Bachelor Informatica

UvA Universiteit van Amsterdam

Informatica — Universiteit van Amsterdam

Evaluation of JavaScript frameworks for the development of a web-based user interface for Vampires

Jaap Koetsier

June 8, 2016

Supervisor(s): Dr. P. Grosso, C. Dumitru MSc

Signed:

Abstract

Vampires is a framework that assists in finding the optimal combination of resources to use for execution of a set of independent tasks in a heterogeneous cloud environment. This thesis discusses the decisions made during the development of a web-based user interface for Vampires. This includes a thorough evaluation of client-side JavaScript frameworks, resulting in the choice of AngularJS to use as the basis of the Vampires user interface.

Contents

Chapter 1

Introduction

Cloud environments are often used to execute resource-demanding tasks. These cloud environments exist of a heterogeneous set of nodes, each with different performance and monetary cost attached. The user allocates resources in this cloud environment to execute these tasks based on the expected or advertised performance and cost. It is hard to know the exact performance in advance and the only way to compare the different cloud instances is to run the task on the different types of nodes. This is of course a costly approach when the completion time of a task runs into hours and the experiments will have to be repeated on a large number of different set-ups.

At the Systems and Network Engineering department of the University of Amsterdam (UvA), research is done regarding this problem using Bag-of-tasks (BoT) applications. BoT applications consist of a number of independent tasks that can be executed in any order. These tasks can be roughly divided in two groups and are either computationally intensive or data-intensive. The research group at the UvA focuses on the group of data-intensive tasks. In their paper [1] they propose a queueing theory to achieve a Pareto-optimal distribution of tasks in a cloud environment. The research is based on the theory that the network will become congested when data is transferred to the worker nodes from a remote data server. With parallelised execution of tasks there is a certain point at which there is no further improvement to be seen in allocating extra or more powerful worker nodes. Transfer of data here becomes the bottleneck. The paper proposes a queueing theory that takes this data transfer time into account. This theory proves to be more accurate in estimating the Pareto-optimal distribution of tasks over a set of heterogeneous nodes than traditional algorithms.

Following this paper, the Vampires framework [2] has been developed that runs a sample execution of a set of tasks. The outcome of this sampling run can be used to determine the optimal combination of resources to use for the execution of this Bag-of-tasks. The Vampires framework currently is a command line application that needs to be configured by editing configuration files. The focus of this thesis is the development of a web-based user interface for Vampires. This user interface should make the process of running Vampires easy to a user without any prior knowledge about the Vampires framework. The first question we will answer in this thesis is:

How to construct a user-friendly user interface for Vampires?

The Vampires user interface will be developed using a modern JavaScript framework. Over the last years JavaScript frameworks have become increasingly popular. There are currently dozens of JavaScript frameworks available to choose from. This constructs our second research question:

Which client-side JavaScript framework is the most suitable for the development of a user interface for Vampires?

In chapter 2 we start with the background on Vampires. Chapter 3 describes the requirements and approach in developing the Vampires user interface. Chapter 4 and 5 focus on the survey and evaluation of client-side JavaScript frameworks. Chapter 6 describes the global structure and elements of AngularJS. Chapter 7 explains the design decisions made during the development of the Vampires user interface. Chapter 8 concludes this thesis with our conclusions and future work.

Chapter 2

Vampires

The Vampires framework is designed to effectively estimate the execution time of data-intensive BoT applications on a variety of cloud resources. The framework is developed following the previous work of [3] on the Budget-aware Task Scheduler (BaTS). BaTS was designed as a response to the shift in paradigm in scientific computing. The approach in scientific computing used to be to deploy the execution of tasks on under-utilised systems. Using this approach, the execution of tasks happens on a best-effort basis without any guarantee of performance. This approach is often free of charge and the common way of deployment was to grab as many worker systems as possible to reach the highest possible throughput.

With the emergence of cloud computing platforms such as Amazon EC2 [4] this approach is no longer justifiable. Cloud computing platforms offer a wide variety of instance types with different specifications. When the user wants to execute a computationally demanding task she allocates a number of instances based on the advertised cost and performance. These instances are commonly charged per time interval, typically an hour. There are currently 40 different types of instances available on Amazon EC2 [5] in 10 categories. It is hard, if not impossible, to determine up front what are the most cost-effective instances to use to execute a certain task. The exact specifications of the instances are hidden to the user, only providing basic information as the number of (virtual) CPU's and the amount of memory. Networking performance is specified as 'low', 'moderate', 'high' or '10Gbit' and levels like 'low to moderate' as sub levels. It may be clear that choosing the right type of instance for a task is difficult with this limited information.

BaTS focuses on the optimal scheduling of computationally intensive tasks in a cloud environment. It allocates a number of cloud instances to run BoT's on and adapts the allocation regularly based on the measured execution times, in order to achieve optimal performance. BaTS takes into account the instance cost and makes the best effort to respect a given budget limitation.

In contrast to BaTS, Vampires takes into account the impact of data transfer time on the total makespan of a task. When a large amount of data needs to be downloaded before starting the computational part of a task, the cloud instance is not using its full computational power. A high ratio of download time to total makespan significantly impacts the observed performance of a particular instance. Vampires focuses on these data-intensive tasks by running a number of samples from a BoT on different instance types. It monitors CPU usage, network usage and total makespan of the task. With these results, the user is able to make an educated decision of the resources to use for execution of the entire BoT.

Chapter 3

Improvements to Vampires

Currently Vampires is as an application that is run from the command line. Executing a task is done by defining the task in the configuration files and running the application. The results from the execution are exported to a large JSON file. This file returns a number of metrics such as CPU usage, network usage and execution time, but this JSON data is hard to read. Running an execution in Vampires from the command line now is a tedious task.

This thesis is about improving Vampires with a web-based user interface. The user interface will assist the user with defining tasks and selecting the resources to include in a sample run.

The following main requirements are defined for the development of the user interface:

1. The user interface should be web-based and developed using a modern JavaScript framework.

2. The user interface must be usable without any background knowledge of the technical details of the Vampires framework.

3. After the sampling phase the user must be presented with the results of the execution in a clear an unambiguous way.

4. After having analysed the results of the sampling phase, the user must be able to make a selection of resources to continue with the execution of the full task.

3.1 JavaScript Framework

The first requirement is the use of a modern JavaScript framework. There are many JavaScript frameworks available that could be used for the development of the Vampires user interface. The question that arises here is which framework to use. A quick look on the internet learns us that there are two heavily discussed client-side JavaScript frameworks at the moment: AngularJS and ReactJS. Although there is an abundant amount of information to find regarding these and other frameworks, most information is biased and lacks founded argumentation, coming from users of one of either frameworks. Due to the fact that the world of JavaScript frameworks is young and moving rapidly, there is no relevant scientific research available regarding this subject. In this thesis we will conduct our own research. We will look for the most popular client-side JavaScript frameworks currently available and evaluate them thoroughly based on scientifically proven software quality metrics.

When we talk about JavaScript frameworks in the remainder of this thesis, we will implicitly refer to *client-side* JavaScript frameworks unless explicitly stated otherwise.

3.2 User Interface

Requirements 2, 3 and 4 are requirements regarding the design of the user interface. To create a user interface that is usable without any background knowledge of the technical details of the Vampires framework, we will limit the number of options presented to the user. The user is only asked for the minimum input required to start an execution.

We are going to display the available statistics resulting from the sampling run in tables and graphs where appropriate. Statistics are returned at different levels of detail, from the aggregated statistics of all clients (worker nodes) to the statistics per client. These levels will be presented to the user as a tree-like structure, giving the user the possibility to view the statistics in more detail as she pleases. Using a tree structure is the most intuitive way to implement this functionality because of the tree-like structuring of clients in groups. A table on the results page will show the cost and performance of each resource type. This will give the user the most important information to base the resource selection on and will thus be presented on top of the results page. Selecting resources and initiating the full execution will be possible to do here as well.

Chapter 4

Survey of JavaScript Frameworks

4.1 History of JavaScript

JavaScript was developed in 10 days in 1995 by Brendan Eich under the name Mocha, but with its initial release it was named LiveScript. It was soon renamed to JavaScript when Sun and Netscape started shipping JavaScript with the Netscape browser in december 1995 [6, 7]. The only portable programming language at the time that ran in the client's browser was Java. Java was a heavy and complex language, aimed at professional programmers. Netscape was looking for a lightweight language that was interpreted, rather than compiled, and that was accessible to less experienced users.

In 1997, the ECMAScript standard ECMA-262 was defined based on JavaScript to make it possible for browser developers other than Netscape to implement the standard. Nowadays the ECMAScript standard is a standard on its own. A number of scripting languages such as JScript and JavaScript are implementations of the ECMAScript standard. After years of slow developments of the ECMAScript standard, JavaScript suddenly became a hot technique with the appearance of AJAX [1] in 2005 [8]. The work on ECMAScript continued again after the last major update dating from 1999 (ES3). Since then, the ECMAScript standard is actively developed, catalysed by the upcoming *server-side* framework Node.js and a large number of client-side frameworks. Currently, JavaScript is based on the last ES6 (also known as ES2015) standard [9].

[1]Asynchronous JavaScript and XML, a combination of techniques used to fetch data from a server asynchronously and update partials of the web page

4.2 JavaScript Frameworks in General

Web pages have long been static pages with little dynamic content within the page. Web servers used to construct the entire HTML file at once, which was then downloaded and presented by the client's browser. Clicking a link caused a new page to be fetched from the web server. The paradigm of a web page has changed rapidly over the past ten years, primarily due to the introduction of some popular JavaScript libraries, frameworks and techniques.

With the advent of AJAX in 2005 and jQuery[2] a year later [10], it became possible to update only parts of a web page without reloading the whole page. jQuery made it easy to manipulate the HTML DOM and dynamically update parts of the page with data fetched from a server. With AJAX and jQuery it was possible to create interactive web pages, but nothing like the singe-page applications (SPA's) we see today. In jQuery, one has to manually find every element and define every action to take on that element. A JavaScript file with jQuery mostly contained a series of event listeners, linked to HTML elements by element type, CSS class or element id.

This brings us to the type of client-side JavaScript frameworks (JSF's) popular today, which emerged around 2009. JSF's today provide an extensive library that takes care of the data binding between HTML files and JavaScript code, instead of having to explicitly update elements in the HTML DOM as with jQuery. JSF's today ease the process of fetching data from a server using HTTP requests and add route handlers to the framework. Route handlers show the visitor of the web page content based on the address in the address bar, without actually changing the page like in classic web applications. These properties of modern JSF's make it easy to develop a client-side run application that is interactive and responsive to the user. This approach is at the same time economical in terms of data transfer and server usage. A server sends out the application once and provides the user only with basic data on subsequent requests. The server does not have to compute every page requested and only sends out the minimum amount of data needed by the client-side application.

4.3 Considered frameworks

JavaScript frameworks are a hot item today. New frameworks are coming to view in rapid pace and it would be undoable to compare them all. In this thesis, we will limit the focus on the most popular frameworks at the moment.

A good place to find out which JavaScript frameworks are popular at the moment is the internet.

[2]A JavaScript library focused primarily on DOM[3] manipulation and making HTTP requests

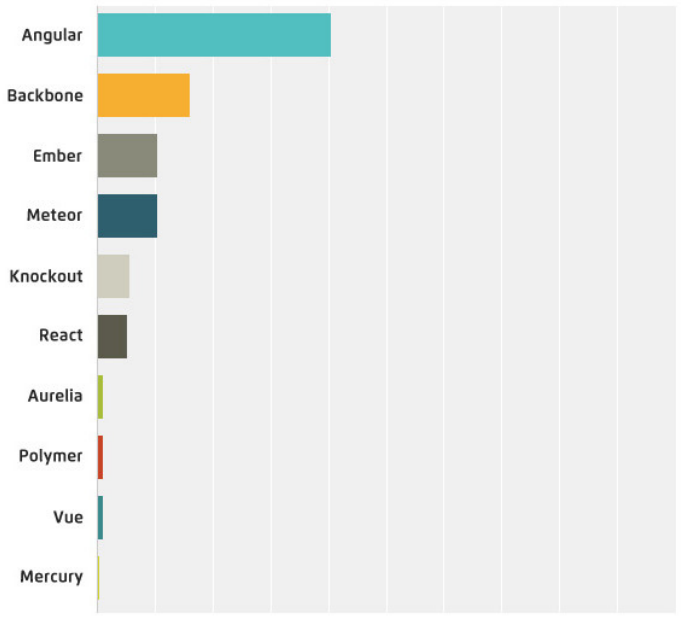


Figure 4.1: The top 10 used JavaScript frameworks according to codeanywhere [11]. April 2015.

Figure 4.1 shows a graph adopted from *codeanywhere* [11], dating from April 2015. In April 2015, the major framework in use was AngularJS, with Backbone, Ember, Meteor, Knockout and React following at a distance. The four frameworks at the bottom were available, but barely used.

Searching the internet for recent articles ([12, 13, 14, 15], among others) about JavaScript frameworks, we see that there is a lot of attention regarding ReactJS and Vue.js, for example. This attention is not consistent with the graph from codeanywhere, where Vue.js is barely used and ReactJS only closes the top of the list.
To get a better view of the current popularity of JavaScript frameworks, we have created a list of all frameworks that are mentioned frequently on the internet, augmented with the frameworks listed on the website TodoMVC[4] [16].

The most discussed frameworks on the internet are listed in tables 4.2 and 4.1, along with their popularity on GitHub [17] and StackOverflow [18]. With this data we are trying to get a better view of the current world of JavaScript frameworks, using the following assumptions:

[4]TodoMVC is a website that gives an overview and short introduction to the major JavaScript frameworks available at the moment. It does so by constructing the exact same to-do list application with each framework. The website is updated on a regular basis and therefore gives a good view of the current environment of JavaScript frameworks

1. **Popular frameworks have a large and active community on GitHub.** People follow repositories on GitHub because it contains something of interest to them. There are a variety of possible reasons to follow a repository on GitHub: The GitHub user wants to be able to find the repository back easily for cloning (framework *user*) or the user wants to be kept up-to-date on the developments in the repository (*interested* in the repository).

2. **Popular frameworks receive more attention on StackOverflow.** The more a framework is used, the more likely it is that users have questions about it and reach out to a platform such as StackOverflow. This assumption is less reliable than assumption 1, because there are many reasons to post a question on StackOverflow. A higher number of questions about framework A over framework B does not necessarily mean that framework A is better than framework B. A high number of questions could be the effect of a poorly written documentation or questionable design decisions within the framework itself. However, the number of questions can give us an indication about the activity within the community.

	Github			
	Watchers	*Stars*	*Forks*	*Contributors*
AngularJS	4291	49418	23767	1464
ReactJS	3117	42441	7204	704
Meteor	1931	33944	4168	288
Backbone.js	1674	24824	5545	288
Vue.js	1088	19270	1926	67
Ember.js	1144	16246	3483	584
Polymer	1038	15092	1496	84
Angular 2	1561	12137	3169	247
Knockout	606	7433	1281	62
Aurelia	444	6160	293	31
CanJS	128	1342	365	130
Dojo	172	829	417	83

Table 4.1: Popularity of JavaScript frameworks on GitHub, sorted by the number of stars, as of 8 May 2016.

	StackOverflow	
	Questions	*Week*
AngularJS	173849	1803
Angular 2	7565	483
ReactJS	15431	441
Meteor	21307	171
Ember.js	18815	68
Knockout	16156	54 (month: 185)
Vue.js	1086	44
Polymer	4936	43
Backbone.js	19726	27 (month: 145)
Aurelia	765	19
Dojo	8539	17
CanJS	200	0 (month: 0)

Table 4.2: Popularity of JavaScript frameworks on StackOverflow, sorted by the number of questions in the week preceding 8 May 2016

Table 4.1 shows us a different trend compared to the graph from codeanywhere in figure 4.1. In the graph from April 2015, AngularJS is the most used framework, with Backbone.js following with less than half of the users. ReactJS has a fraction of its users, and Vue.js is still infant. Now, a year later, ReactJS seems to have reached a popularity close to that of AngularJS. Vue.js, with just 67 contributors, has gained a high number of followers.

Table 4.2 shows the activity regarding the JavaScript frameworks on StackOverflow ordered by the number of questions in the week preceding 8 May 2016. The number of questions in the first column is an all-time aggregate and might give a wrong view of the actual popularity. Frameworks that have been around for a longer period of time may naturally have more questions on StackOverflow. This is why the second column is included, listing the number of questions asked in the last week. Unfortunately, StackOverflow does not provide the questions for every tag (subject) over the last month. A period of a month would give us a better indication, because it is less prone to fluctuations. Where available and needed for clarification, the questions over the last month are added between parentheses. An example is the ratio of questions asked in a week and in a month between Knockout and Backbone.js. The number of questions per month should give us a more reliable view. Either Knockout has received an unusual number of questions in the week of measurement, or there was less attention than usual towards Backbone.js.

In this table we see that there is significant activity around AngularJS and ReactJS. Angular 2, although still under development and not officially released yet, already has a lively community of users on StackOverflow. Meteor seems to have an active community as well. We have to be careful drawing conclusions on the order of frameworks in the remainder of the list. The frameworks in the remainder of the list are close together and this list is, as we saw, susceptible to fluctuations.

4.3.1 Frameworks to consider

Based on the previous findings, there are a number of frameworks that are clearly popular. AngularJS and ReactJS are two frameworks we need to consider. They are both mentioned in every recent article about JavaScript frameworks and have an active online community. Although Meteor and Backbone.js are both popular, they will be excluded from the framework evaluation. A quick look at their websites learns that they both miss their own templating system and will need to work alongside a JavaScript framework that provides that functionality. They both are often used together with AngularJS or ReactJS and are therefore not suitable for our comparison.

Ember.js will be included in the evaluation, because it was popular in April 2015 and is still heavily watched and discussed. We conclude the list with Vue.js. Vue.js is mentioned as a hot new framework on various places on the internet [19][13] and gains popularity quickly.

The complete list of frameworks we will consider:

- AngularJS
- ReactJS
- Ember.js
- Vue.js

4.3.2 AngularJS

AngularJS [20] originates from Google and was one of the first JavaScript frameworks in its form, released in October 2010. The idea behind AngularJS is to provide an abstraction level in making dynamic web pages, abstracting the business logic from the HTML. Angular extends HTML with Angular-specific tags (directives) and uses JavaScript to act on these tags. An AngularJS application keeps the business logic in JavaScript separated from the HTML. Angular provides two-way binding between variables used in HTML and the JavaScript controllers.

Currently, Angular 2 [21] is being developed and is in its final stage before the final release. The exact date is still unknown, but is expected to be within months from now [22]. Although Angular 2 is a total rewrite of Angular rather than an update, migrating from Angular 1 to Angular 2 should be easy according to its website.

One of the major improvements in Angular 2 is the support for the use of TypeScript. TypeScript is a superset of ES6 (see 4.1) and is compiled to JavaScript. It introduces object-oriented programming and static typing, two welcome additions to a JavaScript environment.

4.3.3 ReactJS

ReactJS [23] is another popular JavaScript framework [5], released in March 2013. Developed by Facebook and used extensively on both Facebook and Instagram, this framework gained popularity quickly. ReactJS focuses on performance. A standard approach for many JavaScript frameworks and libraries is to update and redraw the DOM as a whole when one part is updated. ReactJS uses what it calls a Virtual DOM. Changes in the DOM are rendered to a Virtual DOM before rendering the real browser DOM. ReactJS compares the new Virtual DOM with the old Virtual DOM and only updates the changed parts in the real browser DOM [24].

Another React-specific feature is the use of JSX files. JSX files provide the ability to combine JavaScript and HTML in one file. In a JSX file, the developer writes the business logic of an application and the HTML output it generates. Using these JSX files, the `index.html` file might contain nothing more than script imports. All HTML will be rendered by ReactJS from the JSX files.

4.3.4 Ember.js

Released in December 2011 by Yehuda Katz, Ember.js [25] looks a lot like Angular. Ember.js provides a similar templating system and two-way data binding between the templates and the controller. Ember.js also allows the developer to create application-specific tags called Handlebars, which are essentially components that are included by using its HTML tag. These components can be defined in separate `hbs` and `js` files, containing the HTML template and logic for the Handlebar.

In Ember every route has a model attached that defines the route-specific data. These models can be automatically linked with a back-end using a REST API, providing the back-end's REST API is designed following Ember's strict format.

4.3.5 Vue.js

Vue.js [26] is the new kid in town, released in February 2014 by Evan You. Although Vue is relatively young, it has a growing user base, catalysed by the success of the PHP framework Laravel. Laravel recommends using Vue.js in their Laracasts.

Vue uses a template engine similar to that of Angular and Ember and provides two-way data binding between the HTML template and controller. Controllers in Vue are elegantly written[6] and provide a no-nonsense way to bind the HTML template and controller. Vue provides a way to define components,

[5]Although Facebook calls ReactJS a library instead of a framework, I will use the word framework in the remainder of this text to avoid confusion

[6]Elegant code is the key focus of Laravel. This could explain the happy marriage between Vue.js and Laravel.

similar to the Handlebars in Ember and directives in Angular. These components can be defined in `.vue` files, combining JavaScript and HTML in a single file, similar to that of the JSX files in React.

It is clear that Vue is based on a combination of components from both Angular and React.

Chapter 5

Evaluation of JSF's for the Vampires front-end

The selected JavaScript frameworks will be evaluated using the ISO 25010:2011 [27] model for software product quality, based on the paper of Boehm [28]. These are generic guidelines that can be applied to a variety of areas in software product development. The first two sections of this chapter start with a short overview of the characteristics as defined in the ISO standard and how we are going to apply these to JavaScript frameworks. The remainder of this chapter explains the results of the application of these characteristics to the four JavaScript frameworks.

5.1 ISO 25010:2011 characteristics

The model of software product quality as defined in ISO 25010:2011 contains eight characteristics:

Reliability The reliability of a product is based on the *maturity*, *availability*, *fault tolerance* and *recoverability* of the product.

Operability The degree to which a software product is *easy to use*, *easy to learn* and *attractive to use.*

Performance Efficiency The *response time*, *resource utilisation* and *capacity* of a software product.

Security The security of the data processed and the processing of the data: *Confidentiality*, *integrity*, *non-repudiation*, *accountability* and *authenticity.*

Compatibility The degree to which software is capable of *co-existing* with other software sharing the same resources and to which extend it is *interoperable* with other software.

Maintainability Maintainability is about the software's *modularity*, *reusability*, *analysability*, *modifiability* and *testability.*

Transferability The degree to which a product can be transferred from one system to another. *Portability*, *adaptability*, *installability* and *compliance* are the keywords here.

Functional Suitability The degree to which a product meets the requirements stated by the user of the product. This contains *functional completeness*, *functional correctness* and *functional appropriateness.* Is the functionality complete? Are the results correct and accurate? Does the functionality contribute to the means of the product?

In terms of JavaScript frameworks, not all of these characteristics are applicable. Security for example, is impossible to guarantee in a client-side application. All source code is downloaded by the client and executed on the client's computer. Every security measure you would implement in the code could be circumvented by a malevolent user. Security must therefore be implemented on the server-side.

5.2 Application to JavaScript frameworks

Based on the ISO 25010:2011 product quality model we have defined the following JavaScript framework-specific characteristics to use for evaluation:

Maturity (Reliability) How mature is the framework under observation?

Ease-of-use (Operability) What is the learning curve of the framework under observation? Are its API, documentation and structures clear and easy to understand?

Performance (Performance Efficiency) How does the framework perform in terms of speed and resource usage?

Browser support (Compatibility, Transferability) Which browsers does the framework support? Does it function correctly on a wide range of browsers?

Modularity/Reusability (Maintainability) Is it easy to build modular software using the framework? Can we easily construct reusable parts?

Testability (Maintainability) Does the framework come with a test suite, and is it easy to use?

Based on the characteristic of *functional completeness*, we can define two JSF-specific characteristics:

Routing (Functional Suitability) In single-page applications routing between views is handled by the JSF. Does the framework have built-in routing functionality?

Templating (Functional Suitability) How does the JSF build the HTML page? Is this clear an concise?

5.3 Results

This section evaluates the four frameworks on the basis of the characteristics described in the previous chapter. We will assign 1 to 4 points to each framework for each characteristic. The best scoring framework gets 4 points and the worst gets 1 point. When two frameworks end up as equally good they both get the number of points based on that position. The other frameworks will not move a position up, but will get the points assigned as if the two frameworks above had taken two separate places.
We will further note that the evaluation of AngularJS will only be done on version 1. Version 2 is not officially released yet and will not be considered.

5.3.1 Maturity

The initial release dates of the evaluated JSF's range from October 2010 (AngularJS) to February 2014 (Vue.js). In terms of software maturity, this could make a major difference. In this section, we will evaluate the maturity of the four JSF's with the Capability Maturity Model for Software (CMM) [29]. The CMM defines five levels of software process maturity, used to identify the degree of reliability, stability and overall quality of software. The five levels of software process maturity are as follows:

Initial The initial stage of software development. The software in question is largely undocumented, ad hoc and 'occasionally even chaotic'. Success depends on individual effort.

Repeatable The software is documented and basic project management processes are established.

Defined The software is well-documented and is standardised and a large group of people can depend the standard.

Managed A high level of control is reached over the software. The software process is monitored and measured using defined metrics.

Optimising The software has reached the highest level of its maturity. It is constantly monitored and improved.

AngularJS

When we look at the AngularJS changelog on GitHub [30], we see that the API is not changing anymore. The updates contain bug fixes, performance improvements and features extending existing features (rather than new features). AngularJS is maintained and optimised heavily by the Google core team. No new features are added and the framework has stabilised. AngularJS is in the *optimising* stage of maturity and has become a solid framework.

ReactJS

Both the changelog on the ReactJS GitHub repository [31] and the blog on the ReactJS website [23] mention significant changes, in contrast to AngularJS. The ReactJS API is still undergoing changes, but it is largely defined and established. The following quote from the React blog, dated 7 April 2016, tells more about the level of maturity of ReactJS:

> *.., we understand that in order to receive more community contributions like Michaels, we need to communicate our goals and priorities more openly, and review pull requests more decisively. As a first step towards this, we started publishing React core team weekly meeting notes again. We also intend to introduce an RFC process inspired by Ember RFCs so external contributors can have more insight and influence in the future development of React. ...*

Based on the changelog and quote, ReactJS is at the maturity level 3, *defined*. The quote from the React blog shows us that ReactJS is working towards the *Managed* stage.

Ember.js

Like AngularJS, Ember.js is on a high level of maturity. According to the blog on its website [25], Ember.js has a core team that focuses on the direction and vision of the Ember.js project, setting priorities and managing the release schedule. The core team sets up subteams to improve specific parts of the Ember.js project. Based on a quote from the CMM paper [29], we can state that Ember.js is at maturity level 5, the *optimising* stage.

> *At the Optimizing Level, the entire organization is focused on continuous process improvement. The organization has the means to identify weaknesses and strengthen the process proactively, with the goal of preventing the occurrence of defects.*

Vue.js

Vue.js is the youngest of the four JSF's under observation and is also the least mature of the four. Vue.js is being developed by one single person with help from the community. With every new version there are new features or changes to the API. The Vue.js project is well documented and well defined, but it still is a large hobby project. Many people use Vue.js, but the development is far from managed. Therefore Vue.js finds itself on maturity level 3, *defined*.

Conclusion

Based on the maturity levels as defined by the CMM, the final score is as follows:

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Maturity	4	2	4	1

5.3.2 Ease-of-use

This section focuses on the ease-of-use of the observed frameworks. How easy is the framework to get started with? How steep is the learning curve? Is the documentation comprehensive?

AngularJS

Angular has a steep learning curve, according to many articles and complaints on the internet [32, 33, 34, 35]. Angular's concept of a *scope* is something one needs to get used to, as well as the interaction between directives. As a beginner starting out with Angular, the system of dependency injections throw errors with stack traces to the Angular core code itself, providing little useful information. Angular's API documentation is extensive, but it is easy to get lost in the many possibilities it offers.

Angular is extremely powerful and large in terms of functionality, but its many possibilities make it harder to get past the basic functionality.

ReactJS

React is much easier to learn. The React website comes with a clear tutorial and *getting started* section on their website that really gets you started. The React API is small compared to Angular and the function names in React are clear and state exactly what the function will do. As React code is just common JavaScript without React-specific constructs, everyone with a basic knowledge of JavaScript should be able to have a solid understanding of React in no-time.

Ember.js

Getting started with Ember should be easier than with Angular. Ember provides a clear tutorial and *getting started* section on its website. Its documentation is well written and logically structured. A big plus for Ember is its command line interface, Ember CLI. It helps setting up everything when creating a new application, controller or component, increasing productivity.

Vue.js

Vue should be easy to get started with using its clear and short *getting started* section. The rest of its documentation is just as short, but lacks clarity. It is hard to browse through its documentation due to its seemingly random structure. When one manages to find a particular functionality within the documentation, it is not always possible to read the description of that particular functionality without having the entire page as context. Vue's syntax is easy to get used with.

Conclusion

React is the clear winner here. Its API and documentation are both clear, leaving no room for ambiguity. As a good second comes Ember with its command line tool and well written documentation. Vue ends up third with work to do on the documentation, but a clear coding style. Angular comes last with a steep learning curve compared to the other frameworks.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Ease-of-use	1	4	3	2

5.3.3 Performance

Performance is important in client-side JavaScript applications. Today, web applications are not rendered by high-performance web servers any more, but the application is run in the user's browser. The developer does not know the exact specifications of the systems an application will run on. These systems can be anything between a few years old mobile phone and the last desktop computers. To make sure an application runs smoothly on a wide range of systems it is important to make optimal use of resources.

To test the speed of the four frameworks, we have adopted the benchmark tool from [36]. This benchmark tool measures the time for a number of DOM manipulation events to complete using the different frameworks. Because DOM manipulations are the main job of client-side JSF's, this should give a good indication of the speed of the frameworks.

The DOM manipulations included in the benchmark act on rows of a table. The manipulations are adding 10 rows to the table, adding 1000 rows to the table, updating every 10th row in the table, selecting a row in the table and deleting a row from the table. All experiments are repeated 5 times and averaged.

The results of the benchmarks are shown in figure 5.1. The same data is replicated for clarity without the *Add 1000* benchmark in figure 5.2

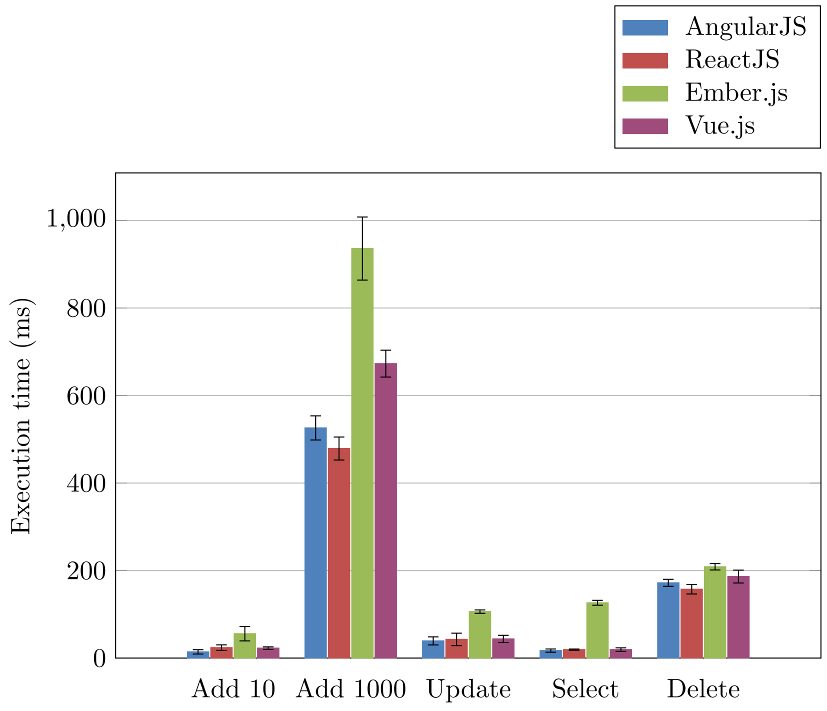


Figure 5.1: Results of the speed test, based on DOM manipulations on a table. The performed actions are adding 10 rows, adding 1000 rows, updating every 10th row, selecting a row and deleting a row. *(Experiments executed on Chrome 50.0.2661.102 on Fedora 23 64-bit)*

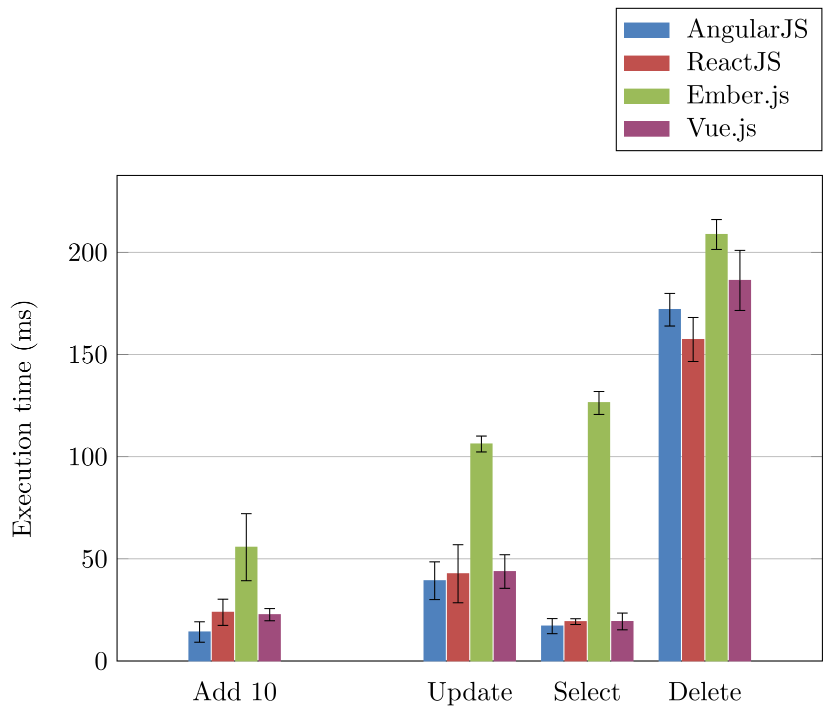


Figure 5.2: Results of the speed test. These are the same results as in figure 5.1, but the *Add 1000* metric is left out for clarity.

The results show that Ember.js has the worst performance over all tests. Angular is the fastest with small DOM updates that take less than 50ms. At the *Add 10* metric Angular is clearly faster than React and Vue, but for the *Update* and *Select* metrics they all perform similar. React's Virtual DOM seems to pay off when the DOM updates get larger and more data has to be updated. React performs best at both the *Delete* and *Add 1000* metrics.

We have tested the memory usage of the four frameworks using the Chrome Profiler. The Chrome Profiler lets you snapshots of the JavaScript heap of a web page. We have taken heap snapshots at two moments for each framework. One snapshot is taken right after the page is loaded, before any action is taken on the page. The second snapshot is taken after performing 5 *Add 10*, 5 *Add 1000* and 5 *Update* actions from the speed benchmark tool. The results are listed in table 5.1.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Initial heap size (MB)	8.7	8.3	13.6	7.6
Heap size after actions (MB)	15.4	15.1	38.3	16.4

Table 5.1: Size of the JavaScript heap per framework in Chrome after loading the page and after repeating 5 *Add 10*, 5 *Add 1000* and 5 *Update* actions.

Angular, React and Vue have a similar memory footprint. The initial memory footprint of Vue is smaller than that of Angular or React. After performing the same actions with each framework, Vue's heap size is larger than that of Angular and React, but the overall results are similar.

Ember here performs significantly worse. Its initial footprint is 56% larger than that of Angular, the largest heap size after Ember. After performing actions on the page Ember's footprint is 134% larger than that of Vue, now the largest heap after Ember.

Conclusion

Ember performs worst at both the speed and memory usage. Angular, React and Vue show comparable memory usage.

The best in the speed metrics is React. It is not the best performing framework over all speed metrics, but it clearly performs better with larger DOM updates. A user will not notice the difference between a 14ms response time (Angular, *Add 10*) or a 24ms response time (React, *Add 10*), but the user will notice a difference of hundreds of milliseconds.

Based on the speed test, React is the best in this section, followed by Angular and Vue. Ember ends last as the framework with the worst performance over all tests.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Performance	3	4	1	2

5.3.4 Browser support

It goes without saying that a JavaScript framework needs to be supported by a wide range of browsers. If the application is accessed on a browser that is not compatible with the framework, the application will simply not work.

All four frameworks claim to support all modern browsers (Chrome, Safari, Edge, Opera and Firefox) and Internet Explorer 9 and up.

Conclusion

Browser support is good for all frameworks under observation. Therefore there is no difference to be seen among the frameworks.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Browser support	4	4	4	4

5.3.5 Modularity

To keep large software applications maintainable, understandable and easily testable, software should be modularised. Pieces of code that provide a certain functionality should be placed in separate modules to provide an abstraction to the rest of the application and to be reusable throughout the project. Creating modules improves maintainability and testability. A well modularised design keeps the developer from violating the DRY-principle[1] [37].
This section focuses on the level of modularity of the observed JSF's. We will not focus on the actual structure of the application to be built with the JSF, as this is up to the developer. Instead we focus on the tools the JSF provides that enable the developer to write a well modularised application with the framework.

AngularJS

Angular is developed with modularisability in mind. Angular has a clever way of automatically injecting dependencies such as services and directives in controllers. Everything can be modularised and extracted into separate modules. The directives and scope systems give the developer full control over how to structure the application. The disadvantage of this is that one sometimes needs to dive deep into the documentation to really understand how this system of directives and scopes are interacting, as it is not always intuitive.

ReactJS

React consists completely out of *components*. Components define an HTML element and all its functionality within one JSX file. These components can be easily reused and nested within other components, in the same way as HTML elements are nested. With these components, React forces the developer to think about and implement a certain level of modularity. *Mixins* are provided to easily share common functionality between components. The simple interaction of components and mixins makes React powerful in terms of modularity.

Ember.js

The structures Ember provides to achieve modularity are similar to that of Angular. The difference lies in the freedom the developer gets from Ember.

[1]Don't Repeat Yourself

Ember is more limited and the developer is pushed towards using the structures as defined by Ember. This makes the interaction between structures easier to understand, but limits the developer in its abilities the same time.

Vue.js

Vue is built with the best parts of Angular and React in mind, which is clearly visible when we look at the structures Vue provides. Vue provides exactly the same structures as Angular, plus the *components* and *mixins* as used in React. With all these structures it is easy to build a modularised application with Vue.js.

Conclusion

All four frameworks provide structures to achieve modularity in the application to build. Although Ember.js puts more constraints in the way this is done compared to the other frameworks, the same level of modularity can be achieved.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Modularity	4	4	4	4

5.3.6 Testability

As client-side JavaScript applications are moving a great deal of application logic from the web server to the client's browser, the client-side application gets more complex. To ensure the application behaves as expected once delivered, we will need to have it thoroughly tested. It is unrealistic to do this by hand every time a new version of our application gets released for production. As the application grows larger, testing by hand will become labour intensive and unreliable. In this section we will explore how testable the frameworks under observation are in terms of unit testing and end-to-end testing.

AngularJS

Angular is designed with testability in mind. Unit-testing can be done with common JavaScript testing libraries such as Karma and Jasmine. Angular has its own library called *ngMock* to make it easy to inject services and dependencies during testing.

For end-to-end testing, Angular provides its own test runner called Protractor. This tool simulates user actions in all major browsers.

ReactJS

Running unit tests in React is easy. Because a React application is entirely defined within JSX files, all actions and DOM manipulations are defined within the same file as the HTML. React provides the *ReactTestUtils* tool kit that can be used with a number of JavaScript test frameworks.

End-to-end testing in React can be done using common JavaScript tools such as Nightwatch.js or Capybara, or even Angular's Protractor.

Ember.js

Ember provides basic testing functionality using third-party JavaScript testing tools and its own test helpers. The Ember documentation contains an entire chapter about testing and despite the fact that the documentation states that testing is one of their core functionalities, Ember has not made it as easy as in Angular or React.

Vue.js

The documentation of Vue contains one paragraph about unit testing where it mentions that it is possible to unit test Vue.js with Karma, including two minimal examples. At Vue, testing is clearly not a focus point.

Conclusion

Angular is clearly the winner here. Testability in Angular is a core principle, with everything well thought out and documented and even providing its own end-to-end test runner. React ends up as a close second, providing the tools needed to test React, but without the extra effort to make it as well thought-out as in Angular. Ember ends third providing common test functionality, and Vue ends up last because testing is barely mentioned in their documentation.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Testability	4	3	2	1

5.3.7 Routing

In single-page applications content is shown based on the route visible in the browser's address bar. The JSF in question should provide routing functionality to be able to handle single-page applications.

AngularJS

Angular supports routing with its native *ngRoute* module, which needs to be declared as an application dependency.

ReactJS

ReactJS does not come with native routing. However, there is a library *React Router* available from the React Community which supports all functionality a router needs.

Ember.js

Ember comes with native routing implemented.

Vue.js

Vue supports routing with the native *vue-router* library.

Conclusion

All frameworks support routing, albeit using an external library in the case of React. Because React's routing library is not a library from React itself and thus not supported by React, React here will get three point. The other frameworks receive four points.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Routing	4	3	4	4

5.3.8 Templating

Templating is an important aspect in the case of JSF's. On one hand we have the HTML DOM which makes up the page, and on the other there is the JavaScript code that acts on the HTML DOM. How are these two interconnected, and is there a clear separation of concerns? Do we have to mix JavaScript in the HTML code, making the HTML unreadable, or the other way around? How is the data binding between HTML and JavaScript? This section focuses on these questions.

AngularJS

Angular's HTML is completely defined in HTML files. The application logic is separated into JavaScript files. The connecting element is the concept of *scope*, which binds the values in HTML to JavaScript and vice versa.
The Angular-specific HTML elements and attributes that control the data shown within the template are easily readable and understandable. However, Angular does not prevent the developer to write logic within the HTML elements, which could make the template unreadable. It is up to the developer to keep the logic in the controller as much as possible.

ReactJS

ReactJS has no separation of JavaScript and HTML. The initial HTML file that gets loaded by the browser is an empty skeleton with JavaScript imports. The page gets build up from JSX files, which contain both the JavaScript logic and HTML. This approach has both pros and cons. Working on components of the application is easy, because everything is in one place and the relation between values in the JavaScript logic and the HTML snippets are clear to see. The disadvantage of this approach however, is that it is hard to see the complete picture of the HTML DOM without rendering it. Because the components of the application are usually small, it is hard to see the relation to the complete HTML document from this perspective.

Ember.js

Ember templating works similar to that of Angular. The HTML and JavaScript logic are separated and Ember provides two-way data binding as well. The one major difference is the syntax of the template tags. Where Angular adds attributes to existing HTML elements to extend the HTML with extra functionality, Ember introduces new tags which are often written as extra lines. This creates a longer HTML file with poor readability. In contrast to Angular, Ember forces the developer to put all the logic inside JavaScript files.

Vue.js

The Vue templating system works just like Angular and even uses the same style of HTML elements and attributes. With Vue *components* it is possible to create the same effect as with React JSX files, although the resulting files are nowhere as readable as React's JSX files. React clearly separates the HTML and JavaScript within the files, but due to syntax highlighting in most IDE's, the Vue templates included in normal JavaScript end up as plain strings, which are hard to read.

Conclusion

Angular and Vue are the best when it comes to templating. The syntax of Ember is not as clean as that of Angular and Vue. Although Ember forces the developer to keep logic out of the template files, it is up to the developer to leave the logic out in Angular and Vue, and that is easily done. The all including JSX files of React have the advantage of having everything in one place, but it does not directly give the developer a better overview of the application. Using a split screen with on one side the HTML template and on the other side the JavaScript logic works just as well, but the overview of the DOM remains visible.

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Templating	4	1	2	4

5.4 Conclusion

The total score of all eight metrics is as follows:

	AngularJS	**ReactJS**	**Ember.js**	**Vue.js**
Maturity	4	2	4	1
Ease-of-use	1	4	3	2
Performance	3	4	1	2
Browser support	4	4	4	4
Modularity	4	4	4	4
Testability	4	3	2	1
Routing	4	3	4	4
Templating	4	1	2	4
Total	**28**	**25**	**24**	**22**

AngularJS scores the maximum number of points over six metrics, but ends up last on *ease-of-use*. Angular has the most extensive API of all four frameworks and is without question the most powerful framework out of the box, but it takes some time to get used to its structure.

At this moment the biggest question on the internet regarding JavaScript frameworks is 'Angular or React?'. Looking at React's score over eight metrics, it lost the most points at *maturity* and *templating*. The low score on maturity only comes from the fact that Angular and Ember are more mature. The React team is working hard towards maturity level 4, which means that React is becoming a stable framework. React scores the lowest on templating. From a software engineering point of view is is often considered poor design to mix user interface and application logic. However, React forces the developer to do so. This is why React scored the lowest on the templating metric, but this is, besides a software engineering question, also a matter of personal taste. React scores well in terms of *ease-of-use*. Its API is small but well written. It does not offer much functionality out of the box, but there are a large number of extending libraries to find on the *React Community* GitHub repository. Its small and clear API makes it easy to grasp, combined with its pure JavaScript coding style.

Vue.js scores surprisingly good over all metrics. It lost a significant number of points on *maturity* and *testability* because it is the youngest framework without real support for testing. On all other metrics it scores surprisingly well. Vue did a good job taking the proven best parts from Angular and React. It has yet to be seen if Vue will compete with the larger frameworks a year or two from now. It is still developed by a 'one person core-team' and the framework is far from stable.

Although we were sceptical at the beginning, we think that the commonly asked question whether to use AngularJS or ReactJS is justifiable. Vue.js's immaturity is a good reason not to go for Vue.js. It's API is still undergoing changes which makes it unusable for large and actively developed. Ember.js is a solid framework that is worth considering. However, it does not have any major advantages over AngularJS. AngularJS and Ember.js are comparable in design and structure. Based on the outcome of our evaluation, especially performance, AngularJS is the better choice.

The two frameworks that are left are AngularJS and ReactJS. Which one to choose? For the development of the Vampires UI we choose to use AngularJS. Is AngularJS better? We think so, but this answer is up for debate.

AngularJS and ReactJS present two entirely different approaches to a framework to build a client-side application with. AngularJS is has a large API with an extensive set of possibilities. ReactJS is the opposite. ReactJS is minimalistic and is build around pure JavaScript. Because of this simplicity it is easy to learn and get started with. The *React Community* offers a large number of libraries to extend ReactJS's functionality. AngularJS has most of this functionality built-in, but for Angular too there are a large number of extending libraries available. The question between AngularJS and ReactJS is a question of what you want, rather than which is better. If you want to get started quickly and do not mind mixing up user interface with application logic ReactJS is the best option. Do you really want to keep your user interface and application logic separated and do not mind having to read many pages of documentation before really getting started, AngularJS is the choice.

Chapter 6

AngularJS

Before discussing and understanding the implementation of the Vampires front-end in AngularJS, we will need some basic knowledge of Angular.

6.1 Angular Structures

Angular applications are built around three main structures. Controllers control the data in the view it is linked to. Services can be injected in controllers to deliver a certain functionality. Services are commonly used to share functionality that is independent to a specific controller, so that it can be shared across multiple controllers. For DOM manipulation, Angular provides directives. Directives, like services, are separate units of functionality. A directive is defined to act on the occurrence of a certain HTML element, attribute, class or comment. Whenever the predefined condition appears in the DOM, Angular triggers the directive to manipulate the DOM.

6.1.1 Application

An Angular application is defined by adding the `ng-app` attribute to a HTML element that encloses all other Angular functionality on the page. Most often this attribute is added to the `body` tag. The application is initialised in JavaScript with a call to `angular.module('appname', [])`. See listings 1 and 2.

```
<html>
    <head>
    </head>
    <body ng-app="myApp">
    </body>
</html>
```

Listing 1: Initialising an Angular application in HTML

```
var app = angular.module('myApp', []);
```

Listing 2: Initialising an Angular application in JavaScript. The second parameter is an array of dependencies.

6.1.2 Controllers

Controllers in Angular are defined by adding the `ng-controller` attribute to an HTML element. This element needs to be encapsulated by a `ng-app` attribute. Controllers can be passed the Angular `$scope` object, which acts as the bridge between the DOM and the controller.

```
<body ng-app="myApp">
    <div ng-controller="myController">
            <p>What is your name?</p>
            <input type="text" ng-model="myNameInput" />
            <button ng-click="setName()">Confirm</button>
            <p>Hello {{ name }} (from controller)</p>
            <p>Hello {{ myNameInput }} (direct)</p>
    </div>
</body>
```

Listing 3: Basic example of a controller in Angular (HTML)

```
angular.module('myApp')
    .controller('myController', function($scope) {
            $scope.name = null; %$

            $scope.setName = function setName() {
                    $scope.name = $scope.myNameInput;
            }
    });
```

Listing 4: Basic example of a controller in Angular (JavaScript)

Listings 3 and 4 give a short example of how the `$scope` works as the bridge between the HTML file and the controller in JavaScript. In HTML the controller is defined with the `ng-controller` attribute, with an encapsulating element having `ng-app`.

The `input` element has an `ng-model` attribute, which links the value of the input field to the `myNameInput` property of the `$scope` object.
The `button` element calls the `$scope` function `setName()` when it is clicked, defined by the Angular `ng-click` attribute. The two `p` elements both contain the `$scope` variables again, enclosed by double brackets. This prints the current value of the variable at that position.

In the JavaScript in listing 4 we define a controller with a `$scope` variable `name` and function `setName()`.

What happens in this piece of code is that as soon as the user starts typing in the input field, the second `p` element is updated. The first `p` element does not change until the user clicks the button. The `$scope.setName()` function is called, updates the `$scope.name` variable based on the current value of the `input` field. The value of {{ `name` }} in HTML will change accordingly.

Note that in 4 the first call to `angular.module()` is missing the second argument that defines the dependencies. `angular.module()` is only called once with the dependencies list for initialisation. Subsequent calls in other places simply return the Angular instance, instead of initialising it.

6.1.3 Services

A service in Angular provides (shared) functionality to controllers. A controller might need to upload files to a server, fetch data from an external source or retrieve the location of the user of the application. This functionality is controller-independent and can be provided by an Angular service.

```
angular.module('myApp')
    .factory('LocationService', function() {
            var factory = {};

            factory.getLocation = function getLocation() {
                    navigator.geolocation.getCurrentPosition(function(p) {
                            [return p.coords.latitude, p.coords.longitude];
                    }
            }

            return factory;
    });
```

Listing 5: Very basic example of a service in Angular

```
angular.module('myApp')
    .controller('myController', function($scope, LocationService) {

            $scope.location = LocationService.getLocation();
    });
```

Listing 6: Example of a controller in Angular using a service.

A basic example of using a service in Angular is given in listings 5 and 6. In listing 5 a service is defined using the `factory` method. The service contains one function `getLocation()` that uses `navigation.geolocation.getCurrentPosition()` (a browser API call) to retrieve the GPS location of the user.

The service is passed as an argument to the controller in listing 6. Angular takes care of injecting the service to the controller, and the controller is now able to use a single call to `LocationService.getLocation()` to retrieve the location of the user. This example is especially useful when we have in mind that the call to `navigation.geolocation.getCurrentPosition()` shows a pop-up to the user every time the function is called. To prevent this we could save the user's location in the browser's Local Storage and fetch it from there. This logic could all be implemented in the LocationService, keeping the controller simple and abstract.

6.1.4 Directives

Directives in Angular are used to manipulate the DOM or take special DOM-related actions when an event occurs. Directives have their own scope, but variables and objects from the encapsulating controller/view-scope can be passed to the directive scope or linked one-on-one to directive scope variables.

```
angular.module('myApp')
    .directive('myDirective', function() {
            return {
                    restrict: 'E',
                            // Bind directive only to elements
                    replace: true,
                            // Replace element with template
                    template: '<div><p>New content</p></div>'

            };
    });
```

Listing 7: A basic example of a directive in Angular

```
<body ng-app="myApp">
    <div>
        <p>Some text</p>
        <my-directive></my-directive>
    </div>
</body>
```

Listing 8: HTML with custom directive before binding directive.

```
<body ng-app="myApp">
        <div>
        <p>Some text</p>
        <div><p>New content</p></div>
    </div>
</body>
```

Listing 9: Listing 8 after binding custom directive.

Listing 7 defines a very basic directive. The directive is defined to restrict binding to HTML elements with the name `my-directive` only (the camel-case `myDirective` becomes `my-directive`). Other options here are C (classes), A (attributes) or M (comments). A combination of these letters is also possible; `restrict: AE` binds to attributes and elements. The `replace` and `template` properties tell Angular to replace the matched element with the given template. See listings 8 and 9.

Directives are powerful and can do much more than just replacing elements. Say we need to have an image slider on multiple places on a website. We could create a directive `imageSlider` and call it like this:

```
<image-slider images="imageList"></image-slider>
```

The element has an attribute `images` which we can access in the directive's scope. An array of images is generated by the controller and the directive replaces this single element with an image slider template loaded from file. This new `image-slider` tag can be used anywhere we want to have an image slider.

Chapter 7

Implementation of the Vampires user interface

This chapter describes our implementation of the Vampires user interface. First, we will show and explain the application from a user's point of view. We will walk through the process from defining a task to eventually executing it on the best possible configuration of resources.

After this high level overview, we will walk through the implementation of the different parts of the application. This includes the global application structure, the chosen directory structure, the separate views and the supporting Angular services and directives.

This chapter concludes with a description of the tools used for testing and building the application for use in production.

The source code of the Vampires user interface is available at [38].

7.1 High level overview

In this section we will walk through the screens the user of the Vampires UI encounters in the process of running Vampires. The screen shots in this section are taken during a real execution of a GraphicsMagick benchmark script [39] on the DAS-5 [40] research cluster.

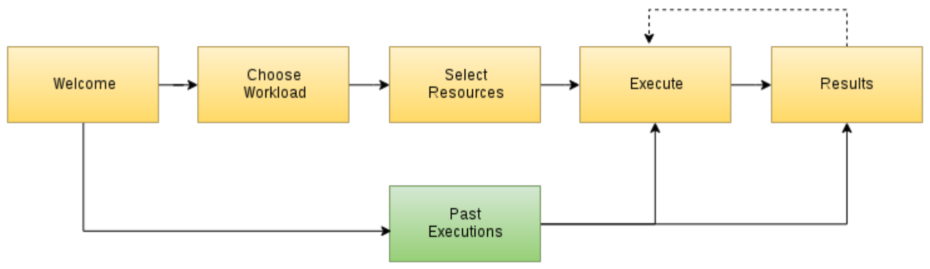


Figure 7.1: The user flow through the application.

Figure 7.1 shows the user flow through the application. The yellow path assists the user in defining and executing a new task. The dotted path from the *results* to the *execute* screen is only available after a sampling execution, after the user has initiated a full execution from the *results* screen. We will explain the different views and paths in more detail in the following sections.

7.1.1 Welcome screen

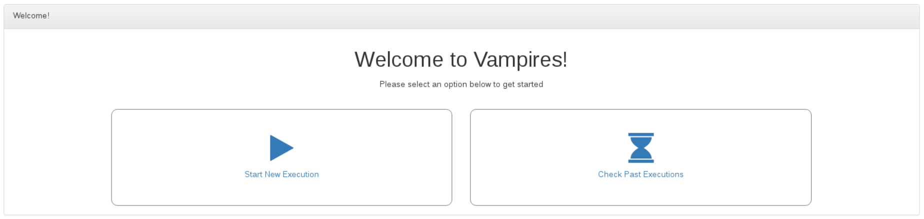
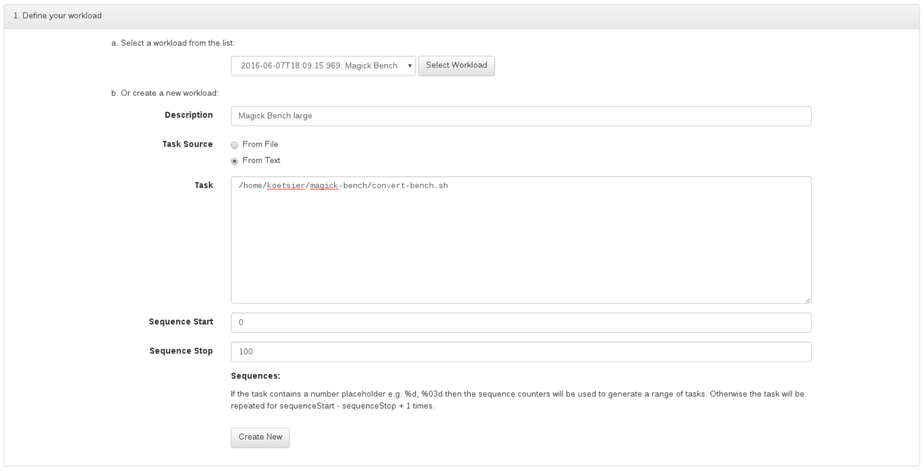


Figure 7.2: The Vampires welcome screen

The first screen the user sees is the login screen, followed by the welcome screen. The welcome screen provides the user with two options. The user can start a new run, which takes the user through the process of defining a new configuration (7.1.2), or the user can click through to a list of past executions (7.1.6).

7.1.2 Define workload


1. Define your workload

a. Select a workload from the list:

2016-06-07T18:09:15.969: Magick Bench

b. Or create a new workload:

Description	Magick Bench large
Task Source	From File [✓ From Text]
Task	/home/koetsier/magick-bench/convert-bench.sh
Sequence Start	0
Sequence Stop	100

Sequences:

If the task contains a number placeholder e.g. %d, %03d then the sequence counters will be used to generate a range of tasks. Otherwise the task will be repeated for sequenceStart - sequenceStop + 1 times.

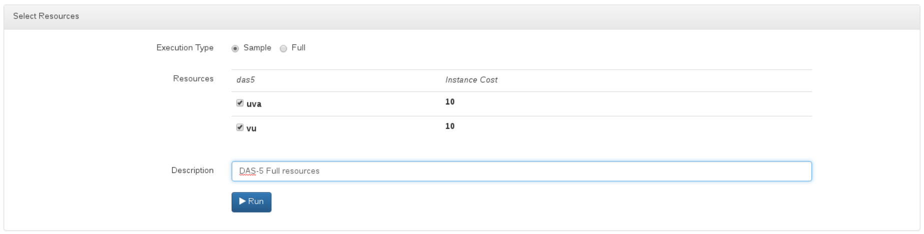
Figure 7.3: The 'define workload' screen

This is the first screen the user encounters when she chooses to start a new execution. The user defines a workload on this screen. A workload is the task the user wants to execute. The drop down on top lists the previously created workloads. One of the previously created workloads can be selected to start an execution with the same workload, but a different configuration of resources. A new workload can be created on the bottom part of the page. This workload consists of a sequence of commands and can either be uploaded from file or entered in the text input field. The `Sequence Start` and `Sequence Stop` are fields used by the Vampires back-end, as shown in the following command:.

```
convert image%03d.png -resize 800x600 output%03d.png
```

The back-end will replace the placeholder `%03d` with the sequence counter. Using this example, we could enter the values `Sequence Start = 1` and `Sequence Stop = 100`. The result would be a conversion of images with file names ranging from `image001.png` to `image100.png`. After selecting or creating a workload, the user will be redirected to select the resources to use for this execution (7.1.3).

7.1.3 Select resources


Select Resources

Execution Type: [✓ Sample] Full

Resources:

das5	Instance Cost
✓ uva	10
✓ vu	10

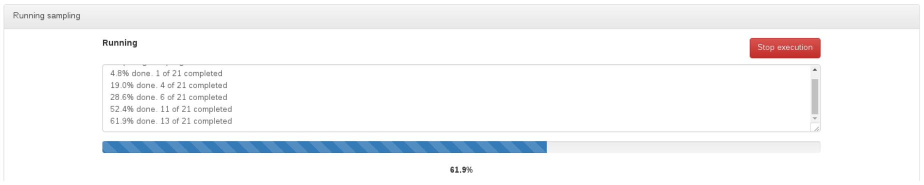
Description: DAS-5 Full resources

Figure 7.4: The resource selection screen

On this screen, the user selects the resources that will be used during execution. Resources are listed per provider, like *DAS-5* or *Amazon EC2*. These providers each have their available instance types. In case of Amazon EC2, this could be a list of Amazon EC2 instance types like *t2.micro* or *m3.medium*. The default execution type here is a sample run, but a user could choose to skip the sampling phase and run a full execution of the task. When a user chooses to go for the full execution at this stage, input fields show up to select the number of instances to use of each resource.

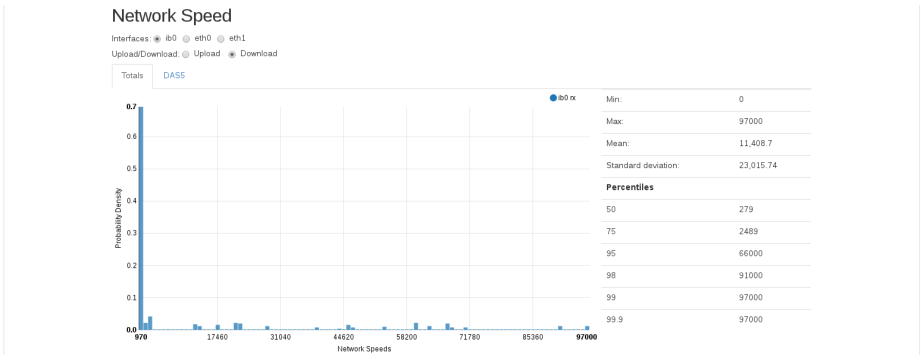
After selecting the resources to use for the execution, the user will be redirected to the *Execution* screen (7.1.4).

7.1.4 Execution



```
4.8% done. 1 of 21 completed
19.0% done. 4 of 21 completed
28.6% done. 6 of 21 completed
52.4% done. 11 of 21 completed
61.9% done. 13 of 21 completed
```

Figure 7.5: A running execution in Vampires.



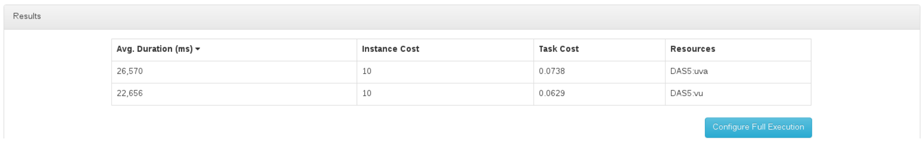
Min:	0
Max:	97000
Mean:	11,408.7
Standard deviation:	23,015.74
Percentiles	
50	279
75	2489
95	66000
98	91000
99	97000
99.9	97000

Figure 7.6: The user is presented with statistics during the execution.

This page shows the progress of the current execution. The text area provides the user with detailed information about the current state of the execution, with visual support of the progress bar. The histograms below the progress information present the user with the CPU and network usage during the execution.

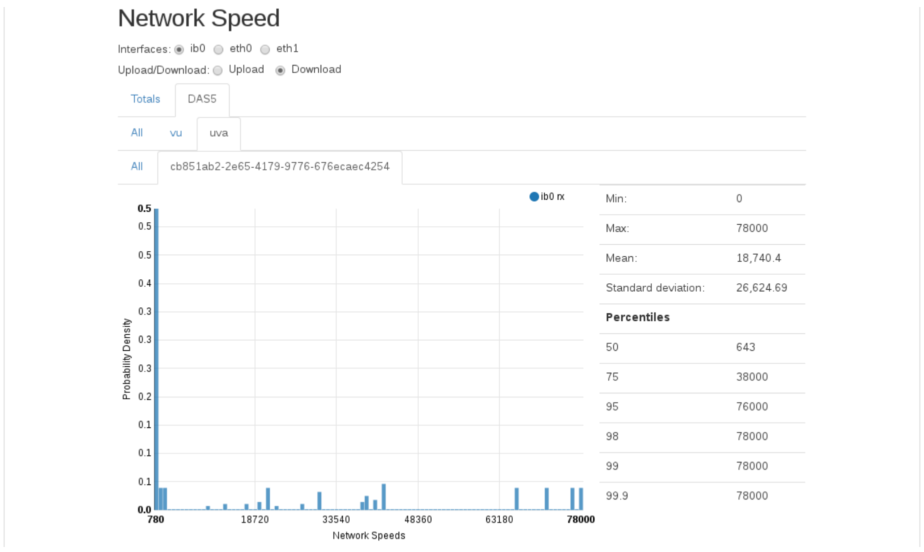
When the execution has finished, the possibility to stop the execution is replaced by a button that redirects the user to the `Results` 7.1.5 page.

7.1.5 Results


Results

Avg. Duration (ms) ▾	Instance Cost	Task Cost	Resources
26,570	10	0.0738	DAS5:uva
22,656	10	0.0629	DAS5:vu

Figure 7.7: The sampling results regarding average duration and cost per instance type.


Network Speed

Interfaces: [✓ ib0] eth0 eth1

Upload/Download: Upload [✓ Download]

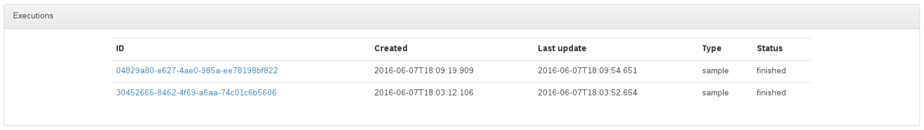
Min:	0
Max:	78000
Mean:	18,740.4
Standard deviation:	26,624.69
Percentiles	
50	643
75	38000
95	76000
98	78000
99	78000
99.9	78000

Figure 7.8: The tabs make it possible to easily see more detail about specific providers, instance types or clients.

After an execution has finished, the user is redirected to the `Results` screen. Detailed execution statistics are presented to the user here. The tabs above the graphs and tables allow the user to view the statistics from different levels, from the total overview of all clients, to the totals per provider, per instance type or client.

When these results are the results of a sample run the table on top lists the used instance types including cost, average duration and estimated cost of the last execution. From here the user can select the number of instances of each resource to use to start a full execution. On starting the full execution the user is redirected to the *execution* screen again.

7.1.6 Past executions


Executions

ID	Created	Last update	Type	Status
04829a80-e627-4ae0-985a-ee78198bf822	2016-06-07T18:09:19.909	2016-06-07T18:09:54.651	sample	finished
30452665-8462-4f69-a6aa-74c01c6b5606	2016-06-07T18:03:12.106	2016-06-07T18:03:52.654	sample	finished

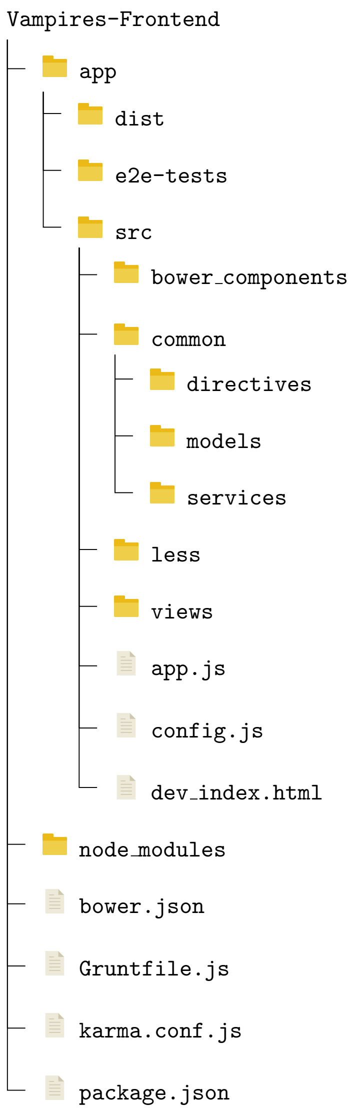
Figure 7.9: The user can choose to view previous executions.

This screen lists all previous executions. A click on the execution ID redirects the user to either the execution progress 7.1.4 screen or the results 7.1.5 screen, depending on the execution status (running or finished).

7.2 Application structure

7.2.1 Directories

In this section, we will explain the directory structure of the Vampires UI application. We start with the global structure and continue with a more detailed overview of the contents of the directories.



```
Vampires-Frontend
├─ app
│  ├─ dist
│  ├─ e2e-tests
│  └─ src
│     ├─ bower_components
│     ├─ common
│     │  ├─ directives
│     │  ├─ models
│     │  └─ services
│     ├─ less
│     ├─ views
│     ├─ app.js
│     ├─ config.js
│     └─ dev_index.html
├─ node_modules
├─ bower.json
├─ Gruntfile.js
├─ karma.conf.js
└─ package.json
```

Figure 7.10: The directory structure of the application

AngularJS is not a classic MVC, MVVM or MV* framework. As Igor Minar [41] stated in 2012:

> *(...) Having said, I'd rather see developers build kick-ass apps that are well-designed and follow separation of concerns, than see them waste time arguing about MV* nonsense. And for this reason, I hereby declare AngularJS to be MVW framework - Model-View-Whatever. Where Whatever stands for "whatever works for you".*

Angular does not come with a predefined directory or application structure. It is up to the developer to structure the application as it fits. The structure of the Vampires UI application is based on various discussions on the internet [42, 43, 44], with adjustments as the development continued.

Figure 7.10 shows the directory structure of the Vampires front-end.

In the root of the structure, we find the `app` and `node_modules` directories, as well as four configuration files. The `node_modules` directory contains modules from the Node.js NPM repository. These modules are used for development, including the test and build libraries and an HTTP server to run the application locally during development. The four configuration files are also used for the build process. The build process is explained in section 7.7.

The `app` directory contains the application itself. In the `src` directory we find the actual source code of the application. During the build process the production-ready files are copied to the `dist` directory, which can then be used to serve the application. In the `src` directory we find `bower_components` containing client-side JavaScript libraries including An-

gularJS itself. The `common` directory holds the shared directives (section 7.5), models (section 7.6) and services (section 7.4). The `less` directory holds the LESS files that will be compiled to CSS files during the build process. The `views` directory contains the views (screens) of the application.
`app.js` and `config.js` contain the initialisation and configuration of AngularJS and `dev_index.html`. `dev_index.html` is further explained in section 7.7.

7.2.2 Components

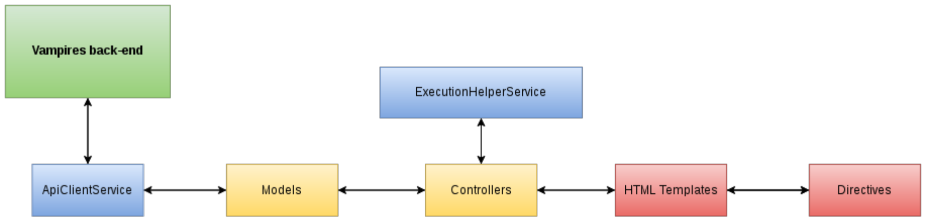


Figure 7.11: The global application structure.

Figure 7.11 shows the components of the application. We will present a short explanation of these components in this section. A more detailed explanation of the separate components will be given in the sections that follow.

The core of our application is the *ExecutionHelperService*. This service keeps the current state of the application and controls the creation of new executions. The controllers are the only components that communicate with this service. Each view has its own controller that manages the data that is displayed in the HTML templates. The data is fetched solely through the models, which in turn communicate with the *ApiClientService*. This separation of *ApiClientService*, models and controllers creates abstraction levels throughout the application that keep the code simple and easy to understand.

7.3 Views

The `views` directory contains directories for every page in the application. These directories contain the HTML template, the JavaScript file containing the controller and possibly page-specific directives and a JavaScript file containing the test scripts. We chose for this structure to keep all files specific to a page in one designated directory because this gives a good overview of the files that make up a page. This is in contrast with the classical MVC [1] design, where the files are often spread over at least three different locations. The (database) models go in the models directory, the controllers can be found in the controllers directory and the HTML templates reside in the views directory. Working on an application with a large number of views forces the developer to browse through

[1]Model-View-Controller

a large directory structure. It is completely possible to structure an AngularJS application in an MVC-like way, but it is not as convenient to work with.
The controller files of the Vampires front-end only contain functionality that is directly linked with the view. All other functionality like processing and fetching data is placed within services and directives to keep the application modular and testable and keep the controllers simple.

7.4 Services

In the `services` directory are four services defined. The functionality these services provide can be shared between different controllers. Services also provide a level of abstraction to the controller.

7.4.1 ApiClientService

The `ApiClientService` has as its sole purpose to communicate with the Vampires back-end [45]. All communication with the back-end is done through this service. The file consists only of functions that make HTTP requests and provide an abstraction to the other parts of the application.

7.4.2 AuthService

The `AuthService` provides an abstraction for user authentication. It manages the login and authentication and provides the `ApiClientService` with the correct HTTP headers to send with the requests to the back-end. There is currently only basic authentication supported by the back-end. In the future this will probably changed. Having the authentication in one separate service makes it easy to implement a different authentication method in the future.

7.4.3 ExecutionDataExtractService

The JSON response from the execution status endpoint of the Vampires back-end is large and needs processing before it can be used to display graphs or tables in the views. This information is used by multiple views and directives and is not specific to a controllers' functionality. Therefore this functionality is placed in the `ExecutionDataExtractService`. Extracting this functionality from the controller keeps the controller simple.

7.4.4 ExecutionHelperService

This service forms the core of the process of defining a task, running a sample, viewing the results and executing the full task.

Controllers in AngularJS have no way to share data directly. This is where this service assists. After posting a new workload, configuration or execution to the back-end we get an identifier back. This identifier must somehow be saved for later use. The *ExecutionHelperService* saves the current state of the application. It also checks if the user has taken all the necessary steps preceding the current step. A user could enter the application on the page where she needs to select the resources to sample without having defined a workload. The *ExecutionHelperService* detects these cases and redirects the user to the correct step in the process.

7.5 Directives

The `directives` directory contains a number of directives. Using directives for specific parts of the application has two advantages. Directives create an abstraction to both the controllers and HTML templates. Logic and HTML templates that create a specific piece of application functionality together are extracted into directives. This results in cleaner code in both the controller and the HTML template file. The second advantage of abstracting functionality with controllers is reusability. Directives can be easily reused in different part of the application, keeping the DRY-principle in mind.

7.5.1 Graph directives

All graphs used on the Vampires front-end are modularised into directives. Using this structure it becomes easy to use the same graphs on both the execution and results pages. The histogram graph directives contain a table with histogram-specific statistics such as the mean, standard deviation and percentiles. This table is defined in its own directive, so that every histogram directive can import the statistics table with one simple HTML tag with a `stats` attribute pointing to the statistics object.

7.5.2 Statistics tabs

The tables and graphs on the execution and results pages all have headers with which the user can select the level of detail when viewing the statistics. This is a relatively large piece of code that every graph or table implements. The header tabs are therefore created in their own directive. The calling directive only has to insert the `statistics-tabs` HTML element along with references to the root element of the statistics tree and a *select function* that will be called to update the statistics view whenever the user selects a new level of detail.

7.6 Models

Models are not common AngularJS entities. In server-side applications, models are used to create an abstraction towards the database, often combined with an ORM [2]. AngularJS is a client-side application with no direct communication

[2]Object-relational mapping. A tool that maps objects in an object-oriented programming language to database entities

to a database engine and thus no need for models in a way one finds in MVC-frameworks.

However, to provide an abstraction level to the application, we have constructed models that represent objects returned from the Vampires back-end. These are the configurations, workloads, executions, resources and providers. The models have methods associated to fetch objects from the back-end, supported by the *ApiClientService*. The controllers and services of the application can then use these models as named JavaScript objects.

7.7 Tools

This section describes the used development tools. These are the tools used to build and test the application.

7.7.1 NPM

NPM is the Node Package Manager that is part of Node.js. Although we do not use Node.js itself, the package manager provides us with many tools we need. When we first clone the repository of the Vampires front-end, the `node_modules` directory does not exist yet. Running `npm install` in the terminal from the root directory of the project starts the installation of all necessary dependencies as listed in the file `package.json`. Among these dependencies are *Bower* and *Grunt*, which are explained in the following sections.

7.7.2 Bower

Bower is a package manager for web libraries, frameworks and utilities. Bower itself is an NPM module and is installed as soon as `npm install` is executed. Bower uses a similar configuration file with dependencies as NPM. Running `bower install` installs the dependencies from `bower.json` into `app/src/bower_components`. Running `bower install` for our application is not needed, as it gets triggered by `npm install`.

Among the packages that are installed using Bower are the AngularJS packages, the JavaScript graph library NVD3 and the libraries containing Twitter Bootstrap and Font Awesome.

7.7.3 Grunt

Grunt is a JavaScript task runner that automates the build process. There are two tasks defined in `Gruntfile.js`. The first task is the `build` task, executed by running `grunt build` from the command line. This task handles the build of the application into the `dist` directory.

Running the *build* task will trigger JSHint [46], a JavaScript quality tool that tries to detect errors in the code. Because JavaScript is not compiled, a tool such as JSHint is the next best thing to prevent runtime errors. When the

JSHint run has finished successfully, Grunt runs the unit and end-to-end tests. After the source code has passed all tests, the actual building of the application starts.

During the build process, all application files are copied from the `src` directory to the `dist` directory, but a number of files need to be processed first. This is handled by Grunt as well.

Grunt runs a LESS compiler that compiles our LESS files to normal CSS. LESS files are used during development because it offers functionality that standard CSS does not have. With LESS we can nest statements and use variables. The LESS compiler translates this all to valid CSS.

Next, Grunt takes all JavaScript source files of our application, concatenates them into one large file and minifies this file. A client-side JavaScript application consists of a large number of separate JavaScript files. In a production environment, all these files would have to be downloaded separately from the web server. This results in a large number of HTTP requests, thereby increasing the load time of the web page. We have set up Grunt in such a way that all JavaScript ends up in one file. After this concatenation step the file is minified; white-space is removed from the source code and identifiers are replaced with short names. This results in a smaller file, decreasing the web page's load time further. Grunt performs the same minify task over all CSS files as well. The end result is that there is only one JavaScript file and one CSS file to be imported in the HTML that makes up the application, instead of 40.

The second Grunt task is the `run` task, used during development. It runs JSHint, starts a local web server and launches a *watcher* process. This *watcher* process starts the Grunt LESS compiler whenever it detects a change in our LESS file so that updates are automatically carried out to the CSS file during development.

Both Grunt tasks also include a `htmlbuild` sub task. This task takes the `dev_index.html` file and inserts references to the correct CSS and JavaScript files of our application. These references are different for the applications in the `dist` and `src` directories. The files in the `dist` directory are concatenated and minified into two files, where as the application in the `src` directory uses the original files as imports to prevent having to build after every update.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

We have answered two question in this thesis. The first question, '*How to construct a user-friendly user interface for Vampires?*' has been answered in chapters 3 and 7. We have developed an interface that guides the user through the Vampires process step by step. The user is presented with relevant feedback through the process using progress bars, tables and graphs. The user is only asked for the minimal information necessary to start an execution on Vampires. During the execution the user gets real-time feedback. On the results page after an execution has finished, the user is presented with information on different levels of detail. The results page includes estimates of the cost of the executed task.

The second question, '*Which client-side JavaScript framework is the most suitable for the development of a user interface for Vampires?*' is answered in the chapters 4 and 5. After a survey regarding the most popular JavaScript frameworks today, we have evaluated the four most popular JavaScript frameworks on the basis of the ISO 25010:2011 product quality model characteristics. The question commonly asked on the internet regarding client-side JavaScript frameworks is whether to use ReactJS or AngularJS. We were sceptical about the validity of this question, which implies that there are only two good JavaScript frameworks available. We have therefore not limited our research to only these two frameworks, but have included Ember.js and Vue.js in our research. We can now conclude that the question of using ReactJS or AngularJS is a justifiable question, although we do not want to imply that ReactJS and AngularJS are the only good frameworks available. Ember.js, Vue.js and AngularJS share a lot of common ground, but we think that AngularJS is the better framework of the three. We see no reason to use Ember.js or Vue.js instead of AngularJS. Comparing AngularJS with ReactJS is difficult, as they are both entirely different frameworks.

AngularJS came out best from our evaluation, but the main difference in points with ReactJS was caused by the way the frameworks handle HTML templates. We like the separation of the view templates and application logic, but this is, besides a software engineering principle, also a matter of personal taste.

8.2 Future Work

The Vampires user interface currently uses basic authentication with the back-end. The back-end handles the authentication with the cloud resources. This means that the Vampires back-end always runs on one user account from the cloud services provider. It is impossible now to run one user interface and with multiple users that will all be charged for their own cloud usage. This will involve some significant security measures.

The Vampires back-end currently only returns cost statistics and estimations per resource type. It does not yet calculate configurations of combinations of different resources. Implementing these calculations on the back-end will provide the user with more useful information in the application.

Bibliography

[1] C. Dumitru, A.-M. Oprescu, M. Živković, R. van der Mei, P. Grosso, and C. de Laat, "A queueing theory approach to pareto optimal bags-of-tasks scheduling on clouds," 2014.

[2] "Vampires Framework." https://bitbucket.org/cdumitru/vampires-akka.

[3] A.-M. Oprescu, T. Kielmann, and H. Leahu, "Budget estimation and control for bag-of-tasks scheduling in clouds," 2011.

[4] "Amazon EC2." https://aws.amazon.com/ec2.

[5] "Amazon EC2 Instance Types." https://aws.amazon.com/ec2/instance-types/.

[6] "Netscape and Sun Announce Javascript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet." https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html, 1995.

[7] C. Severance, "JavaScript: Designing a Language in 10 Days," 2012.

[8] "Ajax: A New Approach to Web Applications." http://adaptivepath.org/ideas/ajax-new-approach-web-applications/, 2005.

[9] "It's official: ECMAScript 6 is approved." http://www.infoworld.com/article/2937716/javascript/its-official-ecmascript-6-is-approved.html.

[10] "jQuery." https://jquery.com/.

[11] "Top 10 Most Used JavaScript Frameworks." https://blog.codeanywhere.com/top-10-most-used-javascript-frameworks/.

[12] "Most Popular JavaScript Frameworks 2015." http://www.improgrammer.net/most-popular-javascript-frameworks-2015/.

[13] "JavaScript Frameworks in 2016." http://www.clock.co.uk/blog/javascript-frameworks-in-2016.

[14] "Top 23 Best Free JavaScript Frameworks for Web Developers 2016." https://colorlib.com/wp/javascript-frameworks/.

[15] "Top JavaScript Frameworks, Libraries and Tools and When to Use Them." https://www.sitepoint.com/top-javascript-frameworks-libraries-tools-use/.

[16] "TodoMVC." http://todomvc.com/.

[17] "GitHub." https://github.com.

[18] "StackOverflow." http://www.stackoverflow.com.

[19] "JS Comparison: Angular vs. React vs. Vue." http://react-etc.net/entry/comparison-js-angular-react-vue.

[20] "AngularJS." https://angularjs.org.

[21] "Angular 2." https://angular.io/.

[22] "Angular 2 Milestones." https://github.com/angular/angular/milestones.

[23] "ReactJS." http://facebook.github.io/react/.

[24] "Interview with Pete Hunt (React: Making faster, smoother UIs for data-driven Web apps)." http://www.infoworld.com/article/2608181/javascript/react--making-faster--smoother-uis-for-data-driven-web-apps.html.

[25] "Ember.js." http://emberjs.com/.

[26] "Vue.js." https://vuejs.org/.

[27] "Iso 25010:2011 - systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," 2011.

[28] M. L. B. W. Boehm, J. R. Brown, "Quantitative evaluation of software quality," 1976.

[29] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability Maturity Model for Software, Version 1.1," 1993.

[30] "AngularJS GitHub." https://github.com/angular/angular.js.

[31] "ReactJS GitHub." https://github.com/facebook/react.

[32] "Why you should not use AngularJS." https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99.

[33] “AngularJS is amazing... and hard as hell.” https://coderwall.com/p/3qclqg/angularjs-is-amazing-and-hard-as-hell.

[34] “The reason Angular JS will fail.” http://okmaya.com/2014/03/12/the-reason-angular-js-will-fail/.

[35] “Advantages and Disadvantages of AngularJS.” http://www.w3technology.info/2015/11/advantage-and-disadvantage-of-angularjs.html.

[36] “JavaScript Framework Benchmark.” https://github.com/krausest/js-framework-benchmark.

[37] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[38] “Vampires UI on GitHub.” https://github.com/JKoetsier/vampires-ui.

[39] “GraphicsMagick Bench.” https://lo-ol.fr/gitlist/shell/tree/64770e18c065ed9e675c67121541f84fecc9ce1e/scripts/image%20manipulations/GraphicsMagick/benchmark/magick-bench/.

[40] “DAS-5.” http://www.cs.vu.nl/das5.

[41] “Igor Minar about AngularJS’s MVW.” https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV.

[42] “The Top 10 Mistakes AngularJS Developers Make.” https://www.airpair.com/angularjs/posts/top-10-mistakes-angularjs-developers-make.

[43] “AngularJS - Recommended Directory Structure for Angular Apps.” https://dzone.com/articles/angularjs-%E2%80%93-recommended.

[44] “AngularJS Project Structure.” http://www.davecooper.org/angular-project-structure.

[45] “Vampires API.” http://docs.vampires.apiary.io/.

[46] “Jshint.” http://jshint.com/.