes are passed on to the method or plugin responsible for file watching. In Gulp this is very similar to using the *task* method, for example *grunt.watch('scripts/**/*.{js, coffee}', ['js', 'test'])*. This example is watching for JavaScript or CoffeeScript file changes inside any subdirectory of *scripts*, and will run tasks *js* and *test* whenever a watched file is changed. The same example using Grunt can be seen in figure 12.



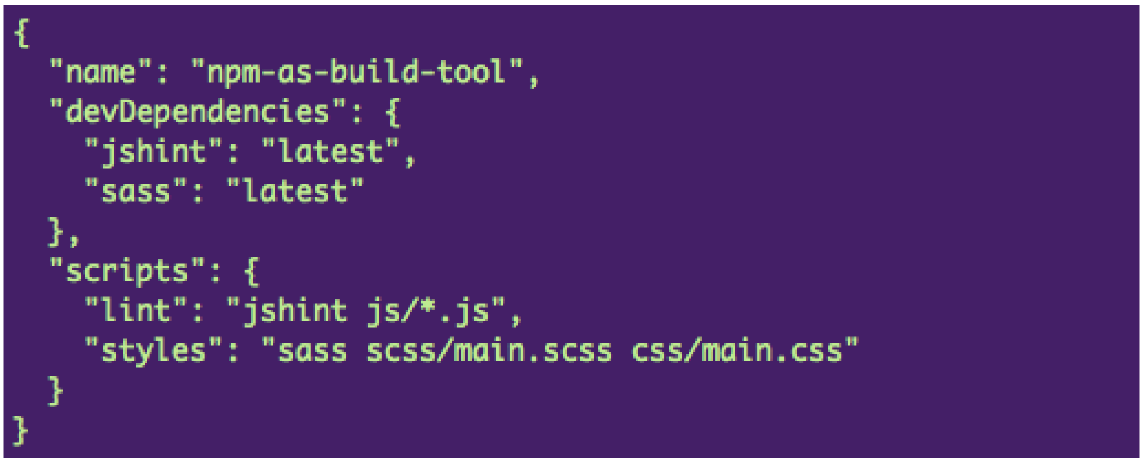
```
grunt.initConfig({
        watch: {
                files: ['scripts/**/*.{js, coffee}'],
                tasks: ['jshint', 'qunit'],
        },
});

grunt.loadNpmTasks('grunt-contrib-watch');
```

Figure 12. Grunt watch task configuration.

There are major differences between Grunt and Gulp, the most visible being the syntax they use. The syntax however is a matter of taste, some developers prefer configuring all tasks separately as JSON objects with Grunt and some like the Gulp way of piping tasks together and thus creating easy-to-understand build streams. Gulp has a speed advantage over Grunt, but a normal user will probably start noticing the differences in task executing times only when working with a very large project or constantly running some time-consuming tasks. For most users, it does not matter whether the execution of a task takes 400 ms instead of 40 ms.

Running build tasks does not necessarily require an external tool at all. As with dependency management tools, npm can also be used as a substitute task runner. This would again remove the issue of installing and introducing an external task running tool like Grunt or Gulp, as npm is already used in most projects anyway. According to Keith Cirkel, using other tools than npm in task running are adding unnecessary bloat to projects. For example, instead of using the *jshint* command line tool itself, Grunt and Gulp rely on separate wrapper plugins in order to use its features. In addition to this, to achieve identical functionalities as the original tools might also require other plugins that add features to Grunt or Gulp themselves, such as Grunt's *grunt-contrib-watch* for file watching or *gulp-util*, a Gulp plugin that adds more functionalities to Gulp tasks. Whenever a command line tool like *jshint* is updated, Grunt and Gulp users might have to wait for the plugin developers to react to the update and release an updated version for the plugin as well. Cirkel says that the Grunt and Gulp way is relying too much on wrapper plugins written for existing tools and thus complicates maintaining projects. [45, 46]

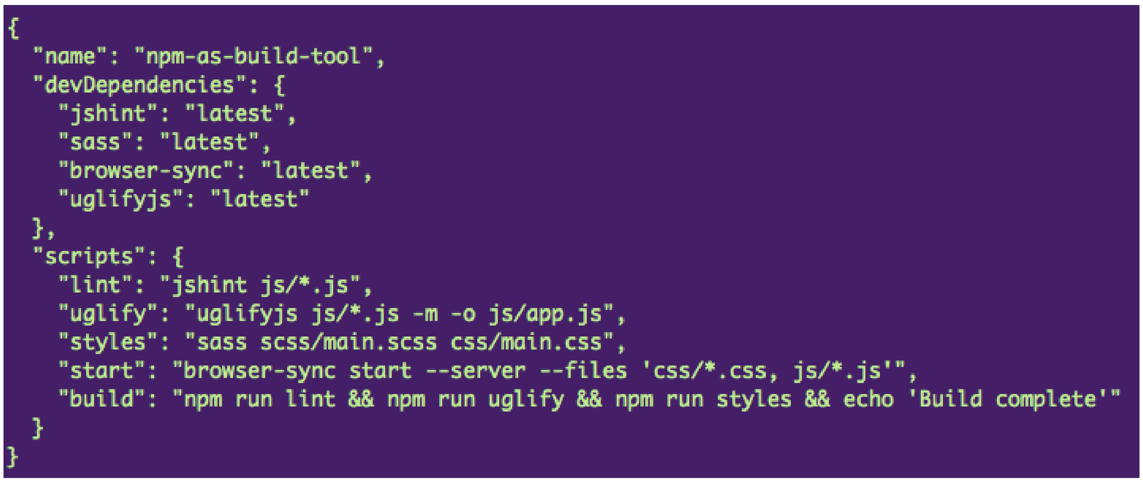
Npm can be configured to run external tools quite easily. In *package.json* configuration file, there is a property called *scripts*. Tasks can be defined within this property as JSON key-value pairs. Running external tools with npm requires the tool to be registered as a development dependency inside the *package.json* file, the same as when installing Gulp or Grunt or any of their plugins. The difference in running the tasks from npm compared to Gulp or Grunt is that npm does not require a wrapper plugin. It can run tools such as *jshint* or *sass* directly. An example setup for running an npm task can be seen in figure 13. In the example, both *jshint* and *sass* tools are installed as development dependencies, and two tasks are registered inside the *scripts* object. Tasks can be executed by running *npm run <taskName>*, in the example file *npm run lint* or *npm run styles*. Running a task will simply execute the command inside the value of the task in the default shell of the operating system [45]. The example in figure 13 is very simple, and the only advantage in running the example tasks from npm compared to running them directly from shell is that the syntax is simpler, as developers do not need to remember the correct syntaxes of each tool separately, only remembering the task name is enough. This is especially handy when running more complex tools that have multiple specific flags and options that need to be set.



```
{
  "name": "npm-as-build-tool",
  "devDependencies": {
    "jshint": "latest",
    "sass": "latest"
  },
  "scripts": {
    "lint": "jshint js/*.js",
    "styles": "sass scss/main.scss css/main.css"
  }
}
```

Figure 13. Simple task definitions in npm.

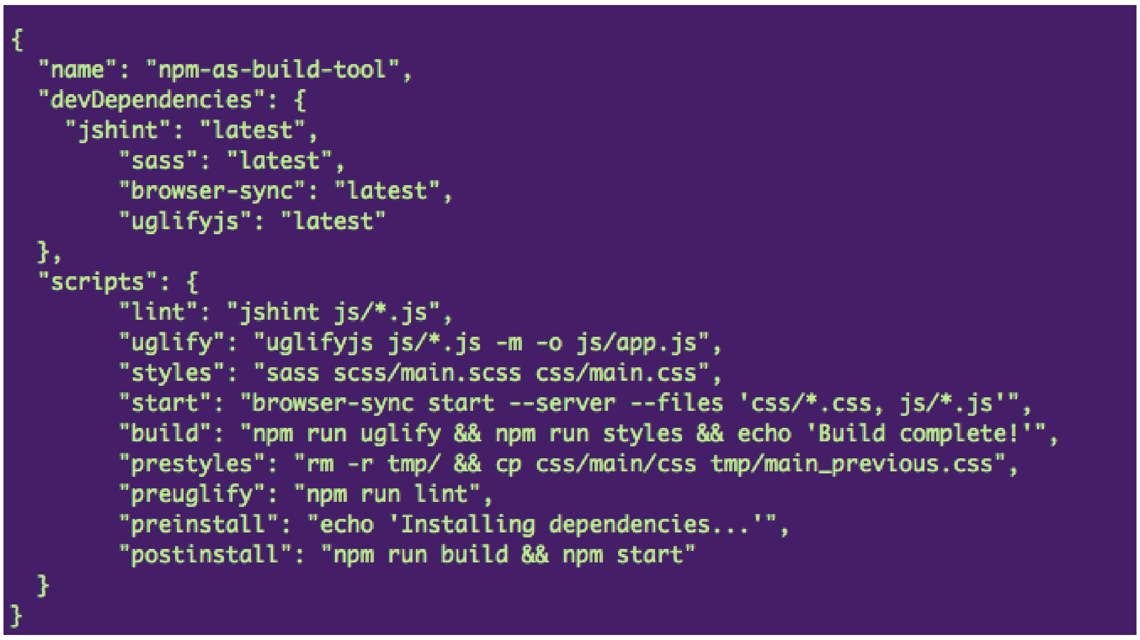
The true advantages of running npm as a build tool come from the utilization of regular shell commands. This means that tasks can be chained or their output piped together and normal shell commands can also be used inside the task definitions. Whereas Gulp or Grunt require a plugin for removing files or directories, npm can utilize the *rm* shell command directly. [45] An extended version of the example configuration utilizing more npm capabilities can be seen in figure 14. Running *npm run build* with the configuration in the example file, three npm tasks would be run with shell command *&&* chaining them together and moving onto the next task in the chain whenever the previous task was completed successfully. In the end there is also a shell command *echo* that will simply print the following string to the command line after all tasks are completed. The *start* task is a built-in task in npm, and it can be executed with simply running *npm start*. The *start* task is usually used for starting the application, for example spinning up a Node.js server. In the example file an external command line tool *browser-sync* is used for starting the application. Npm also includes other built-in tasks such as *test*, which is used for running test suites for the application.



```
{
  "name": "npm-as-build-tool",
  "devDependencies": {
    "jshint": "latest",
    "sass": "latest",
    "browser-sync": "latest",
    "uglifyjs": "latest"
  },
  "scripts": {
    "lint": "jshint js/*.js",
    "uglify": "uglifyjs js/*.js -m -o js/app.js",
    "styles": "sass scss/main.scss css/main.css",
    "start": "browser-sync start --server --files 'css/*.css, js/*.js'",
    "build": "npm run lint && npm run uglify && npm run styles && echo 'Build complete'"
  }
}
```

Figure 14. Extended npm task configuration utilizing shell features.

Developers can also hook into built-in npm tasks with *pre*- or *post*-prefixes. For example, creating a *postinstall* task will be automatically run after *npm install* is called, usually meaning someone cloning the project and installing all dependencies. Developers can therefore for example start the application automatically when the installation is complete. Furthermore, npm also supports hooking into custom tasks. Each time a task, even a task created by the developer such as *styles* in figure 14, tasks with prefixes *pre*- and *post*- are run automatically. [45, 47] This is a good way to separate tasks into smaller pieces. Figure 15 introduces these hooks into the npm script configuration. In this example the *build* task introduced in figure 14 has been simplified by moving *lint* task to *preuglify* hook. Therefore, it is executed before *uglify* task, as whenever minifying the JavaScript files into production-ready minified files with *uglifyjs*, the developer might want to test that there are no errors in the files with *jshint*. In the *prestyles* task the *tmp* directory is first removed with the shell command *rm* and after that, the existing CSS file is copied into *tmp* directory. The *preinstall* and *postinstall* tasks are used for easier project initialization.



```
{
  "name": "npm-as-build-tool",
  "devDependencies": {
    "jshint": "latest",
        "sass": "latest",
        "browser-sync": "latest",
        "uglifyjs": "latest"
  },
  "scripts": {
        "lint": "jshint js/*.js",
        "uglify": "uglifyjs js/*.js -m -o js/app.js",
        "styles": "sass scss/main.scss css/main.css",
        "start": "browser-sync start --server --files 'css/*.css, js/*.js'",
        "build": "npm run uglify && npm run styles && echo 'Build complete!'",
        "prestyles": "rm -r tmp/ && cp css/main/css tmp/main_previous.css",
        "preuglify": "npm run lint",
        "preinstall": "echo 'Installing dependencies...'",
        "postinstall": "npm run build && npm start"
  }
}
```

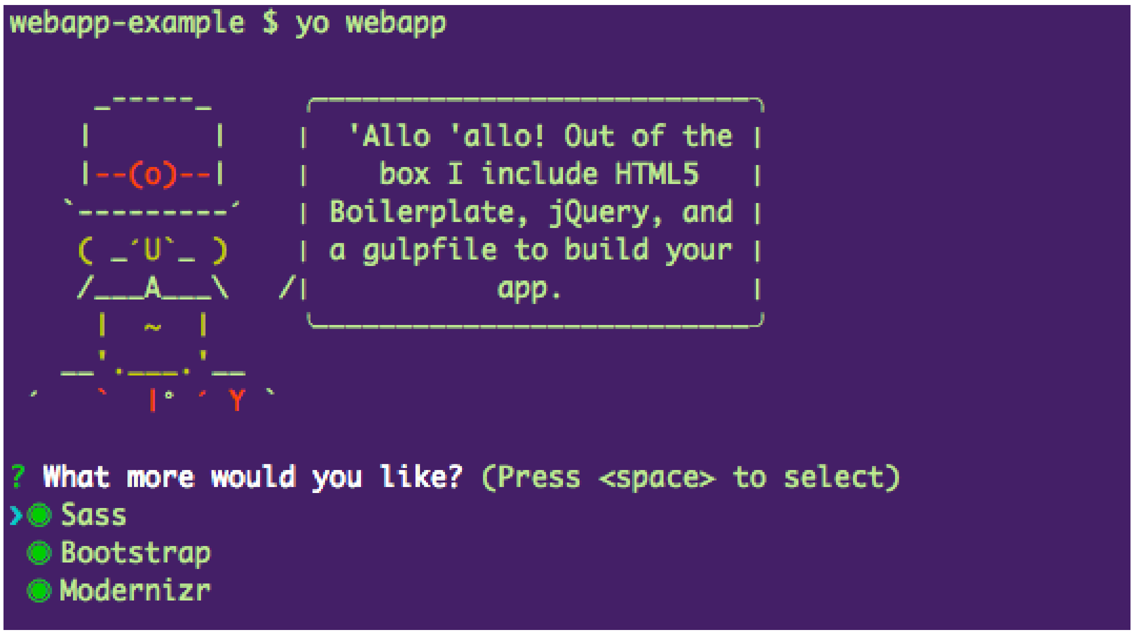
Figure 15. Extended npm configuration utilizing hooks.

4.3 Scaffolding

Client-side scaffolding is a term used for the process of setting up or creating the initial template for a project. The scaffolding itself is a one-time-only process run as the very first task in a new project, but scaffolding a project also installs several tools that help in building the project, and suggests best practices and solutions for common problems.

The de-facto tool in client-side scaffolding is Yeoman. Yeoman is based on generators that can be thought of as plugins. Yeoman itself is non-opinionated, which means it does not make decisions on behalf of the users, it is merely a toolkit that provides the ecosystem for the generators [48, 49]. Generators are built and maintained by either the developers of Yeoman or the wide open-source community around it. Currently there are 4 321 published generators [50]. Generators are basically plugins that provide developers with the best practices that have been established throughout the community. They create the initial skeleton of the application, meaning the directory structure, and a set of tools that help in development and maintenance of the project. [48] As there are specific generators for each framework, for example Angular, they are often opinionated, but are also usually modular to provide flexibility for the developers.

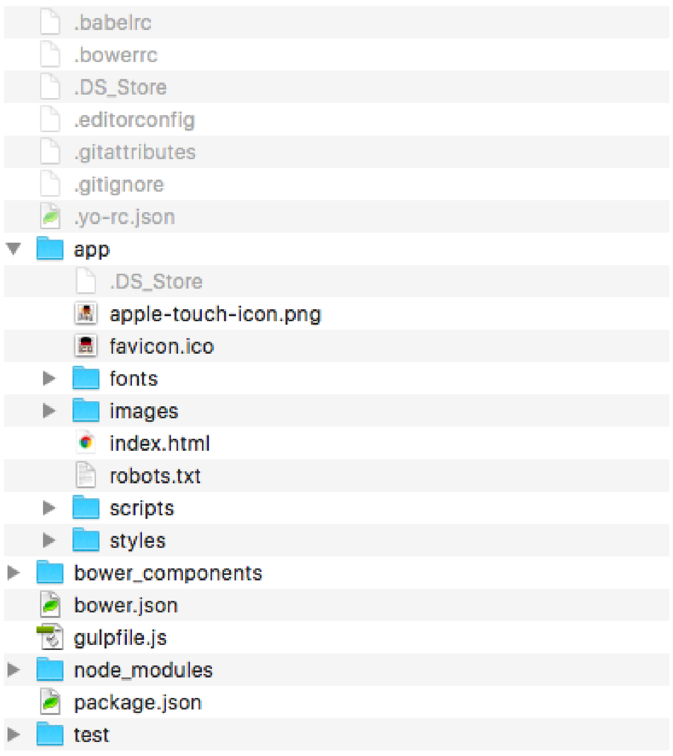
Yeoman can be installed as a global npm package by running *npm install yo --global*. Yeoman generators are also npm packages and they must also be installed globally. The names of the generators always consist of the *generator*-prefix followed by the name of the generator, for example *generator-webapp.* After installing the tools, a generator can be run with a simple command *yo <generator-name>*, for example *yo webapp*. Generators usually allow the developer to interactively choose from a set of built-in tools and libraries when running a generator, as demonstrated in figure 16.



```
webapp-example $ yo webapp

     _-----_     .--------------------------.
    |       |    |  'Allo 'allo! Out of the |
    |--(o)--|    |    box I include HTML5   |
   `---------´   | Boilerplate, jQuery, and |
    ( _´U`_ )    | a gulpfile to build your |
    /___A___\   /|           app.           |
     |  ~  |     '--------------------------'
   __'.___.'__
 ´   `  |° ´ Y `

? What more would you like? (Press <space> to select)
❯◉ Sass
 ◉ Bootstrap
 ◉ Modernizr
```

Figure 16. Example of interactive options when running Yeoman generators.

After going through all the options, the generator creates the directory structure and some boilerplate files to start with. This forms the skeleton of the application. The generator also automatically runs *npm install* to install all of its dependencies, and in the case of *generator-webapp*, it also automatically runs *bower install* to install all front-end dependencies. Because *generator-webapp* uses Bower without asking or providing alternatives, it is therefore opinionated. The creators of the generator have also decided to use Gulp as the task-running tool instead of Grunt. The final directory structure can be seen in figure 17. As can be seen in the figure, the generator has created *package.json* and *bower.json* configuration files, and installed all npm and Bower dependencies into *node_modules* and *bower_components* directories. All of the application-specific front-end files are separated into a directory called *app*.



```
.babelrc
.bowerrc
.DS_Store
.editorconfig
.gitattributes
.gitignore
.yo-rc.json
▼ app
    .DS_Store
    apple-touch-icon.png
    favicon.ico
  ▶ fonts
  ▶ images
    index.html
    robots.txt
  ▶ scripts
  ▶ styles
▶ bower_components
bower.json
gulpfile.js
▶ node_modules
package.json
▶ test
```

Figure 17. Final directory structure for *generator-webapp*.

The generator enforces the usage of Gulp as the task-runner, and the created *gulpfile.js* includes multiple pre-defined tasks ready to be used. For example, running *gulp serve* from the project's root directory creates a Node-based web-server using Browser-sync. Running the command opens up the application's *index.html* in a web browser with some boilerplate content. The command will also tell Gulp to start watching for changes in the project files and run the needed tasks and refresh the browser automatically. The application is ready to be developed without the developer ever having to touch any of the files.

Although the advantages of using Yeoman generators are most visible during the setup of a project, many generators also provide help during the development phase in the form of sub-generators. While the actual generator takes care of the initial scaffolding, sub-generators are particularly useful when dealing with more complex frameworks like Angular or React. Sub-generators provide means to create smaller parts of the application fast, such as controllers in Angular. [48] Sub-generators are also prescribing best practices so that the developer does not have to guess in which directory the new component should be placed. For example, *generator-angular*, a generator for Angular framework built by the Yeoman team, provides 11 different sub-generators, all creating and configuring small components, that can be run whenever needed during development [51].

5 Optimization setup for a custom web application

The web application in this project implements a headless, sometimes called decoupled, content management system (CMS) architecture, in which a CMS handles storing the data into a database and serving it through an application programming interface (API). In this architecture viewing the data is not tied to the CMS itself as the front-end side of the application is decoupled from the back-end side. [52] Content management systems are utilizing themes to view the data, and all of them have their own structures for presenting data, and they often have limitations. The headless architecture is utilizing the advantages of both the administration interface of the CMS, in which it is easy to create, edit and maintain content, and the flexibility of a completely independent front-end system, in which the developers have more freedom to use whichever tools and front-end frameworks to view the data. In many cases the front-end side of the application is built using a JavaScript framework such as Angular or Backbone, but because the content is being served through an accessible API, it is also possible to utilize the data for example in a mobile application. [53]

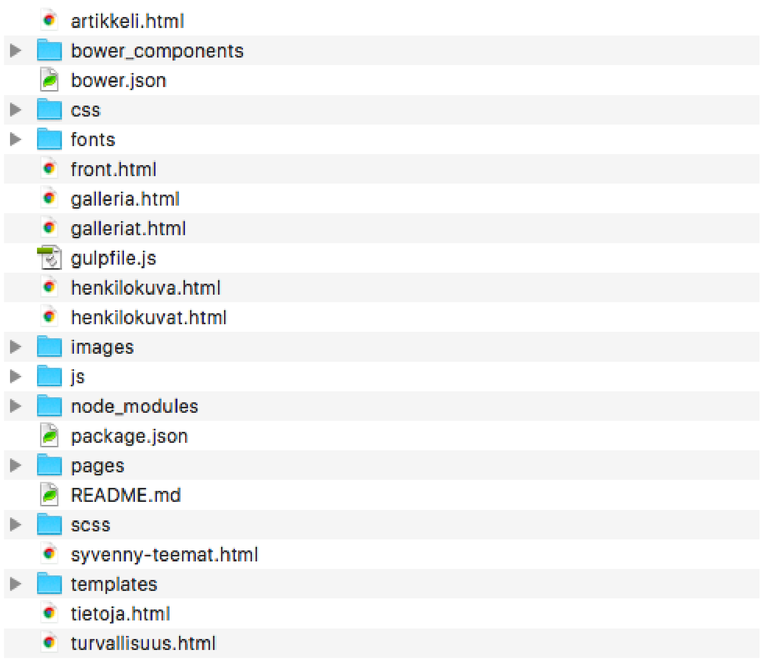
In the beginning of the project Drupal 8 was selected as the back-end system storing all content, but no front-end technology decisions were made at that point. In the project it was important to come up with a minimum viable product fairly quickly to be able to demonstrate the overall layout in a live environment. It was then decided that an HTML mockup of the application should be created first according to the designed layouts. The initial views of the application had many identical content blocks, and it was noticed that the project would benefit from an HTML templating engine that would allow modifying those blocks jointly to reduce copying the same adjustments to each view individually. As it was clear from the beginning that the templating engine system would later be replaced by the actual front-end framework, it was crucial that the optimization tools that would be selected for the project should be above all flexible. Flexibility in the setup allows developers to make major changes rapidly.

Most Yeoman generators are framework-specific, and as there was no framework decided yet, it was decided that no client-side scaffolding generator would be used in the project. Using a ready-made project template created by a generator did not feel as flexible or easily-customizable as the project needed. Yeoman generators create the whole project skeleton, and switching from a generator to another during development is not reasonable, because in most cases the project would need to be re-created from scratch as the directory structures for different frameworks vary.

Instead of using a scaffolding tool, a decision was made that a task runner and a dependency management tool were needed in the project. Node.js and npm were installed for the project as they are requirements for most modern tools. We wanted to keep task running and dependency management separated from npm to achieve a simpler setup. Although individual tools for each process can be thought of as more difficult to maintain, it felt easier to have an individual tool for each process rather than assigning all processes to a single tool.

I had previously worked with Grunt as a task runner, but after comparing Grunt with Gulp, the "code-over-configuration" approach that Gulp is utilizing convinced us to use it for running tasks in the project. Other facts that advocated Gulp over Grunt were that the Gulp plugin for compiling HTML from Nunjucks, an HTML templating language that was used in the project, had more downloads, was more actively developed and was in a more mature state according to the version of the module. Bower was installed as the project's front-end dependency management tool.

The initial project structure can be seen in figure 18. In the beginning there were four major tasks that Gulp was running. *Serve* task utilized Browser-sync to initiate a Node.js-based server that not only keeps the application synchronized on all browsers on the developer's computer, but also grants other devices in the same network access to view the application through a specified IP address. This task was also responsible for watching for file changes. Changes in *scss* directory fired a task that compiled all SCSS files into a combined CSS file, piping the stream through *gulp-autoprefixer* which automatically adds browser-specific CSS prefixes, and *gulp-sourcemaps* which creates a CSS source map to help in debugging the application. The directory *pages* includes layouts for all views created with Nunjucks, while *templates* has smaller parts of the HTML included in the actual views, such as header. File changes in either of these directories fired *nunjucks* task, that compiled the Nunjucks files into HTML. The compiled HTML files were placed in the root of the project, as can be seen in figure 18. The last task was *bower* that utilized *main-bower-files*. This plugin traverses through the *dependencies* property of the *bower.json* file, finds a matching dependency in *bower_components,* and returns an array of all the files marked as the main entry points in each of the components' own *bower.json* files. The output of this stream was filtered by file extensions, JavaScript files were concatenated and minified into a single *vendor.js* file, and CSS files into a *vendor.css* file.



```
  artikkeli.html
▶ bower_components
  bower.json
▶ css
▶ fonts
  front.html
  galleria.html
  galleriat.html
  gulpfile.js
  henkilokuva.html
  henkilokuvat.html
▶ images
▶ js
▶ node_modules
  package.json
▶ pages
  README.md
▶ scss
  syvenny-teemat.html
▶ templates
  tietoja.html
  turvallisuus.html
```

Figure 18. Directory structure of the application in the start of the project.

Later on in the project when the actual server for the application was setup, two new tasks were created, one handling the building process of a production-ready version of the application and the other transferring the production version to the server. At the same time several environment flags were introduced, for example running *sass* task with flag *--production* would minify the compiled CSS file and not create a source map for the CSS file. The flag would also tell Gulp to save the file into the production version of the application rather than the directory used for development. By using flags there was no need to make a copy of each task for production purposes, which would have considerably lengthened the *gulpfile.js,* since the basic functionalities of each task are similar with only minor differences. At this point Angular was selected as the front-end framework for the project, and as Angular has its own way of including HTML templates, Nunjucks was dropped and uninstalled from the project. The Gulp task that handled compiling Nunjucks into HTML was replaced with a task that concatenates all application JavaScript files into one single *app.js* file and validates it before telling Browser-sync to reload the page.

The *build* task utilizes a plugin called *run-sequence* which allows running defined tasks in a specified order. In this case the task first removes the existing *production* directory with *gulp-clean* plugin, then copies files not compiled through any task, such as images and fonts, into the directory, and in the end runs the compiling tasks in parallel. When the *production* directory is ready, Gulp automatically creates a new Browser-sync instance that serves the production version of the application ready to be tested before deploying the new version to the server. The *build* task has to therefore always be run with the flag *--production*, so that other tasks know which version of the files is needed, with which options and where to save the files.

The *deploy* task uses a plugin called *gulp-rsync*, which is a wrapper that enables using the tool *rsync* in Gulp. *Rsync* does not transfer the complete production version to the server, it synchronizes the destination with the source, so that only changes made after last synchronizing are transferred to server. This reduces the time it takes to deploy the changes significantly, especially with large applications. The *deploy* task can be modified with several flags, *--staging* or *--production* to distinguish which server *rsync* should transfer the files, and an optional *--dry* that only prints out the files that would be changed without actually making the changes, which is useful and more secure if there are lots of files that need synchronizing.

6 Results

The setup created for the application optimizes or automates five of the six parts of the development workflow introduced by Zell Liew. Scaffolding was the only part this setup did not touch, it was done manually due to possible changes in the structure of the application, and also to learn more about the tools and techniques by doing, rather than using a ready-made solution. Several Gulp tasks were created to automate parts of the development phase, such as validating and concatenating files. The optimizations in the development part reduce repetitiveness and let developers work on building the code instead of running several compilers time and again.

The Node.js server created with Browsersync helped a great deal in testing the application. It allows easy integration with multiple browsers and mobile devices simultaneously, as well as quickly reviewing changes with other people working on the same project without the need to first commit, push and pull changes to or from the git repository.

Integrating with other developers' changes is made easier with the usage of Bower and npm especially. As all packages and libraries are listed in configuration files that can be updated, there is no discrepancy between developers on which package versions to use. By creating a *.bowerrc* file and adding a *postinstall* script property, the integration can be made even easier, as that script is automatically run after each *bower install* command. In this application *gulp bower* is run after *bower install*, so that all new front-end dependencies are instantly added into the application. Similar scripts could also be added to the project as git hooks, for example automatically running *npm install* as a *post-merge* hook whenever *package.json* file has been changed by another developer, but they need to be added on each computer individually, as git files are not usually included in version control systems.

Optimizing assets for production version was also achieved with Gulp tasks. Separate task configurations for development and production environments keeps the production version of the application as small and fast as possible by minifying all assets while still keeping the development version easy to read and debug. Gulp also completely handles deploying code to the production servers with a single command. The deployment process is also secure, as usernames and other server configurations are kept in a separate file that is not in the version control system, and SSH keys are required for a successful deployment.

Re-installation process of the application is almost identical for different operating systems. It takes eight or nine steps to view the application in a browser, depending on the operating system in use. The first part is installing Node.js and npm, which differs for all operating systems. For Mac OS X and Linux-based operating systems it is suggested that the path of npm is changed to a location that does not require superuser rights, whereas this step is not needed for Windows computers. Other steps that differ are installing Git and cloning the application from the Git repository. All other tasks are identical as they are based on Node.js which is a cross-platform environment. Gulp and Bower must be installed as global dependencies to be able to compile and run the application. After cloning the application, only three terminal commands are needed to run the application in a browser. *Npm install* installs all local development dependencies and *bower install* installs all front-end dependencies. *Gulp bower* is run automatically after *bower install* has completed because of the *postinstall* script added into *.bowerrc* configuration file. The last step is to run *gulp* which defaults to the *gulp serve* task, with which the application can be viewed at http://localhost:3000 address. The process is even more straight-forward if all global tools, such as Node.js and Git, are already installed on the computer.

Future

As new tools are emerging each week, it takes some effort to try to keep up with the constantly changing field of different dependency management and other tools. Once you learn how to use one tool and are getting familiar on the best practices in using it, there is a chance that someone has already created a new tool that solves some of the problems the older tools have, such as being faster or more reliable. These modern development tools are constantly going towards a more open-source future, as the tools are developed, managed and enhanced by a large amount of people, but at the same time the field is getting more and more fragmented, as completely new tools are created instead of working together and trying to fix the possible problems a tool might have and overcoming them.

In October 2016, four large companies, including Facebook and Google, announced a new tool called Yarn, which is, in their own words, providing a faster, more reliable and more secure way of managing dependencies. [54] It is basically an improved version of the npm client, however, it is not intended to replace npm as a whole, as it uses the npm registry for downloading the dependencies, combining it with Bower registry. The basics of Yarn are very similar to npm, but there are differences, especially in the installation process of new dependencies, and slight variations in the commands it offers. Yarn is capable of running tasks parallel when installing dependencies, which makes Yarn faster compared to the sequential installing process of npm [55]. In a comparison by Nikhil John, Yarn proved to be on average 4.7 times quicker than npm when installing dependencies [56]. Unlike npm, Yarn uses a flat dependency tree and saves all packages in a global cache, so that when a package is once installed, next time it can be installed from cache, which enables installing packages also offline. Differences in commands include for example dividing *npm install* into two different commands, where *yarn install* reads and installs the dependencies from *package.json*, and *yarn add* installs new dependencies from the registry. Each added dependency is also automatically saved into *package.json*, removing the need for an extra flag like npm's *--save*. As both tools have almost identical commands and Yarn is using npm registry, there could have been a way to make these improvements directly to npm instead of creating a new tool, as Tim Severien states [55]. Yarn is therefore a good example of the fragmentation of development tools.

Another popular approach to optimizing web development is module bundling. Webpack is currently the most popular JavaScript module bundler. Webpack is a powerful tool capable of splitting all kinds of assets into bundles, that can be loaded into browser whenever needed. This allows for smaller file sizes, and with the Hot Module Replacement feature, bundles can be loaded without refreshing the webpage. Webpack is able to bundle not just JavaScript, but also other kinds of assets such as CSS, images and fonts with the help of its dependency graph concept. Webpack uses loaders to pre-process assets, for example to compile Sass into CSS. With Webpack you could for example create a bundle for the home page of the application, that only includes assets needed in it thus making the file size smaller. Webpack is optimal when the application is using a modular JavaScript design pattern, where the application JavaScript is divided into small, reusable modules. Webpack does not have to completely replace task runners like Gulp or Grunt, they can be used side by side, for example with *gulp-webpack* plugin. [57, 58]

Learning to use and configure Webpack correctly, however, can be really tricky, especially for beginners. The configuration files and their syntax are complicated. The documentation of Webpack is also not very easy to understand according to Andrew Ray. [58] A similar module bundler, but with less massive core features and gentler learning curve, is Browserify, which may be easier to learn for beginners. Browserify can also be a better suit for smaller projects with less assets to bundle and configure. With smaller projects Webpack might be an over-complicated solution.

7 Summary

The goal of this thesis was to research different ways to automate and optimize the development workflow with modern JavaScript-based tools and techniques, and to find a functional setup for a customized web application.

New tools are being created each day, all of them trying to fix problems of preceding tools. This makes keeping up with the pace hard, it is almost impossible to know which tools are worth learning and which are not. The field of front-end development tooling is also getting more and more fragmented; new tools are being introduced instead of working together to improve the existing tools despite the open-source philosophy that most of the tools have.

The most important thing is not to learn the syntax of every single tool, but rather to understand the concepts and solutions the different tools are trying to accomplish. After learning why and how to use one tool, it is often not complicated to learn new ones and switch to them in case the other tools seem more suitable. It is not as important to memorize for example all of the differences in Gulp and Grunt commands, as it is to understand the common concept of those tools, and what those tools are trying to accomplish and what problems they are solving.

Most web application projects are different. There is no need to find a single setup that works for all projects, but rather to learn about different options and to find a suitable set of tools that work for you. These tools can then be extended according to the needs of each project. For the web application in this project, the tools selected were npm, Bower and Gulp, with many other tools working with Gulp through plugins. The setup covers all of the most important parts of the workflow, excluding only the scaffolding part of the workflow. Most importantly, the setup is relatively easy to install and use, with no complicated commands to memorize. This setup can serve as the base for future projects, as it is easy to enhance it by adding more automated tasks for specific needs or removing obsolete ones. Yeoman could also be added to future projects for quicker initialization of the project.

References

1 Smith, Robert. 2015. A modern front-end workflow [online]. <http://rbrtsmith.com/2015/08/a-modern-frontend-workflow>. 29.8.2015. Accessed 31.10.2016.

2 Young, Alex R. 2015. Small Modules: Tales from a Serial Module Author [online]. <http://dailyjs.com/2015/07/02/small-modules-complexity-over-size>. 2.7.2015. Accessed 31.10.2016.

3 Liew, Zell. 2015. An Overview of a Development Workflow [online]. <http://zellwk.com/blog/workflow-overview>. 3.6.2015. Accessed 29.7.2016.

4 Bracey, Kezz. 2015. The Command Line for Web Design: Taming 3rd Party Packages [online]. <https://webdesign.tutsplus.com/tutorials/the-command-line-for-web-design-taming-3rd-party-packages--cms-23451>. 17.3.2015. Accessed 24.10.2016.

5 Bracey, Kezz. 2015. The Command Line for Web Design: Grasping The Basics [online]. <https://webdesign.tutsplus.com/tutorials/the-command-line-for-web-design-grasping-the-basics--cms-23318>. 10.3.2015. Accessed 24.10.2016.

6 About Node.js [online]. Node.js. <https://nodejs.org/en/about>. Accessed 15.6.2016.

7 Nguyen, Don. 2012. Jump start Node.js. Collingwood: Sitepoint.

8 What is npm? [online]. npm. <https://docs.npmjs.com/getting-started/what-is-npm>. Accessed 8.7.2016.

9 DeBill, Erik. Module Counts [online]. <http://www.modulecounts.com/>. Accessed 27.6.2016.

10 Package.json [online]. npm. <https://docs.npmjs.com/files/package.json>. Accessed 8.7.2016.

11 Lee, Edward A. 2006. The Problem with Threads [online]. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf>. 1.10.2006. Accessed 22.6.2016.

12 Norris, Trevor. 2015. Understanding the Node.js Event Loop [online]. <https://nodesource.com/blog/understanding-the-nodejs-event-loop>. 20.1.2015. Accessed 22.6.2016.

13 Hall, Adron. 2013. Understanding the Node.js Event Loop [online]. <https://strongloop.com/strongblog/node-js-event-loop>. 19.12.2013. Accessed 22.6.2016.

14 Node.js - Event Loop [online]. Tutorials Point. <http://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm>. Accessed 22.6.2016.

15 Kasiuk, Aleksander. 2015. On problems with threads in node.js [online]. <https://www.future-processing.pl/blog/on-problems-with-threads-in-node-js>. 22.4.2015. Accessed 22.6.2016.

16 Node.js programs versus Ruby [online]. The Computer Language Benchmarks Game. <http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=node&lang2=yarv>. Accessed 19.6.2016.

17 Aissani, Camel. Node.js [online]. <http://slidedeck.io/camelaissani/nodejs-presentation>. Accessed 22.6.2016.

18 Schlueter, Isaac. 2011. npm 1.0: Global vs Local installation [online]. <https://nodejs.org/en/blog/npm/npm-1-0-global-vs-local-installation>. 24.3.2011. Accessed 8.7.2016.

19 sudo - Unix, Linux Command [online]. Tutorials Point. <http://www.tutorialspoint.com/unix_commands/sudo.htm>. Accessed 8.7.2016.

20 Installing npm packages globally [online]. npm. <https://docs.npmjs.com/getting-started/installing-npm-packages-globally>. Accessed 8.7.2016.

21 Fixing npm permissions [online]. npm. <https://docs.npmjs.com/getting-started/fixing-npm-permissions>. Accessed 8.7.2016.

22 All Docs [online]. npm. <https://docs.npmjs.com/all>. Accessed 22.7.2016.

23 Ogden, Max. 2015. Nested Dependencies [online]. <http://maxogden.com/nested-dependencies.html>. January 2015. Accessed 22.7.2016.

24 Osmani, Addy. 2013. Automating Front-end Workflow [online]. <https://speaker-deck.com/addyosmani/automating-front-end-workflow>. 25.10.2013. Accessed 29.7.2016.

25 Bower [online]. Bower. <https://bower.io>. Accessed 29.7.2016.

26 Getting Started - Installing Git [online]. Git. <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>. Accessed 29.7.2016.

27 Configuration [online]. Bower. <https://bower.io/docs/config>. Accessed 29.7.2016.

28 Lindley, Cody. 2014. The Mystical & Magical SemVer Ranges Used by npm & Bower [online]. <http://developer.telerik.com/featured/mystical-magical-semver-ranges-used-npm-bower>. 6.10.2014. Accessed 29.7.2016.

29 Osmani, Addy. 2013. Checking in front-end dependencies [online]. <https://ad-dyosmani.com/blog/checking-in-front-end-dependencies>. 29.7.2013. Accessed 1.8.2016.

30 Salonen, Jaakko. 2015. Why We Should Stop Using Bower – And How to Do It [online]. <https://gofore.com/stop-using-bower>. 25.5.2015. Accessed 1.8.2016.

31 Rachev, Preslav. 2015. Gulp vs Grunt. Why one? Why the Other? [online]. <https://medium.com/@preslavrachev/gulp-vs-grunt-why-one-why-the-other-f5d3b398edc4>. 6.1.2015. Accessed 8.8 2016.

32 Pataki, Daniel. The Battle of Build Scripts: Gulp Vs Grunt [online]. <http://www.hongkiat.com/blog/gulp-vs-grunt>. Accessed 8.8.2016.

33 Hsu, Jack. 2014. 30 - Beyond the Numbers [online]. <http://jaysoo.ca/2014/01/27/gruntjs-vs-gulpjs>. 27.1.2014. Accessed 8.8.2016.

34 Martineau, Zander. 2014. Speedtesting gulp.js and Grunt [online]. <http://tech.tmw.co.uk/2014/01/speedtesting-gulp-and-grunt>. 15.1.2014. Accessed 8.8.2016.

35 Grunt releases [online]. GitHub. <https://github.com/gruntjs/grunt/releases>. Accessed 12.9.2016.

36 Gulp changelog [online]. GitHub. <https://github.com/gulpjs/gulp/blob/master/CHANGELOG.md> Accessed 12.9.2016.

37 Plugins [online]. Grunt. <http://gruntjs.com/plugins> Accessed 12.9.2016.

38 Plugins [online]. Gulp. <http://gulpjs.com/plugins>. Accessed 12.9.2016.

39 Getting started [online]. Grunt. <http://gruntjs.com/getting-started>. Accessed 8.8.2016.

40 Stream [online]. Node.js. <https://nodejs.org/api/stream.html> Accessed 12.9.2016.

41 Kappert, Lars. 2014. Getting gulpy [online]. <https://medium.com/@webprolific/getting-gulpy-a2010c13d3d5>. 6.5.2014. Accessed 12.9.2016.

42 Vinyl-fs [online]. GitHub. <https://github.com/gulpjs/vinyl-fs>. Accessed 12.9.2016.

43 Gulp API docs [online]. GitHub. <https://github.com/gulpjs/gulp/blob/master/docs/API.md>. Accessed 12.9.2016.

44 Gulp Dissection [online]. Delapouite. <http://delapouite.com/ramblings/gulp-dissection.html>. Accessed 12.9.2016.

45 Cirkel, Keith. 2014. How to Use npm as a Build Tool [online]. <https://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool>. 9.12.2014. Accessed 16.9.2016.

46 Cirkel, Keith. 2014. Why we should stop using Grunt & Gulp [online]. <https://www.keithcirkel.co.uk/why-we-should-stop-using-grunt>. 30.10.2014. Accessed 16.9.2016.

47 npm-scripts [online]. npm. <https://docs.npmjs.com/misc/scripts>. Accessed 23.9.2016.

48 Getting started with Yeoman [online]. Yeoman. <http://yeoman.io/learning/>. Accessed 26.9.2016.

49 Strumpflohner, Juri. 2014. Node, Grunt, Bower and Yeoman - A Modern web dev's Toolkit [online]. <http://juristr.com/blog/2014/08/node-grunt-yeoman-bower/>. 15.8.2014. Accessed 26.9.2016.

50 Generators [online]. Yeoman. <http://yeoman.io/generators>. Accessed 26.9.2016.

51 AngularJS Generator [online]. GitHub. <https://github.com/yeoman/generator-angular>. Accessed 7.10.2016.

52 Headless and decoupled CMS: the essential guide [online]. Contentful. <https://www.contentful.com/r/knowledgebase/headless-and-decoupled-cms/>. Accessed 7.10.2016.

53 Decoupled CMS: Why “Going Headless” Is Becoming So Popular [online]. Pantheon. <https://pantheon.io/decoupled-cms>. Accessed 7.10.2016.

54 Getting started [online]. Yarn. <https://yarnpkg.com/en/docs/getting-started>. Accessed 28.10.2016.

55 Severien, Tim. 2016. Yarn vs npm: Everything You Need to Know [online]. <https://www.sitepoint.com/yarn-vs-npm/>. 19.10.2016. Accessed 28.10.2016.

56 John, Nikhil. 2016. Facebook’s Yarn vs npm — Is Yarn really better? [online]. <https://medium.com/@nikjohn/facebooks-yarn-vs-npm-is-yarn-really-better-1890b3ea6515>. 12.10.2016. Accessed 28.10.2016.

57 Vepsäläinen, Juho. Webpack compared [online]. <http://survivejs.com/webpack/webpack-compared/>. Accessed 28.10.2016.

58 Ray, Andrew. 2016. Webpack: When to Use and Why [online]. <http://blog.andrewray.me/webpack-when-to-use-and-why/>. 9.4.2016. Accessed 28.10.2016.

Package.json

```
{
  "name": "sananvapaus",
  "version": "0.8.0",
  "description": "Angular webapp",
  "private": true,
  "main": "src/app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Hermanni Piirainen",
  "repository": {
   "type": "git",
   "url": "https://github.com/FloAppsLtd/Sananvapaus"
  },
  "devDependencies": {
    "browser-sync": "^2.11.1",
    "connect-history-api-fallback": "^1.3.0",
    "fs": "0.0.2",
    "gulp": "^3.9.1",
    "gulp-autoprefixer": "^3.1.0",
    "gulp-clean": "^0.3.2",
    "gulp-clean-css": "^2.0.6",
    "gulp-concat": "^2.6.0",
    "gulp-debug": "^2.1.2",
    "gulp-filter": "^4.0.0",
    "gulp-if": "^2.0.1",
    "gulp-imagemin": "^3.1.0",
    "gulp-jshint": "^2.0.1",
    "gulp-prompt": "^0.2.0",
    "gulp-rename": "^1.2.2",
    "gulp-rsync": "0.0.6",
    "gulp-sass": "^2.2.0"
    "gulp-size": "^2.1.0",
    "gulp-sourcemaps": "^1.6.0",
    "gulp-uglify": "^1.5.3",
    "gulp-util": "^3.0.7",
    "main-bower-files": "^2.11.1",
    "run-sequence": "^1.2.0",
    "yargs": "^4.7.1"
  },
  "dependencies": {}
}
```

Bower.json

```
{
  "name": "sananvapaus",
  "description": "Angular webapp",
  "main": "src/app.js",
  "authors": [
    "Hermanni Piirainen"
  ],
  "homepage": "https://github.com/FloAppsLtd/Sananvapaus",
  "moduleType": [],
  "private": true,
  "dependencies": {
    "jquery": "1.11.2",
    "angular": "^1.5.5",
    "clipboard": "^1.5.10",
    "ngclipboard": "^1.1.1",
    "angular-filter": "^0.5.8",
    "jquery-smooth-scroll": "^1.7.2",
    "fancybox": "^2.1.5",
    "angular-flash-alert": "^2.2.7",
    "jcf": "^1.2.3",
    "angular-bind-html-compile": "^1.2.1",
    "angular-route": "^1.5.6",
    "nsPopover": "^0.6.8",
    "angular-socialshare": "angularjs-socialshare#^2.3.1"
  },
  "overrides": {
    "fancybox": {
      "main": "./source/jquery.fancybox.js"
    },
    "jcf": {
      "main": [
        "./js/jcf.js",
        "./js/jcf.select.js",
        "./js/jcf.angular.js"
      ]
    }
  }
}
```

Gulpfile.js

```
// Utilities
var gulp = require('gulp');
var debug = require('gulp-debug');
var argv = require('yargs').argv;
var gutil = require('gulp-util');
var filter = require('gulp-filter');
var gulpif = require('gulp-if');
var runSequence = require('run-sequence');
var prompt = require('gulp-prompt');
var fs = require('fs');
var size = require('gulp-size');

// Server
var browserSync = require('browser-sync').create();
var historyFallback = require('connect-history-api-fallback');

// Files
var concat = require('gulp-concat');
var clean = require('gulp-clean');
var rename = require('gulp-rename');

// Styles
var sass = require('gulp-sass');
var sourcemaps = require('gulp-sourcemaps');
var autoprefixer = require('gulp-autoprefixer');
var cleanCSS = require('gulp-clean-css');

// Scripts
var uglify = require('gulp-uglify');
var jsHint = require('gulp-jshint');

// Images
var imagemin = require('gulp-imagemin');

// Deployment
var mainBowerFiles = require('main-bower-files');
var rsync = require('gulp-rsync');

try {
   // JSON file for environment-specific passwords etc.
   var envConf = require('./envConfig.json');
} catch(e) {
      console.log(e);
}

// Build environment variables
```

```
var productionDir = './production/',
	developmentDir = './',
	isProduction = false,
	isDebug = argv.debug ? true : false,
	basePath = developmentDir;

// Change environment to production if --production flag used
if(argv.production) {
	isProduction = true;
	basePath = productionDir;
}

// Filters
var filterJS = filter('**/*.js');
var filterAppJS = filter('src/**/*.js', { restore: true });
var filterCSS = filter('**/*.css');
var filterSCSS = filter('**/*.scss');

gulp.task('default', ['serve']);

gulp.task('serve', (isProduction ? null : ['js', 'sass']), function() {
	browserSync.init({
		server: {
			baseDir: basePath,
			middleware: [ historyFallback() ]
		},
		port: (isProduction ? 3010 : 3000),
		open: (isProduction ? 'local' : false),
		browser: ['google chrome', 'firefox' , 'safari'],
	});

	gulp.watch(['./scss/**/*.scss'], ['sass']);
	gulp.watch(['./src/**/*.js'], ['js']);
	gulp.watch(['./src/**/*.html', './index.html']).on('change', browserSync.re-
load);
});

gulp.task('bower', ['bower:js', 'bower:css', 'bower:scss']);

gulp.task('bower:js', function() {
	return gulp.src(mainBowerFiles())
		.pipe(filterJS)
		.pipe(gulpif(isDebug, debug({title: 'js'}))) // List files if --debug flag
		.pipe(concat(basePath + 'js/vendor.js'))
		.pipe(size({
			title: gutil.colors.red('JS: unminified')
		})
		.pipe(uglify())
		.pipe(size({
			title: gutil.colors.underline.green('JS: minified')
		}))
		.pipe(gulp.dest(basePath));
});

gulp.task('bower:css', function() {
	return gulp.src(mainBowerFiles())
		.pipe(filterCSS)
		.pipe(gulpif(isDebug, debug({title: 'css'})))
		.pipe(concat(basePath + 'css/vendor.css'))
		.pipe(size({
			title: gutil.colors.red('CSS: unminified')
		}))
		.pipe(cleanCSS())
		.pipe(size({
			title: gutil.colors.underline.green('CSS: minified')
		}))
		.pipe(gulp.dest(basePath));
});

gulp.task('bower:scss', function() {
	return gulp.src(mainBowerFiles())
		.pipe(filterSCSS)
		.pipe(gulpif(isDebug, debug({title: 'scss'})))
		.pipe(rename({
			prefix: '_'
		}))
		.pipe(gulp.dest(basePath + 'scss/vendors/'));
		// Insert file reference to scss/main.scss manually!
});

gulp.task('sass', function() {
	return gulp.src('./scss/**/*.scss')
		.pipe(gulpif(!isProduction, sourcemaps.init()))
		.pipe(gulpif(isProduction, sass({ outputStyle: 'compressed' }).on('er-
ror', sass.logError)))
		.pipe(gulpif(!isProduction, sass().on('error', sass.logError)))
		.pipe(autoprefixer({
          browsers: ['last 4 versions'],
          cascade: false
      }))
      .pipe(gulpif(!isProduction, sourcemaps.write(basePath)))
		.pipe(gulp.dest(basePath + 'css/'))
		.pipe(gulpif(!isProduction, browserSync.stream()));
```

```
});

gulp.task('js', function() {
	return gulp.src([
			'./src/**/*.js',
			(isProduction ? './js/*.js' : '')
		])
		.pipe(gulpif(isProduction, filterAppJS))
		.pipe(jsHint())
		.pipe(jsHint.reporter('default'))
		.pipe(concat(developmentDir + 'app.js'))
		.pipe(gulpif(isProduction, filterAppJS.restore))
		.pipe(gulpif(isProduction, uglify({ mangle: false })))
		.pipe(gulp.dest(basePath + 'js/'))
		.pipe(gulpif(!isProduction, browserSync.stream()));
});

gulp.task('images', function() {
	return gulp.src('./images/*')
		.pipe(imagemin())
		.pipe(gulp.dest('./images'));
});

gulp.task('clean', function() {
	return gulp.src(productionDir)
		.pipe(clean());
});

gulp.task('copy', function() {
	// Copy uncompiled files to production directory
	return gulp.src(
		[
			developmentDir + 'fonts/**/*',
			developmentDir + 'images/**/*',
			developmentDir + 'css/vendor.css',
			developmentDir + 'src/**/*.html',
			developmentDir + 'index.html'
		],
		{ base: developmentDir}
	)
	.pipe(gulp.dest(productionDir));
});

gulp.task('build', function(callback) {
	// Run build tasks in defined order
	runSequence(
		'clean',
		'copy',
		['sass', 'js'],
		'serve'
```

```
        );
});

gulp.task('deploy', function() {
        // Default options
        var rsyncConf = {
                progress: true,
            incremental: true,
            relative: true,
            emptyDirectories: true,
            recursive: true,
            clean: true,
            exclude: [],
            dryrun: argv.dry ? true : false
        };

        // SSH keys are needed to use rsync to push code into servers!
        if(argv.staging) { // Test server, --staging flag
                rsyncConf.hostname = envConf.staging.hostname;
                rsyncConf.username = envConf.staging.username;
                rsyncConf.destination = envConf.staging.destination;
        } else if(argv.production) { // Production server, --production flag
                rsyncConf.hostname = envConf.production.hostname;
                rsyncConf.username = envConf.production.username;
                rsyncConf.destination = envConf.production.destination;
        } else {
                throwError('deploy', gutil.colors.red('Missing or invalid target!'));
        }

        rsyncConf.root = 'production'; // production directory should not be copied
to server web root

        return gulp.src('production/**/*')
                .pipe(prompt.confirm({
                        message: 'You are about to push code to ' + (isProduction ?
'production' : 'staging') + ' server. Are you sure?',
                        default: false
                })).pipe(rsync(rsyncConf));
});

function throwError(taskName, msg) {
  throw new gutil.PluginError({
      plugin: taskName,
      message: msg
    });
}
```