An empirical study of the transformation of a Node.js based web application, Entize Planning, into a cloud service

En empirisk studie av transformationen av en Node.js baserad webbapplikation, Entize Planning, till en molntjänst

Rikard Lagerman
An empirical study of the transformation of a Node.js based web application, Entize Planning, into a cloud service

Rikard Lagerman
Department of Mathematics and Computer Science, Karlstad University

Abstract

The Node.js based web application Entize Planning needs to evolve and become distributable, and the logical transformation is into a SaaS cloud service that can compete in a pressured economic climate concerning resources. The aim is to provide an insight into the challenges in mitigating a web service to the cloud. By utilising the edge concerning low-cost and rapid development associated with Open Source use, a NoSQL supported and Node.js with related technologies, cloud service prototype development process and environment is proposed. Keeping guidelines such as: security, maintenance, complexity, scalability and mobility; emphasised, in order to carefully utilise existing architecture will result in an flexible solution that preserves aforementioned attributes, making the project as a whole a success towards its endusers. Key remarks that surfaced was the distinct separation between Entize Planning - the product - and the delivery mechanism - the cloud service plattform - and also the explicit need to investigate any utilisation of preexisting cloud architecture pertaining limitations in their delivery, before adoption.

Keywords: cloud service, web service, entize planning, web application transformation, software development
This thesis is submitted in partial fulfillment of the requirements for the Master’s degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

______________________________
Rikard Lagerman

Approved,

______________________________
Supervisor: Karl-Johan Grinnemo

______________________________
Examiner: Thijs Jan Holleboom
Acknowledgements

For the provided support.

Karl-Johan Grinnemo
For valuable insights and suggestions regarding content and its disposition. Thanks.

Carl-Johan Blomqvist
For providing a corporate perspective. Thanks.

Karlstad University, June 5, 2014

Rikard Lagerman
1 Introduction

In a competitive market, being able to provide consumers with software services without the need for the corresponding hardware, is indeed an attractive prospect for small businesses. During the last decade, we have seen a growing market of services in the cloud, called cloud services or cloud computing. The cloud service architecture addresses issues that changes the playground for software production. What is needed to prepare and transform your product to a cloud service might not be altogether clear. This thesis will document such a transformation, with the aim to provide an insight into how such a transformation might be done in order to make possible the next evolutionary step.

Secure storage and processing of private data - while using hired hardware - online - and being able to change allocated resources instantly - and at the same time avoid the employment of the necessary experts for hosting, makes the market lucrative for software development companies. Moreover, it makes a documentation that exemplifies and provides guidelines for how to migrate an existing Web service to the cloud of real interest to these companies. A company that shares this interest, has a product that they wish could be delivered as a cloud service and has to that end provided the corporate perspective and support for this prototype and its development.

Its expected that readers of this thesis will find facts and ideas that can support their own development of a cloud service platform. The thesis do not aspire to cover all aspects or present all solutions that is associated with a cloud service platform, but it do include all the major discoveries and important notes that has been found during the construction of this platform. The initial criteria was: Node.js as framework, low maintenance, low maintenance cost, low complexity, sufficient security and scalable functionality. Software technologies and functionalities such as Node.js, Express, MongoDB, MongoWatch, Sharding, Replica set, Jade, Stylus, CSS, Overlay, Javascript, Web sockets, Sessions, SSL, SSH, Shell script, REST, JSON, DigitalOcean API and more, will be briefly explained. The thesis content will hopefully provide insights into problems that might arise, clues to overcome such problems and similarly inspire to other solutions, more suited for the readers intended service. Since there is probably no such thing as the old proverb, "One size fits all", so do the methods and the familiarity with different software often enable many different solutions with their respective drawbacks and strengths, which essentially means that the solution presented in this thesis, is but one of many, possible solutions.

Chapter 2 - Background, explains the current state of Entize planning, the product and related technologies. This chapter also presents concepts such as what a cloud service is, the different types of services and its stack. Economic aspects pertaining cloud services are discussed briefly. Other aspects such as complexity, maintenance, software dependencies, monitoring and recovery are introduced. An outline of the scope for the development is formed and also a brief review of expectations on the prototype and associated goals are summarised.

Chapter 3 - Design and implementation, in which the cloud service type or level is selected along with a presentation of the chosen cloud service, since the prototype will not implement
all the levels of the cloud service stack. A comparison of a couple of cloud services is done to introduce some of the requirements and prerequisites. An initial overview of the cloud service’s platform, the prototype’s initial stage and its intended structure, is conveyed. The interaction with the chosen cloud service is explained in more detail. Then follows information needed to understand the structure of a Node.js project and a general explanation of web application development within that environment. Highlights and powerful features are mentioned and snippets of code is presented to clarify how basic functionalities are achieved for Node.js applications. After comes a more detailed walkthrough of chained events in an event driven application, which in the same time serves as a demonstration of one simple feature of the platform. The walkthrough will also illuminate key constructs and patterns that is common in Javascript applications. Following the walkthrough is an explanation of persistent storage, which will cover the use of a MongoDB database and parts of its setup. The section about persistent storage will also share how some of the platform’s problems will be solved. The preparation of the distributed entity, the product, security related problems and deployment will be reviewed. Before delivery of the product, an order must be placed, the order service is briefly explained and following ordering comes a more comprehensive section presenting the more complete view of the prototype. The overview shows how the platform is structured and how maintenance of services is enabled. How the prototype’s parts are connected will also be revealed. The support for scalability is solved and what the production ready prototype will need is introduced. How the major features regarding security are obtained including what support the chosen environment provides.

Chapter 4 - Results and evaluation, presents the results of some of the major choices made during the development of the prototype. The first section explains the effects of separating the prototype into parts and also the current state of the platform. The following section addresses the environment and the result of taking the environment into consideration during the development. Following the discussion of environment, performance metrics is provided and an evaluation of the product delivery mechanism. The overall cost for the project is revealed and an estimate of how much development that remains before the cloud service prototype can be taken into production is done. Whether the goals for the prototype has been met is discussed in more detail and also what is missing before the prototype goes into production. Lastly, the functionality provided by the service’s interfaces is discussed and also which functionality that is still in the planning stage.

Chapter 5 - Conclusion, summarises the project’s relevance, its execution, possible improvements and lessons learned.

The thesis will not cover all the aspects of cloud service hosting and management. Mostly since the overall time set for this project is too short and partly since there are many possible solutions. The monitoring aspect of the product is not really discussed in detail and the solution is still somewhat vague: Should the product signal its own status or should the cloud service platform acquire monitoring data intrusively? Even though a non-intrusive solution is the likely choice, this issue has not been completely resolved yet and is therefore not fully documented.
The handling of payment and accounting is also areas that lacks detailed documentation. Apart from these functionalities, many aspects are well represented.

The cloud service’s platform has the ability to setup new virtual servers, order, start and deploy a chosen product with resources that can be scaled vertically. To ensure a future where the platform can grow, the platform has been split into a collection of services that can be scaled both vertically and horizontally. Since the services are isolated the services can be started, stopped, updated, scaled or removed independently, without affecting any other service in the platform.

All the services are registered under the same domain and will before production use a wildcard certificate to enable the use of a SSL. The required choice to use the framework Node.js and Javascript as programming language has resulted in functionality built modularly, generically and has as a result afforded great mobility and therefore easy migration or removal. The platform is more general than was initially expected and is the result of placing the responsibility for product’s own functionality on the product, rather than on the platform and which has made the support of products, built in other frameworks, possible as well. The produced platform’s distribution mechanism can be used to deploy the platform’s own parts and further serves as the maintenance tool for the platform itself.

Although the platform uses another cloud service level for the hosting of virtual machines, the platform still entails much work and involves developers capable of both harnessing the web environment and also the resources of different operating systems along with database management. The hosting service that was chosen for the prototype offers good prices for their virtual servers and differs quite substantially from other vendors. This fact, in itself, should warrant an investigation of relied upon services, prior to the start of the development of a cloud service.

The Node.js framework was a new experience, only Javascript (and some shell scripting) did enforce the need to know how to use callbacks and event driven applications. The most notable difficulty with the Javascript structure might be the scope of functions and consequently when an objects value is accessible and when it is not.

The cloud service still needs to be expanded to support cluster configurations for products. Since the initial goal was to launch a product into the cloud with its resources located locally on the same virtual machine, so is this support an extension of that goal. The platform’s code base does lack some documentation, but is structured in the same way among the services, and further, time has been spent on naming functions and objects to reflect their use. Another limitation that the platform still has is that it lacks the ability to setup multi virtual server configurations for the product, but this a prioritised functionality. Since the platform’s services are built without blocking the program execution for external tasks, that takes time, so can with just a slight increase in time for deployment, 20 deployment be performed concurrently, a maximum has not been gauged.
The platform will be used firstly by the Entize Planning product crew to distribute and manage their product and will therefore at some point be developed further by their crew, to that end so has the overall complexity been kept low to avoid confusion and promote readability. Its estimated that one developer could take the product from current state to production status in one month.
2 Background

The Background chapter will explain the reason for this thesis, and clarify some general concepts. It will also address the current status of the application used in relation to the cloud service prototype that will be constructed. Some guidelines and expected problems will be identified. There will also be parts explaining what the endresult, the prototype, is supposed to be capable of doing.

2.1 The next evolutionary step for Entize Planning

Let us review the software that will be transformed. The Entize project’s product - Entize Planning [1] which will be the base for this thesis is an application that uses the browser as a user interface. The platform is the well-known Javascript platform, Node.js [2]. Node.js has numerous frameworks built upon its provided core libraries. One of those frameworks is called Derby [3]. Derby is a Model-View-Controller (MVC) framework. Using a book as an abstract example of what an MVC is then: the book is the model; that is presented by a webpage which is the view; which is changed by the turning action - done by the controller - to the next page. Software modules is used to provide an abstraction-layer against different database solutions. Racer [4], also a Javascript module is used by Derby for the handling of data synchronisation and propagation between database, model-objects and view. As many other web-frameworks Derby is written with the Client-Server model but with a slight difference. Code used in the server can be reused in the client. This reusable ability in Derby is restricted by the developer but has the potential to significantly reduce the Lines of Code (LOC) needed for an application.

Until now, the Swedish company B&B Web [5] that hosts the project Entize Planning has had one major consumer, and is now looking to expand the product’s usefulness by making the product, Entize Planning, more generic and available to a broader clientele. The company is interested in a functional cloud service that hopefully will eliminate the need for individual customisation from the development team towards its consumers. Entize Planning is a service that simplifies the visualisation during planning for the processing industry. Their current consumer is a company that specialises in the packaging of food.

The product maps the production as a series of stations or machines that can process a planned amount of material. Each of these stations or machines corresponds to a template that has user-defined rated values. An example could be for instance, speed. The consumer of this service can determine the number of machines that their production flow contains. A drag-and-drop supported interface now makes it easy to drop (attach) an amount of material to a machine. Depending on the machines metrics, a timeline will show an estimate for processing completion of the material attached. For a conceptual view, see Figure 1 which is Entize Plannings own official representation [6].
To be able to import or process orders that need to be planned in Entize Planning, there exists a layer that makes requests to a Microsoft SQL server (MSSQL). The handling of orders is managed by the Enterprise Resource Planning (ERP) system, Jeeves [7]. These orders are the continuous input to the Entize system. In some future version of the cloud service solution, there will most likely exist software packages to connect with other order and invoice-handling systems. The prototype will not include this functionality, since it is not the intention to build a complete solution but rather to demonstrate a proof of concept.

Entize Planning makes use of MongoDB [8] for long-term storage. MongoDB is a Not Only SQL (NoSQL) [9] database, or more generally a document database. A document database allows for storage of data structures that includes objects stored in objects. In an SQL database, the objects would be stored in different tables and be connected by a reference such as an id. In the document database, the second object can be stored as is within the parent object. Besides storage capabilities so are other means of data retrieval enabled for NoSQL databases. MongoDB supports the format JavaScript Object Notation (JSON) [10] and stores in the Binary JSON (BSON) [11] format. JSON enables storage of structures like arrays with strings and arrays within arrays, while BSON is the equivalent for serialised binary data. An example of a JSON formatted document could look like this:

```json
{
  "id" : "headline",
  "title" : {
    "en" : "Change title",
    "sv" : "Ändra titel"
  },
  "tooltip" : {
    "en" : "Some help text",
    "sv" : "Någon hjälptext"
  },
  "_type" : "http://sharejs.org/types/JSONv0"
}
```
Existing Node.js NPM modules used in the Entize project are: Racer, Tracks, Derby, Share, Express, Redis, Livedb-mongo, Racer-browserchannel, Connect, Mongodb, Derby-ui-boot, Tedious, Moment, Underscore, Async, Nodemailer, Derby-auth, Mongoskin, Coffeeify, Racer-fixtures, Racer-util, Lodash, Expect, Expect-almost, Msnodesql. Some of these have already been introduced; other modules worth mentioning includes: Redis [12], which is a key-value store used by Derby to handle the runtime changes on models. By subscribing to a model, any changes made upon that model will be automatically pushed to all its subscribers. This enables a seamless update of clients when the model change. Express [13] is a middleware that makes Derby function. Express ties together different components. It is a middleware that lets a Derby application handle model changes, routing and configuration. Underscore [14] is a utility library that adds utilities that developers often need to implement themselves since they do not exist natively in Javascript (ECMAScript). All these libraries are Javascript libraries which makes them easy to integrate. Basically any library written in Javascript could be integrated into the Derby framework, something which makes the available functionality at any time quite huge. Moreover, it also becomes a problem when choosing support libraries since there exist quite a few that overlap in functionality.

Currently, all installation, deployment and configuration in Entize Planning are done by hand, and per request from the consumer. There is no plug and play or configuration functionality in store for new consumers. These features should ultimately be provided by the cloud service, and ideally be managed and handled by the consumers' interaction alone. In the libraries previously mentioned, there exist a lot of functionality to accommodate such features. However, requirements and design decisions will be addressed during the design and implementation phase, at which time the management resources, deployment and configuration, also will be discussed.

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumers may need to be able to create new templates on demand</td>
</tr>
<tr>
<td>Must support personalised protected resources for each consumer</td>
</tr>
<tr>
<td>Automatic deployment for new consumers</td>
</tr>
<tr>
<td>Support for initial data (settings) retrieval for new consumers</td>
</tr>
</tbody>
</table>

### 2.2 Cloud Service

Cloud Services [15, 16] are services that enables the consumer to choose when, what type, and how much hardware and software resources to pay for in order to host or utilise the consumers own, or by the provider pre-defined environments, applications or services [17]. The services often come with different types of security features since the consumers does not handle the data or processing of the data locally. The service provider often ensures the integrity of the consumers' data by managing backups, encryption and transfer protocols. The accessibility of
the data is often regulated through a client with different types of security features.

As specified in the definition of cloud computing [16], provided by the National Institute of Standards and Technology (NIST) [18], there are different forms of cloud computing. The main differences being what the infrastructure for the cloud system allows in terms of: control, access, applications or services, operating system interaction, virtualisation, monitoring, storage, networking, and scalability for the consumer;

Different types of cloud architectures have different purposes regarding the general service they provide. For instance, choosing the cloud environment service level:

- **Software as a Service (SaaS)** - allows the consumer to gain access to the provider’s applications, through a client, managed by the provider, supplied by the provider and where the consumer often has the ability to partially configure the service. For example: Google Apps [19].

- **Platform as a Service (PaaS)** - enables access to the prepared platform of the provider’s tools and some configurability in order to manage the consumers’ own applications or services. The consumer is still confined to the environment put in place by the provider, but has more freedom compared to the SaaS level. For example: Heroku [20], Linode [21] or DigitalOcean [22].

- **Infrastructure as a Service (IaaS)** - with which the consumer will gain access to virtually the entire platform within the bounds of the infrastructure. The ability to deploy and run operating systems, applications or services supported by the infrastructure is enabled. Some control regarding network-features may be provided but the consumer is still acting out of the confines of the cloud infrastructure. For example: Amazon EC2 [23] or DigitalOcean [22].

Although these levels may seem discrete there can exist combinations or variations in the level of bounds setup for the consumer. Some parts of a providers architecture may be set up as a SaaS service and another as a PaaS. Essentially there might be a greyscale regarding what type of service level a provider actually provides.

If we consider the three service levels or modes as a stack as seen in Figure 2, then the IaaS level would be the foundation. The IaaS level would normally be managed by platform-engineers. The PaaS level would be the platform for the application-developers building services or applications, used by the SaaS level, which provides the services and applications as is, to the end-users. Basically, in the IaaS level, the consumers are platform-engineers. In the PaaS level, the consumers are application-developers and finally in the SaaS level, the consumers are endusers. From a providers point of view, it is a matter of how much of the complete stack the
provider intends to provide, and ultimately who the intended consumer is suppose to be.

Choose what service level to provide the consumer
Choose when, what type, and how much hardware and software resources to utilize

2.3 Economy

Small projects can have a hard time competing in a software market that does not accept failures regarding service delivery nor security. In order to prevent this kind of failure, the built software must adhere to some minimal quality standard in order to generate revenue. The ability to scale resources up and down to reflect the need is a very important step, perhaps a crucial step, for the economy of a small project. Consumers may come and go more often, or the overall budgets buffer might be small. However, depending on resources, both structural as well as intellectual [24], the up and down scaling can be the edge needed for smaller projects to be able to supply its consumers with a reliable product, where the resources have been spent solely on development. Additionally, without the need for the consumer to own the corresponding hardware or software, a more cost-effective model for the consumer is in effect and may in its turn attract more consumers. Instead of support of locally installed systems, the resources can be placed on developing a solid product with a greater chance of survival on the market. Also strategically, a cloud service does not need to allocate more than what is needed for the consumers’ immediate needs, and requires no pre-estimates for allocation.

Cloud services requires a component stack that might include third-party software. Open source software may be preferable when it comes to choosing technology to incorporate into the tool-chain for a cloud service solution. To minimise the cost for software licences, stable open source solutions [25] - with traction on the market - can be used instead of technologies that seem promising - but that has just seen the light of day. The reason for avoiding new tech-
nology might be the lack of documentation, rapid software changes, uncertainty regarding its continued development or unclear community response. Any of these mentioned factors might introduce indeterminable cost.

Sufficiently stable open source software might be preferable in order to cut cost

2.4 Latest fashion

Amazon EC2 [23] or CloudDrive [26], Microsoft Azure [27], Apple iCloud [28] or VMware vCloud [29] are all examples of cloud hosting or computing services. When there are regular or intermittent changes in the need for computational capacity or storage, Amazon and other actors on the market can provide the means to fulfil the needs of their consumers.

As the Internet becomes more and more interactive, and users can live with the fact that their data is stored somewhere else - beyond their control - rather than on their own local machines, services grows to accommodate this change. Except for data, even wordprocessors (Microsoft Word [30], Pages [31]), games (Steam [32]) and other applications move out into the cloud. Cloud services or cloud computing becomes a more attractive solution for companies that wish to reduce their cost for hardware and software purchase and management. This thesis will attempt to clarify and in some fashion describe the process of moving an application into the world of cloud services.

A reason for cloud-services being more and more common is that there are now technologies able to protect non-local data and securely transfer the same data. Data being accessible and not inhibited by any time or place, also makes cloud services popular. Furthermore, the popularity of social media and personalised content contributes to make the bridge of trust related to this architecture more solid.

Security technologies must be able to protect non-local data integrity and transmission

2.5 Aspects

In software engineering there are many aspects to consider. For example, maintainability [33] alone, is a hard thing to obtain and includes: Maintenance capabilities now and in the future; keeping the services up and running; the ability to upgrade technology in a smooth way, transparently even, in terms of consumers, upscale and downscale capacity (memory) on demand without affecting the accessibility; the protection of the integrity and the content of the data without introducing too much overhead or latency are just some of the problems that have to be taken into account when dealing with non-local storage. By choosing a less compatible set of softwares working together, then any of these previously mentioned features may become unsatisfactory. These aspects warrants a thorough investigation of the selection and testing of the underlying software.
Another important aspect regarding cloud services is the ability to monitor status. In order to be able to react to problems that might arise, there must be functionality in place to easily determine the current state. Cloud environments need reliable tools for monitoring, and also for recovery. Some problems might have easy fixes and could be automatically recovered. Of course, there is much more intricate problems that might require a pair of analysing eyes, but with a good design and low technical dept [34], many problems can be prevented or augmented. Automatic flows often require more development but saves resources later on. It can be valuable to have this in mind during development, since these features, both monitoring and recovery, tend to become rather complex when added as a last measure. Monitoring, for instance, is often tied into the normal flow of processes, and sometimes the structure is built so that only at that certain point in time, so can status output be generated.

Cloud environments almost span across the entire software stack from operating system specific processes (scaling, monitoring, logging, virtualisation, memory management, networking), to the applications (toolkits, databases and server softwares) running on that platform, to services hosted by the applications (management-interfaces and consumer-related services) all the way to the web (clients). This puts the developer in a difficult situation, since each step in the chain must provide input and consume output to and from other components. The risk of introducing unmanageable complexity is imminent. There are some techniques worth having in mind to combat complexity:

- Make isolated features - Keep the number of internal and external links between components as low as possible throughout the entire environment, ideally just one communication-channel between layers or components. The fewer connections components have the easier it becomes to swap, update or remove components.

- Make features autonomous, having its own resources, so that if one feature stops working, the effect is limited to that feature, essentially avoiding single point of failure (SPOF) [35].

- Avoid timing sensitivity or dependency - If a feature is dependent on another feature, and this dependency is not avoidable, then time should not affect the feature in such a way that it crashes. Still, it is acceptable to wait until the necessary data is available. There is an advantage to incorporate such a strategy since a lot of applications nowadays can provide, a so called offline mode, meaning that changes to settings or data might be synchronised at a more convenient time.

- Map the requirements, and make a rather complete view of a feature before initiating development. Sometimes it can be hard to implement additional functionality into a feature at a later time without introducing complexity or introducing propagating needs for changes, elsewhere. Usually, it is also more time consuming.
• Reuse components - If components are made modular, i.e. constructed in such a way that they have one specific purpose, then the components can be reused. This is very common in object-oriented programming languages (OOPs) [36], but can also work with more functional programming languages. The point being that many different versions of the same feature decreases readability, which in itself increases complexity.

Of course there are many technical and philosophical debates regarding software structure, design patterns, documentation and interconnectivity. Still, the aforementioned techniques to combat complexity are only reminders for the work that lies ahead, pointers that might be valuable to adhere to when implementing a prototype for a cloud service. Most of these strategies mentioned here comes from experience and support can be found here [37].

<table>
<thead>
<tr>
<th>Upgrade and capacity changes must be transparent to consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud services must be sentient to some extent and be able to recover from bad states</td>
</tr>
<tr>
<td>If possible, avoid unnecessary complexity</td>
</tr>
</tbody>
</table>

2.6 The general picture

Given the general information regarding cloud services, it is time to narrow the scope a bit. Seeing the application, the product or the distributed entity as a component that is supposed to "slide into an empty slot" in the cloud service architecture, might help as a means to understand how the cloud environment is supposed to function. It is safe to assume that middleware or software intended to connect the component with the cloud environment will be needed, like an umbilical cord. Nevertheless, since the consumers are end users, that do not need more flexibility than the Entize Planning product itself provides, it follows that the service level that the cloud platform in charge of distributing Entize Planning is to provide, is Software as a Service (SaaS). Being the developer of the prototype, and as such afforded the choice of what to implement, the entire stack from Infrastructure as a Service (IaaS) to SaaS do not need to be built. The SaaS level however must be implemented for a prototype to be considered as done.

Since there exists many IaaS and PaaS services that can provide a suitable platform to build upon, one task will be to decide which service(s) to use. This essentially cuts a large part of the workload, and distributes the overall responsibility regarding quality assurance, control, monitoring, accessibility, delivery and security for the prototype. It is important to not overdo and start implementing the entire stack, since the control of all functionality is not required. Resting on other services also naturally provides a distinctively layered structure that can facilitate change.

Components in the Entize planning product that work as templates for parts of the product pose a bit of a challenge. These templates are currently programmatically maintained. During the initial step, the deployment of the service, it normally occurs a phase where data are collected for the customisation of the product. Such a collection phase could perhaps include optional standard templates and also the ability to model a customised template specific for a consumer. Regardless if the prototype will support this kind of feature or not, these kinds of
customisations do compromise the separation between the application and cloud service logic. Ultimately, it may prove to be desirable to leave the responsibility for such customisations to the application itself.

Software should act as a connecting layer between cloud environment and application to separate the application logic from the cloud architecture logic.

2.7 The prototype

Chapter 2 - Background, covers reasons for the thesis, and explains the development potential for the application Entize Planning and also strategies to help during the process. To that end, so has a couple of outlines been identified and exposed in the various sections of this chapter. These bullets include expected functionality and development guidelines. In a fashion these bullets also provides a reference for evaluation later. A summarised list of these bullets follows here:

- Choose what service level to provide the consumer
- Choose when, what type, and how much hardware and software resources to utilise
- Sufficiently stable open source software might be preferable in order to cut cost
- Security technologies must be able to protect non-local data integrity and transmission
- Upgrade and capacity changes must be transparent to consumer
- Cloud services must be sentient to some extent and be able to recover from bad states
- If possible, avoid unnecessary complexity
- Consumers may need to be able to create new templates on demand
- Must support personalised protected resources for each consumer
- Automatic deployment for new consumers
- Support for initial data (settings) retrieval for new consumers
- Software should act as a connecting layer between cloud environment and application to separate the application logic from the cloud architecture logic

By listing these bullets, verbalising them and in a way pointing out the expected characteristics, it will be easier during development to revitalising the concepts and continuously ask whether they are preserved. The time allotted for completion of a prototype that adheres to all these bullets mentioned above might not be enough though. Aside from the time needed there are also demands from the Entize Planning product’s current state that has to be observed. All in all, the prototype should however generally exhibit these characteristics.
3 Design and implementation

3.1 The lay of the land

During meetings with the company supervisor for the Entize Planning product, a few requirements were introduced, e.g. that the Entize Planning project does not wish to spend resources on management of hardware; hardware support should be insourced on a need basis. This essentially means that the infrastructure (IaaS) and platform service (PaaS) should be managed by third parties. This requirement transforms the project from a several-person endeavour to a manageable one-person project. Particularly, the project becomes scaled down to only include the control of the PaaS service hired and the SaaS service hosted.

A PaaS service can provide the means to spawn new virtual machines, with different resources, and also the means to control what status, size, operating system, access, recovery, security and the cost the virtual machine will have. The prototype will therefore remotely control such a service in order to provide the platform for the Entize Planning software. Although, it would have been a great challenge to build the virtualisation software, it will ultimately be more prudent to rely on a service that has around-the-clock support and supervision. The solution will thus rest upon other cloud services that have the necessary expertise in that kind of hosting. Today, the Entize Planning is running in an Ubuntu environment, and there is no apparent reason to change the environment, which also adds the need for a service that supports Ubuntu distributions.

Before the prototype can control a PaaS service, the service has to be chosen. Three alternatives have been reviewed, two similar and one more comprehensive:

- Heroku
- Amazon Elastic Compute Cloud (EC2)
- DigitalOcean

The first of the services in the list are the Heroku [20] cloud platform. This platform has support for Node.js preinstalled on a newly spawned Dynamo (Dyno), which is their equivalence of a virtual machine. The Dyno is however not the same as a virtual machine: the consumer has access to a prepared environment that runs on the virtual machines operating system instead. Today, there are a total of ten software packages provided for Heroku-platform developers:

- Ruby
- Rails
- Java
- Spring (Java framework)
Heroku provides a command line toolbelt [38] that can be used to manage local or remote clients running in the Dyno. The toolbelt includes a simulated cloud environment for local development of applications, as well as extensive support for migration of the finished applications to the Heroku cloud. Moreover, the toolbelt offers version control. Had the Dyno been a normal virtual machine, then the platform’s support would have been the same, as the operating system’s support. Herokus Dynos share the CPU with other Dynos. A standard Dyno has 512MB RAM and one share of the CPU power from a quad core Intel Xeon X5550 @ 2.67GHz. Since the usage of the shares are currently unspecified - unclear - what amount of the actual computational performance the Dyno delivers. Even though the Heroku solution can support the prototype, it comes at a relatively high price. If 100 Dynos was to be allocated for clients, then the total cost would end up at around $3,500 per month.

Next, let us consider the Amazon Elastic Compute Cloud service (EC2) [39]. Amazon offers the developer virtual machines called Elastic Cloud Units (ECU), and a variety of distributions, including Ubuntu. Technically the service is both an IaaS service and a PaaS service when combined with Amazon Web Services (AWS) [40], which also makes Amazon’s services the more comprehensive option. Supported operating systems can be found on the Amazon Marketplace [41]. Amazon offers a variety of configurations, including configurations for more specialised needs. A small ECU delivers roughly the same computational power as a Heroku Dyno. Depending on whether the task is memory intensive or computational intensive the Dyno and the ECU will offer slightly different results. The Amazon platform seems to be built around Amazon’s Marketplace and offers the consumer many options regarding operating system images, software packages and monitoring services. Since Heroku actually rests upon Amazon EC2 instances for its service, the main difference, compared with EC2, is the available software, the large number of configurations and pricing models. For example, EC2 provides for the deployment of virtual machines with Node.js pre-installed. If 100 small ECUs are allocated for clients, then the total cost would end up at around $4,700 per month. The small ECU also comes with 160GB magnetic storage.
Finally, let us consider the service provided by DigitalOcean [22]. This service has similarities with EC2. Particularly, DigitalOcean offers virtual machines with different operating systems and distributions. Ubuntu is included as one of the available distributions, and also comes with a software package suitable for Node.js applications. Other options for pre-installed packages include, MongoDB, Express, Angular and Node.js (MEAN). The smallest size and performance configuration offers 512MB RAM for a single core CPU (virtual), but with 20GB solid state (SSD) storage. A benchmark of DigitalOcean against EC2 [42] shows that DigitalOcean do indeed deliver roughly the same, and for certain sizes better results, in terms of performance. The sizes can be resized up and down when needed. DigitalOcean calls their virtual machines, Droplets. If 100 Droplets was allocated for clients, then the total cost would end up at around $500 per month, i.e. much less expensive. A major drawback with the cloud solution offered by DigitalOcean is that they do not provide load balancing, and thus their cloud solution does not offer automatic distribution of requests among non-saturated machines, in order to deliver a fairly constant response time. However, what they do provide is a simple and straightforward Restful interface (REST) [43] for the developer. The overall functionality is adequate despite its relative low price. Since DigitalOcean provides just the right service when it comes to functionality and environmental resources, and also has an easy interface to work with, it was the PaaS service chosen for the prototype. As a side note, so could an equivalent choice have been Linode [21] which delivers roughly the same service, but with different pricing.

DigitalOcean charges the consumer with a fixed price for the entire service, that changes with the size of the Droplet. Since more resources may be needed for any Droplet that is a part of the prototype’s platform, and a Droplet is easily resized, the Droplet does have the capability to accommodate changes in requirements, both for the product’s clients and also for the platform itself. The fixed price also makes it easy to charge consumers for the Entize Planning cloud service. As a feature, every Droplet can be backed up automatically. However, the automatic backup will add 20% of the initial Droplet cost. The popularity that DigitalOcean has acquired during recent years could be the result of a good average performance and a developer-friendly environment [44]. Since the Entize Planning product uses a MongoDB database, and the support for such a database is provided, it matters not that the databases sometimes are stored locally on each Droplet.

The PaaS service will be used indirectly by the ordering service, Order, and directly by the Nexus service. But the Nexus service is only the maintenance application for the clients in the cloud. All features, such as ordering, maintenance, or tracking of logs will be standalone ser-
VICES as shown in Figure 4. The prototype is actually a collection of services, each with a distinct purpose, running in its own Droplet.

The reason for this partitioning is both to avoid single point of failure, and also to enable the ability to independently scale resources. If a service interruption occurs in Nexus, the ordering functionality should not be affected. By the separation of main features, a single failure will essentially be limited to that feature. Another benefit of the separation is that the services become smaller and more manageable. The partial overview shown in Figure 4 reveals that each service running, on individual Droplets exposes its functionality by providing an interface, a client, to its users. The interface is accessed from the web - by a browser.

3.2 The PaaS layer

The PaaS platform as well as IaaS service provided by DigitalOcean can be introduced further by looking at the control of its service and operations. The DigitalOcean Application Program-

17
ming Interface (API) [45] is a REST interface, which is essentially a mapping of resources and relations. For instance, by sending a request to a REST interface, one may change the state of an object, add a new object to a collection or simply retrieve an object. This is done by building the URL in such a way that it maps to a certain action for the resource. To illustrate:

```javascript
/* Api domain url */
var url_domain = 'https://api.digitalocean.com';

/* URL to power-on Droplet - effectively change the state */
var url_on = url_domain + '/droplets/[droplet_id]/power_on/';

/* Create a new Droplet - adding a new object */
var url_new = url_domain + '/droplets/new?' +
    'name=[droplet_name]&' +
    'size_id=[size_id]&' +
    'image_id=[image_id]&' +
    'region_id=[region_id]&' +
    'ssh_key_ids=[ssh_key_id1],[ssh_key_id2]';

/* Get the available sizes - get an object */
var url_size = url_domain + '/sizes/';
```

As seen by the example for the power-on action, the URL reflects the resource `droplets`, and requires an id for the selection of a specific Droplet, and lastly, the action `power_on` is invoked to change the state. For the action `new`, data is required in order for the API to successfully create a new Droplet, which may be obvious by the parameters forwarded in the example. Subsequently, part of the data required to create a new Droplet can be retrieved by sending a request, with the method GET [46] to - for instance - the URL for available sizes, the `url_size` in the example above. Had the resource been a representation of a hand instead, so would the hand have fingers, which might have been affected by sending a request to `/hand/right/thumb/stretch`.

In order to change the state, or create new Droplets the developer needs an account at DigitalOcean. All the new Droplets created using that accounts `client_id` and `api_key`, will be bound to the account for billing purposes amongst others. The `client_id` and `api_key` is forwarded with every request as authentication. A remarkable feature is that a new Droplet deploys in under 55 seconds at DigitalOcean compared to Amazon’s ECU that requires almost 10 minutes.

The DigitalOcean API provides different resources to host the management of Droplets. The `Droplet` resource is the main resource, and the actual virtual machine that is controlled by actions such as `rename` or `snapshot`. The resources are:

- Droplets
When starting a new Droplet, there is a geographical issue to account for. The Regions resource is the remedy. If the site is primarily used in Europe, it might be a good idea to deploy the products on Droplets in the Amsterdam server cluster. If the region closest to the end users is chosen, so will the response time, the time for a request and a response to be exchanged via the Droplet’s service, be reduced. The current clusters are located in:

- San Francisco
- New York
- Amsterdam
- Singapore

The idea is to have world-wide coverage and deliver platforms (PaaS-services) whose geographical location is of little consequence to both the developer and the end user.

The Images resource, is the Operating Systems, with potentially bundled software packages, that can be preinstalled on the Droplet. For the prototype, that is dependent upon MongoDB, Express and Node.js, there is an Ubuntu image with the preinstalled MEAN bundle. Having the MEAN pre-installed reduces the preparation of the Droplet’s environment and also increases efficiency since the time needed to supply a consumer with a new cloud service shrinks. The prototype uses the MEAN bundle for all its own services as well.

There are always security measures that have to be prepared when communicating with web resources. The SSH keys resource enables storage and deployment of SSH keys that are included when creating a new Droplet. In this way, the developer can distribute his public SSH key during Droplet creation and at a later time connect to the Droplet by using an encrypted SSH (Secure SHell) session. The ability to distribute keys can also be used between different cloud services to allow them to communicate securely with one another. A good example for service communication that requires preinstalled keys is the communication between Nexus
and the Droplet for the product. With a different Droplet hosting the product, the Nexus service needs a secure connection for the preparation and the maintenance of the product’s service.

In the Background chapter, there were discussions concerning the economic aspects of cloud hosting. Both small and large companies can save resources by scaling the computational power and storage capacity for the Droplets by utilising the resize action. The Sizes resource delivers all currently available configurations that a Droplet can support. The prototype will make use of the smallest configuration mainly - for its own internal services, and adapt the size for any client serving the Entize product.

The Domains resource provides a means for the developer to setup the access for different services running on Droplets. By setting up a domain pointing to a certain Droplet, the developer are able to make use of subdomains to enable the end users or any internal service access. If the Droplet is viewed as a house, then the address for this house would be the domain and a subdomain the numbering for a certain door within the house. The point being that for every Droplet, there can be many different services, and the way to reach these services is the routing provided by the Domain feature. This setup will be discussed in greater detail later.

All these resources is necessary for the setup of a functioning environment for the virtual machines. The DigitalOcean API can be used in different ways. The API is exposed either by sending requests, server to server, or by the use of DigitalOcean’s web site. In the Node.js stack, there are many modules that can handle the http request-response model. In the beginning of the prototype’s construction, the Request [48] module was used to remotely use the DigitalOcean API. The use of Request mostly entailed the maintenance and the supply of well-formed URLs.

```javascript
var url = 'https://api.digitalocean.com/images/?client_id='+id+'&api_key='+key;

Request(url, function (error, response, body) {
  if (!error && response.statusCode == 200) {
    var responseBody = JSON.parse(body);
    images = responseBody.images;
  } else {
    res.render('error', { recoverUrl: '/droplet' });
  }
});
```

From the example above, it is clear that if nested with other requests, the readability of the code would become low, and more error prone. Luckily a Node.js module was constructed for the communication with the DigitalOcean API. This third party module significantly simplifies the handling of resources when working with Droplets, images, domains etc., since the portion
of self-maintained code shrinks. The module, named rather suitable, digitalocean \[49\], removes the need of URL maintenance, and makes the code readable.

```javascript
/* api is the handle */
api.images.all(function(images){
});
```

Since the api handle is initiated with authentication from the beginning, the developer can focus on the actions to perform - instead of how the URL for the request, is supposed to be built. Even if changes occur in the DigitalOcean API, an update of the module will ensure that the code in the prototype stays the same.

All authenticated requests will provide a response that reflects the status of the request but may also create an error or an event ticket. The error tickets functionality provides feedback on errors that occur while the event ticket is created to allow the developer to listen for the completion of an event that takes time. The creation of a new Droplet takes 55 seconds, and the actions taken to create the Droplet can be viewed, in real time, by the use of the event resource. The reason there is a need to monitor the status of an events completion, is that the web client sometimes is responsible for visually providing accurate feedback for actions issued. Another case may be that, when resizing a DigitalOcean Droplet, the action `resize`, must be preceded by, the action `shutdown`. The fact that we have to wait for the shutdown to occur, before issuing a resize, creates the need for the `events` resource. Subsequently the `errors` resource can be used as a log for errors and can be used later on to determine why an action failed.

### 3.3 An Express project

As mentioned in \textit{Chapter 2}, the product Entize Planning is a client-server application. The client is web based. The platform underneath both client, server and the Derby framework, is the Javascript-driven platform Node.js. This greatly relieves the job of building the prototype, mainly since almost all components are written with the same programming language. Many other web applications use different programming languages for server and client. The most notable advantage of using the same language is that the client and server can share code resources. However, from a prototype point of view, this similarity means that if the need to hook in to the normal operation of the services running out in the cloud, there will be no compatibility issue. During the initial meeting with the company that houses the Entize Planning product, one important requirement was introduced. To be extensible, and thus be able to serve as a basis for a finished product, the prototype must be built on the same underlying platform, Node.js, as Entize Planning, see the Figure 5 for a reference to the public logotype for Node.js.
The requirement that Node.js has to be used, and the fact that Entize Planning is built upon Derby, means that by looking at Derby many libraries used to solve common functionality, can be found for the prototype just by inspection. The main purpose of this inspection is to provide some core functionality that is necessary for the construction of a platform. The main library or framework, and also the plumbing of Derby, is Express. So frameworks within frameworks. This behaviour is most likely a result of the way Javascript is built. Since a Javascript application essentially is a large object, consisting of other objects, there is no restriction upwards, that is, if a finer grained layer can be built upon the previous layer, and that exercises more code, with less code written by the developer, then this layer will ultimately be done.

Although Derby solves most of the problems that an experienced developer requires, Derby also adds “magic”, functionality that is solved by Derby’s inner workings, that is somewhat hidden from the developer. Adding layers upon layers is positive structurally, but also less positive since the abstraction makes it difficult to understand how certain functionality is obtained. When examining what the prototype is suppose to do, so is the functionality, and the “magic” that Derby adds of no need, and the lower layered framework, Express, is sufficient. Just to provide a reference as to how easy Express is to use, see the code section below:

```javascript
/* Import express and a https client */
var Express = require('express');
var Https = require('https');

/* Encryption data */
var certData = {
    certificate: 'CERTIFICATE',
    serviceKey: 'RSA PRIVATE KEY'
};

/* Create an application instance */
var app = Express();
```
/* Set the template engine that Express should use when generating html */
app.set('view engine', 'jade');

/* Set a typical value */
app.set('port', 443);

/* Instruct Express to use Middleware to encode URLs */
app.use(Express.urlencoded());

/* Start the web server */
Https.createServer({key: certData.serviceKey, cert: certData.certificate}, app)
  .listen(app.get('port'), function()
    console.log('Express server listening on port ' + app.get('port'));
  });

As follows, it only takes 19 lines of code to create and start a new web application. Although the application does not perform anything yet, it still serves as an example, of how easy it is to control and manipulate different resources in Express. Derby adds yet another frame around this usage and introduces functionality like Share, which can instantly propagate data between multiple client instances. The developers of Derby have ideas as to how an application should look like, function, and be structured. They enforce this structure in a way that can be beneficial for normal web applications. For the prototype, however, there is no immediate need for such structure, and also no need for many of the features Derby provides. To this end, the first service for the prototype, henceforth known as Nexus, will use Express for the bulk of the program.

By selecting Express for the production of Nexus, a Node.js framework is used and naturally it follows that the core functionality of Node.js, is the actual core. To include new functionality, Node.js provides its developers with the Node.js Packaged Modules (NPM) [50] manager to handle management of dependencies. Whenever there is a dependency manager involved, there is an implicit structure too. This provides for a natural starting point for the review of a general Node.js/Express projects’ structure:

```
nexus (project root)
  application.js
  node_modules (imported dependencies)
    express
      node_modules
      index.js
      package.json
  public (public resources)
    media (pictures, movies, etc)
    javascripts (external javascripts)
```
The file package is formatted using JSON and defines dependencies, the application name, version and where the application starts its execution:

```
{
    "name": "nexus",
    "version": "0.0.1",
    "private": true,
    "scripts": {
        "start": "node application.js"
    },
    "dependencies": {
        "express": "3.4.8"
    }
}
```

Since a project can start more than one application, the scripts member can contain many members. When a module is built and deployed to the NPM module repository, the name and version is used to make the new module available as a dependency for other applications. The dependencies member displays such modules, i.e. modules that have been pushed to the NPM repository. By adding a dependency, Express, and its version, 3.4.8, so can the NPM, when instructed, download the module to the folder node_modules, and the new module can be imported by using the require keyword, from the server, the application.js file. The folder public serves all static resources for the web application. Static resources include stylesheets, Javascripts and media that can, without any restriction, be served publicly by the the web application. The routes folder contains scripts, that expose resources based on the URL requests to the web server. The views folder contains files or folders with templates or HTML pages that the web application can render. Render means that the page or part of a page can be requested by an http client like for instance a web browser. The project can include many other folders containing resources but these folders will not be accessible or used automatically by the Node.js application. The projects' structure differs mostly as a result of the framework that is used. The structure presented here is the standard sample of a project using Node.js with the Express framework.

The prototype’s development was conducted with the operating system Mac OS X - v10.9.2
which is Unix-based, and provides its developers with the Terminal application. The Terminal provides different shells, and can run scripts or commands within those shells. Generally, a Node.js application will be extended within the node_modules folder and grows when the NPM command install is used from the command line. Node.js own core modules are installed for all projects, and can be extended by using the flag -g in conjunction with the NPM install command. Now the structure of Node.js, the ability to extend the application, and the installation have been covered. To start an application, the command npm start is used from the command line.

The Node.js web application serves the consumer HTML (HyperText Markup Language) [51] pages using a view engine. There are many types of view engines available that can generate HTML pages. The default view engine for Express though is Jade [52]. Express gives the developer the choice of which view or template engine to utilise. The strengths of Jade are that it removes all unnecessary markup that are normally associated with html pages and includes the tools needed to create dynamic pages, as well as the use of template logic i.e Javascripts bundled with Jade to produce a certain HTML, based upon some conditions made from the web server. The prototype’s services uses the default view engine, Jade. One of the markup element Jade removes from the HTML is the closing tags which instead are replaced by indentation, similar to the programming language Python. So the scope of the tag is the indentation, and the tag itself omits the angle brackets to make the code more clean, and easier to read for the human eye. Here follows a snippet of Jade template code, see Figure 6.

```
<each log in nexus_log>
  <div log-entry.col-sm-12(data-hook=log.status)>
    <p method=log.method>
      <p url=log.url>
        <p status(style='color: #'+log.color+' !important')=log.status>
          <p response-time=log.time>
            <p content-length=log.length>
              <p datetime=log.datetime>
  </div>
</each>
```

Figure 6: Jade code block

The corresponding HTML code disregarding the the first line in the Jade code can be seen in Figure 7.
The main difference here being that the first line in the Jade code example allows for the presentation of all log elements in the JSON formatted array `nexus_log`. Whereas, the HTML code segment would, without a template engine, grow with eight lines of code for each log element in the `nexus_log` array. The `nexus_log` array is served from the web application’s server as a variable in the render statement. The end product is always the same but the code base managed by the developer is constant no matter the length of the array. This simple loop example is a powerful feature when needing to present large sets of data. Another powerful tool is conditions. A simple `if` statement can decide whether a portion of a template is supposed to be rendered or not. This feature allows the developer to switch code blocks on and off when rendering different views, perhaps depending on the consumers role, or the presentation mode of the page. Toolkits, error labels or other components can be included at all times but not rendered since the user does not have the privilege, or the state presented, does not require the components. There are of course similar template engines for HTML also. Jade supports inline Javascript by the `script` keyword, just like HTMLs’ `<script>` tag.

When presenting HTML pages, then their visual appearance are mostly a result of the style sheets, CSS (Cascading Style Sheets) [53]. The styling can be applied in different ways. If the style is applied inline, in the html page elements themselves, then the style is applied using the style attribute for each tag or within a common style tag included in the HTML pages head tag. Another way is to use the style sheets. Style sheets are files that include, parts of, or all the styling, for the entire site. These are then imported from the `head` tag of an HTML page. Although the stylesheets is a nice way to keep all the CSS rules in the same place, so do large web applications tend to produce - so many CSS rules, that they have to be split among several files, sometimes one for each web page, or one for each feature, may be needed. Regular CSS is structured by applying style rules onto the HTML document tree. Selectors, which are used to single out which elements the styling should be applied to, can be:

```
/* Tag selector - all div elements will have this style rule */
div {
  color: #333333;
}
```
/* Id selector - all elements with the attribute id="logSection" will have this style rule */
#logSection {
  color: #666666;
}

/* Class selector - all element with the attribute class="log" or class="cubic" will have this style rule */
.log, .cubic {
  color: #999999;
}

/* Id selector - just added to make a point in the next example */
#list {
  width: 100%;
}

/* Combined - all children that are links under element with id="list" */
#list a {
  color: #AAAAAA;
}

The CSS example above is the code style that can be parsed when included in the HTML page. However, there is ways to generate the same CSS while using fewer lines of code, make it dynamic, and at the same time cleaner. Leaner CSS (LESS) [54] is perhaps the preferred way of producing CSS now. Just as the acronym suggests, the same CSS can be produced, more readable and can also partially be reused. There is a Node.js module called Stylus [55], also a CSS pre-processor, that is very similar to LESS. Stylus takes the abstraction a little bit further. The CSS in the previous example can be produced with Stylus in the following way:

div
  color: #333333

#logSection
  color: #666666

.log, .cubic
  color: #999999

#list
  width: 100%
Although this example does not do justice to the more complex and powerful things that can be done with Stylus, the example does serve as an example of a more readable syntax. Just as with the Jade templates, the use of indentation allows for nested rules that changes the scope, and also reflects the way an HTML document tree is structured. Instead of creating a new rule, the rule is added for the link a within the rule for the id list. Stylus was used for the development of the prototype's services.

To be able to get the shell of the application up and running as fast as possible, a library was used. There is a lot of open source CSS libraries available. To get styles for common components such as tags, buttons and overlays etc., the quite well known Bootstrap [56] library was selected. In addition to more lean and readable code, Stylus and Bootstrap also provides additional functionality. They add CSS rules to compensate for both the screen used when consuming the web page and also for the Browser used. Old versions of Internet Explorer do not support some of the newer CSS features that exist, and other browsers use web kits that differ slightly. To make the site easy to use for everyone, a good idea is to use CSS pre-processors and user agent [57] adaptable CSS. Since it is time consuming to lookup what CSS support all web browser have, the CSS pre-processor provides an easy way to ensure that the basic functionality at least works.

3.4 The prototype in action

To illustrate the different layers, services and how they interact with each other, a review of the application’s flow for the change of a Droplet’s name will follow. Since Express applications are event driven, so would the natural starting point be, the click event, initiating the entire chain. Figure 8 shows some of the functionality that is provided by the Nexus service.
The Droplet we will change name for is *db*. By clicking the *Rename* button, a client side function takes over. The name and the id has been prepared as input to the buttons *onclick* event. The event attached consumes the Droplet id and name in order to present an overlay with the current name as placeholder. The placeholder attribute in the input tag can be used as a reminder to the user of the previously set value.

Figure 9 shows an overlay, it is just a layer above the current screen that has been pushed forward by CSS in the web pages z-axis in order to present data, or handle custom events, within a finite space (the screen). Instead of loading another page, the overlay is placed above. When a new name is set, the user clicks on the *Rename* button, and triggers another event that was
attached during the previous click. The new event instructs the browser to request a URL:

```
/droplet/1384509/rename/ { "name": "db2" }
```

This URL is relative, it has no domain or subdomain prefix (like https://www.google.com/), meaning that it will be handled by the web server that served the current page, the request will therefore be handled by the Nexus Express server. The URL has the same formatting as a URL in a REST interface, with the addition of the JSON data, :data that the controller in Express accepts. There has to be a controller on the server that can handle the request, otherwise the request will be collected by a general controller that displays an error - when trying to request a resource that does not exist. For the request mentioned above, however, a controller is prepared:

```
/* Change state for a Droplet */
app.get ('/droplet/:droplet_id/:command/:data', isLoggedIn, function(req, res, next) {
  var data = verifyData(req.params.data);
  var droplet_id = verifyDropletId(req.params.droplet_id);
  var command = req.params.command;
  var commands = [];

  if (!data || !droplet_id) { res.render('error'); }

  if (ApiControl.validDropletCommand(command)) {
    commands.push(command);
    performDropletAction(res, commands, droplet_id, '/droplet', api, data);
  } else { res.render('error'); }
});
```

If the URLs pattern can be matched with the controllers signature, then the parts of the URL can be retrieved through the requests params member. An important, but not very protruding feature, is the function isLoggedIn. Without covering the concept Middleware [58][59] in great detail, isLoggedIn is a Middleware. Middleware is similar to a software layer that extends the functionality. Before the function that handles the request and response is processed, the isLoggedIn function must be processed. The entire Express stack is essentially Middleware layers, upon layers. A request travels through some of these layers and is used and processed in different manners. There are Middleware for sessions, logs, JSON-parsers, URL encoding, cookies, flash-storage etc. The isLoggedIn middleware inspects the current session to evaluate if the users requests are authenticated or not.

After retrieving the data to send, the command (the action) to perform, and the id for the Droplet to change, a verification of all the parameters is necessary in order to continue. The
call to performDropletAction includes the handle to the DigitalOcean module, api, and the final page to render; along with the response object needed to forward the request in the new scope. The strategy to forward the response object might seem strange, but has a natural explanation. Since the performDropletAction receives a list of commands, and iterates that list to enable Droplet changes that require more than one action, the function must in charge be able to break the flow. The rename is a simple action, it does not require the Droplet to be shutdown, or any other action, for that matter.

When following the program flow into the performDropletAction function another main concept emerges, asynchronous and synchronous requests. Take a look at the following function:

```javascript
// PERFORM CHANGE REQUEST
function performDropletAction (res, commands, droplet_id, endPath, api, data){
    api.droplets.get(droplet_id, function(droplet) {
        if (droplet) {
            droplet[commands[0]](JSON.parse(data), function(response){
                commands = shiftArray(commands);
                completeAction(response.event_id, res, commands, droplet_id, endPath, api, data);
            });
        } else { res.render('error'); }
    });
}
```

One of the trademarks for Node.js applications is non-blocking I/O (input/output) [60]. This generally means that the application code does not wait for requests to return results, but rather continues execution until the request is ready to produce a response. All requests are handled in parallel but the code is executed sequentially. This means that when requesting data from another API, as is the case above, where a GET request for the information of the Droplet with droplet_id is made. Then a second parameter, a callback function, is provided. The callback will be run when the api.droplet.get receives a response. This is an asynchronous request (call), the request does not block the program execution. When the callback is being processed, the droplet object is available, and can be used to perform actions upon. Also the droplet[commands[0]] request is asynchronous and thus has a callback to process the response. The state change request on the Droplet returns a response containing an event id. Meanwhile the change is made at the DigitalOcean server; the change’s status can be monitored by using the event resource provided by the DigitalOcean API. For example, consider the next function that is called, completeAction:

```javascript
// WAIT CONSTRUCT FOR EVENT (PROMISE) COMPLETION THEN REDIRECT
```
The `completeAction` function is special. It actually uses blocking to wait until the response from the event object returns an `event.percentage` that is 100%. Although this construct differs from the conventional Node.js application’s flow of control, there are cases when blocking is the only way to produce a certain result. The Promise construct [61] with the `promiseWhile` loop makes asynchronous calls, in a synchronous manner, essentially blocking further execution. The reason for this use is that the next command or state change on the actual Droplet can only be done when the previous action has been completed. There is also no practical reason for the user interface to present a state that does not reflect the actual state of the Droplet. The need to wait for state completion, leaves the developer with either the option of using Promises, or the use of web sockets (long polling), or any other means of signalling. The strategy that is most cost effective regarding memory, traffic and latency is the Promise construct. The Promise will wait for a response, and when that response arrives fire the `resolve` action to continue processing. The `promiseWhile` loop will check if the condition is met, and if so, allow the `then` clause to process. The three functions presented in the previous examples, perform roughly 60% of the available state changes of the virtual machines. Another solution could have been to make a controller for every single action and let subsequent steps carry out each state change. However that is a poor use of the construction of the DigitalOcean API and module. The need to wait for each action to finish, would still have to be addressed.
The only step left after the completion of the commands that started the chain of events, is to produce feedback to the presentation of the Droplet’s state. Essentially using the response object to render the view with the new data available.

3.5 Persistence

When a new order arrives, a new session is created, a request is logged, or a new user is added, there is a need for storage capable of keeping persistent data. That is, data that must survive the next page request, a client shutdown or for a certain period time. Persistent data often entails the use of a database solution, contrary to a cached, in-memory database [62]. Express provides Middleware for storage of sessions, and modules for the use of a variety of database types. The type of database to use is mostly decided on the basis of the type of data that will be stored, the internal relations for that data, and consequently how that data is accessed. If the data has many internal relations, then some data might be accessed through other data records, like social media for instance. In order to access records transiently, for the purpose of querying other records, for instance, retrieve my parents buddies, then the database must support such features. Some applications make use of many different databases for different kinds of data, in order to optimise the storage efficiency, and provide a clean separation of storage logic. The kind of data that the cloud service prototype is suppose to store is in JSON format. The internal linkage of the data will be small, and a document database is sufficient for the storage of the prototype’s data models. MongoDB is a document database and has Node.js modules that provide abstractions for different kinds of data. For session storage, there is the connect-mongo module for model data there is the mongoose module and for web socket data, there is the mongo-watch module etc. Except for the explicit storage and access of data, there are other considerations. Strategies for backup, load balancing or redundancy, for a service that has to survive system failures, malicious attacks, or heavy loads, needs to be in place. The prototype will not need to endure heavy load but might be subject to attacks or systems’ failures. Since the prototype delivers a product to other consumers, there are actually two databases to consider. Even though the Entize product is configured to use the MongoDB database for local storage, there is a reason to rethink its database setup and usage. Since the Entize server software now will run on a Droplet, there are different resources available. The usage of memory, CPU and the amount of data normally transferred must be considered. When allocating the smallest Droplet with 512MB RAM, one core and 20GB of storage, the consumer receives a transfer limit of 1TB of data per month. By placing the database on a separate Droplet, both the performance, fault tolerance, and transfer limit change. Also by setting up redundant databases spread among a variety of Droplets, would result in a considerable change.
MongoDB gives the database administrator many options regarding the setup and management of data. To meet the requirements for a database exposed to a heavy load, the MongoDB crew enables a technique called *Sharding* [63]. A cluster setup with Sharding means that the data is spread among many machines: a set of shards of the whole database, accompanied with configuration servers, that stores the database metadata:

![Sharded MongoDB Setup](image)

By Sharding the database, see Figure 11 for a conceptual overview, the overall load will be spread among many machines and the scaling, that is, the expansion of the storage capacity, will be done by adding a new machine, a new shard, to the cluster. Clusters set up using Sharding is horizontally scaled. A Sharded MongoDB database will distribute the load, increase the read/write throughput, and enable scaling of large databases. Since the prototype is unlikely to have to handle a heavy load, will not need to endure heavy load, so even while deployed on a
small Droplet, will not need to be set up with Sharding. Farther into the future, however, reasons may arise for the support of a Sharded database cluster for other products, then Entize Planning.

Fault tolerance, redundancy and security is however issues that need to be handled by the prototype. Fault tolerance can be assured by the use of backup strategies and redundancy. In the case with Droplets, there are more than one strategy regarding backup. Either enabling backup for the entire Droplet or a backup enabled for the MongoDB database-instance [64]. When enabling the MongoDB backup feature for a replica set, there are two different types of backup modes. The first is a mode that the MongoDB developers call “point in time recovery” that has the ability to restore any state of the database within a limited time frame. The other mode is “snapshot” that differs from “point in time recovery” by only restoring periodic states of the database. The prototype will have backup enabled for the entire snapshot but might in the future use the snapshot strategy for backup. The snapshot mode does not require constant health monitoring, and is in that sense more forgiving regarding monitoring omissions in addition to Sharding, the database can be setup as a replica set [65], see Figure 12 for a conceptual overview. A replica set provides redundancy by keeping many copies of the same data:

![MongoDB - Replica set setup](image)

The database exists as copies of a Primary database which is the database on which all the operations occur. By tracking the log of all operations, i.e. keeping the so-called oplog under surveillance, so can the Primary database, the Secondary, replicated databases, be kept synchronised. The Secondary can at any point in time take over, i.e. be elected the primary, should a system failure or a service interruption occur. Service interruptions can be the restart of a Droplet or the system is updated. The maximum loss would be one single operation on the Primary. As the figure suggests, the replica set setup feature provides fault tolerance if the databases resides on different machines. The replica sets members continuously vote for which member should be the Primary. So to avoid deadlock in the voting procedure, there should always exist an odd number of members in the replica set. One machine is the so-called Arbiter.
and has the sole purpose of casting the final deciding vote, since the Arbiter cannot vote it self. The arbiter does not store any replicated data, and can, contrary to the architecture drawn in Figure 12, exists on the same machine as one of the other members. The prototype’s services all use the same database, which is configured as a replica set, and where each corresponding rack-server are a Droplet instance. To cut some cost during development, a test with the replica set setup was made with three different Droplets, just for verification, but later changed to run on the same Droplet. Running all replica set members on the same Droplet exposes the system for failures, and only prevents individual replica set members, mongod, service interruptions.

Since the database stores entries for logs, sessions, orders and users, all of which are different types of collections. These collections are associated with different requirements regarding lifetime and usability. Session documents only survive until the Time To Live (TTL) counter expires, at which time the session expires. Since the prototype only tracks users of its services, the sessions number of collected sessions are always quite small, and the collection for the log application always grows. The collection of the orders made, always grows. Since some collections always grows, there has to be functionality to handle ever growing collections. Regarding the logs collections there are no practical reasons to track the requests or system outputs for a longer period of time. MongoDB has a couple of options for handling the automatic removal of old entries. The one used for the logs collections is enabled during creation of the collection. By adding the option capped and specifying a size limit for the collection, so can the service deamon, mongod instance, automatically discard the oldest entries when the pre-set limit has been reached. By allowing the mongod instance to handle the removal, unnecessary database logic can be avoided in the application that tracks the logs.

The application Logs - that displays all the logs, for the prototype’s services can track all the logs in real time. Since the MongoDB is setup as a replica set there exist an operations log, oplog, that pushes all changes to the other members in the database cluster. By listening on changes in the oplog for a specific collection, those changes can be tracked and updated in realtime. The module mongo-watch enables this functionality. To push these changes from the Logs application server to a client, a web socket can be used. Node.js has the module socket.io that uses the Express https server to setup a socket that can be connected from the Web service to the client.

```javascript
/* Server */
// watch the nexus log collection
try {
  watcher.watch(Config.dbName + 
    Config.applications.nexus + 's', function (event) {
    io.sockets.emit('nexus', { log: event.data });
  });
} catch(err) {
  console.log('ERROR - Websocket is unable to broadcast!');
}
```
The server registers an event emitter that pushes the data to the clients event listener - through the socket. Each service in the prototype saves its log entries in the database, and for each entry, there is a listener that pushes the live updates to the view (web page). Sockets will most certainly be used in the future for the prototype’s internal chat function. Another use case for sockets is the continuous push of monitoring data for the purpose of keeping track of the health status of the services running within the cloud. The web sockets enable the web page to present new data without having to reload the page.

Using MongoDB which is a NoSQL database of type Document store, for storing dynamic non-structured data is suitable when the need to access certain data does not include complex joins or queries [62], which is, for example the case when utilising a Relational Database Management System, (RDBMS) [66]. Even though the data items is non-structured and dynamic, there are some data that are grouped together indirectly. To alleviate the read and write of documents, the use of models, or schemas, can be used. A model enables the developer to group documents together, both for database store and load procedures, but also in order to simplify validation, set the same lifespan (TTL) or for the forwarding of the data. Another benefit of using a model is that the developer does not need to actually access the database in order to inspect the grouping, since that information exists in the applications’ models instead. The mongoose module is one, among many, modelling middlewares that can be used. An example of a mongoose model may look like this:

```javascript
/*
 Define the schema for our request log model
 "method" : "GET",
 "url" : "/",
 "status" : 302,
 "time" : "4ms",
 "length" : "68b",
 "date" : ISODate("2014-03-31T21:44:45.772Z"),
 "color" : "0AB8C7"
*/
```
var logSchema = Mongoose.Schema({
  method : String,
  url : String,
  status : String,
  time : String,
  length : String,
  date : String,
  color : String
});

By using module exports [67], which is an import mechanism in Node.js, the code that handles different tasks can be structured together and imported when needed. The use of exports, enables all models to be placed in the same folder:

```javascript
// Exported from the models folder - log.js
module.exports = function(model){
  return Mongoose.model(model, logSchema); break;
}

// Load the log models from the dblayer folder - applog.js
var Log = require('../models/log')(Config.applications.logs);
var NexusLog = require('../models/log')(Config.applications.nexus);
```

Being able to separate the models and database logic from the applications by providing a distinct layer, a middleware, is a great way of increasing the readability. During the prototype’s early development stages, there were a considerable change in the perception of the code after the exports functionality and the modelling feature were utilised.

### 3.6 Product preparation

When a new product is deployed into the cloud, a few steps have to be taken in order for the prototype to serve a functioning product to the consumer:

- The Droplet is created
- A subdomain is assigned to the Droplet
- The Droplet’s IP is added to the managing Droplet’s ssh config
- The product along with a deploy and a start script is transferred to the Droplet
- The product is installed
• A soft link to the automatic startup script is created
• Connection and login information is sent to consumer

To create a new Droplet at DigitalOcean, so must the Droplet know:

• The Droplet’s name, which will also be the subdomain name (selected by the consumer)
• Which prepared SSH keys to add (are preset)
• That it is supposed to run an Ubuntu 12.04.4 distribution with the MEAN package (is preset)
• Which size configuration to use (might be preset)

If the consumer enters a unique name, containing only letters and numbers, and also a size configuration (not entirely decided yet), are the parameters for the Droplet creation step fulfilled, and a virtual machine is waiting for a product to host. The Droplet also needs additional configuration in order to serve the product’s service to the end user.

After the creation of the new Droplet, the Droplet is assigned an IP address that needs to be bound to a subdomain. The domain name is an alternative to the IP address - easier to remember and more readable. The domain name is later used to locate the Droplet’s services on the Internet. When accessing a URL on the Internet, that name is used to route the request to the correct machine serving the requested resource, a web page, a binary stream, or some other media. Consequently, a site that is hosted on a specific machine needs to be registered somewhere, where these domain names are registered and distributed for everyone. A registrar [68] is a certified vendor for domain names. The domain for the prototype is cloud-nexus.se in the .se top-level domain (TLD) [69]. The registrar for the cloud-nexus.se domain is Crystone [70]. Via their domain service, the domain has been configured to link to DigitalOcean’s name servers (DNS servers). Such information is public for everyone, and can be accessed by a simple command line tool (Unix) called whois:

```
whois cloud-nexus.se

// Response on whois command
...
state: active
domain: cloud-nexus.se
...
nserver: ns3.digitalocean.com
nserver: ns2.digitalocean.com
nserver: ns1.digitalocean.com
```
By pointing the .se domain name to DigitalOcean’s name servers, their Domains resource can be used to enable routing to individual Droplets. The prototype’s service Logs has been assigned a subdomain [71] named logs, and can therefore be reached on the URL logs.cloud-nexus.se. The new services that the cloud service is setting up also have to be reachable. Each new client is therefore assigned a subdomain under the .cloud-nexus.se domain name. When a new subdomain name is setup, it takes time to distribute the new domain configuration among the name servers. When testing the name server in the DigitalOcean server cluster, its sometimes takes as much as 10 minutes before the subdomain was reachable. Even though the Droplet is up and running, the Droplet can not be reached by any other means then the IP address. Since it takes a considerable amount of time for the domain name to be distributed, the request for a new domain is made as fast as possible in the chain of events pertaining the configuration of a new Droplet.

When the response for the creation of a new domain has returned from the DigitalOcean API, and during the distribution of the domain name, the product is transferred to the Droplet. In order to connect safely to the Droplet, SSH is used. The public key of the service that remotely controls the Droplets, has been distributed during creation of the Droplet, and allows for the service’s access. The Node.js module ssh-control [72] allows for programmatic control of a remote host. The module enables the setup of a controller that can use SSH commands as well as use the Secure Copy Protocol (SCP) [73], which is used to transfer files to a remote machine. During a normal setup of the SSH session between the local machine and the remote machine, some steps require human interaction, which is a problem when both machines are Droplets, and no human is present to interact. To overcome this absence of human presence, a SSH config file can be used to set options on behalf of the user. This config could look like this:

```bash
/*
NOW WE AVOID STRICT HOST KEY CHECKING BY ADDING HOST TO LOCAL SSH CONFIG
Host [remote_host.ip]
  StrictHostKeyChecking no
  UserKnownHostsFile=/dev/null
*/
```

By setting the StrictHostKeyChecking to no, the local Droplet will automatically accept the public key for the remote Droplet. By setting the UserKnownHostsFile to /dev/null, the key and the host will not be saved to the local SSH known hosts file. If the local Droplet later removes the config file, then there will be no saved information or configuration that can be used to further connect the two Droplets without a human present to interact. Another option is to provide a key file to use to register the remote host and key in the local SSH config file. However by providing a key file, the Droplet will be allowed scripting access to all the registered hosts, without
a human present. Since the functionality to remotely control another Droplet are currently only used once (with reservation for future changes); there is no need to save this information in the prototype’s Install or Maintenance Droplet.

With an available SSH session and Droplet on standby, the product, accompanied with a deploy and a start script is pushed to the remote Droplet. The deploy script is a shell script that extracts the product archive and installs the necessary dependencies. The deploy script also creates a soft link from the Droplet’s /etc/init.d folder (Ubuntu) to the start script. The init.d folder contains services that will start together with the Droplet’s system’s start. By placing a link to the start script for the Node.js application, the application automatically boots up when the Droplet is restarted.

The product’s execution is another issue. If the service running on the new Droplet encounters an error, there is currently no way to restart the service. Unless the service restart functionality, is built into the product or started by other software, the service will not survive any failures. All the cloud service prototype’s services use the node-forever module that encapsulates the service, and handles restart upon service interruption. The prototype does not currently handle recovery of the product’s service. If Entize Planning were to crash, the service would not restart. Only a Droplet reboot would trigger such an event. It is not decided yet, who has the responsibility for this functionality, the product or the cloud service environment. Most likely, the cloud service environment should be able to provide such functionality.

### 3.7 Ordering

When ordering a new product, the only interaction done is between the future consumer and the Order service. The prototype is a collection of services with distinct purposes. The only purpose of the ordering service is to collect the data needed for a successful creation of a Droplet and its product. The ordering service is essentially a couple of forms and flows to present the products available, and collect the data required for each product to work. The Order service has fulfilled its role when a future consumer has created a document, an order, in the database. By separating the ordering from the rest of the services, the deployment of the Droplet and its product’s service can be done when possible. Many orders may arrive within a short period of time, and there is a great difference in the time consumed for the placement of an order, as compared with the time consumed for the creation and deployment of a Droplet. Thus, by separating the order and deployment of the two services, the resources for the deployment service can be scaled to meet the load created by the ordering service.

### 3.8 Maintenance

By building services for each main task, as seen in Figure 13, instead of one service performing all, so can each service be stopped and started whenever needed.
The separation also clearly provides a distinct separation between functionality and where its corresponding logic starts and stops. Except for starting and stopping of services, the separation does provide a natural protection against single point of failure (SPOF) [35]. If a service encounters a faulty state, the remaining services will continue unaffected. The ability to upgrade or update a specific service’s functionality without affecting the other services is also a natural consequence of separation. The separation, or modularity of the services also enables the developers to transparently perform maintenance, without affecting the consumers. As the cloud service becomes more popular, and more functionality is needed, then this functionality could be integrated as a new service.

Another important aspect of development is that there are many factors contributing to the success or failure of a software product. By starting the prototype as a small set of services, with the potential to grow, the resources can be spent where they are most needed. Since the services can grow incrementally and can be deployed fast, the entire cloud service can receive feedback early from the community. The early response from the community can outline areas
for improvements, or give a vague estimate whether the cloud service has any future on the market. An early deployment also provides ongoing public relations (PR) for the service, and provides the means for building a repertoire with the users.

The prototype does not share so many code resources between services. The services are standalone. A natural consequence of the modularity is that each service has its own life and its own progress. Each service needs the basic functionality to function, which also results in a bulk of code that is duplicated among each service, from the prototype’s perspective. A drawback of the services being isolated from each other is that code sometimes has to be duplicated, and changes to this code affects several modules, not just one. For example, an update of the prototype’s brand icon has to be propagated to all services within the prototype. Currently, the services are small and the only items that could be shared are perhaps the models. The models are, however, modular and can easily be copied to the other service’s projects. In the future there may arise a need to investigate ways to reduce this duplicity.

Aside from the individual development of each part of the prototype, the prototype does not communicate directly between services. As illustrated in Figure 13, there is no arrow depicting communication flow between any of the services, Nexus, Logs, Install or Order. All services interact via the database. In this way, the data will - that any given service use will always be available. However, the Nexus application is special in the sense that it serves more than one purpose, and can perform more than one task. By allowing the Nexus to have the same functionality as the Install service, tests for new products can be made. Apart from new product’s tests, Nexus can also be used for support, since it provides the means to alter any Droplets in the cloud. There is no actual monitoring of the product’s services, but the Droplet’s status can be maintained from here, if needed.

3.9 Scalability

If the cloud service becomes popular and the load on the services increase, the Droplet’s resources will eventually be taxed. To compensate for growing requirements regarding performance, each service’s Droplet scale independently, that is vertically, by the Droplet’s resize feature. By increasing the computational power and the memory resources, the Droplet will be able to service at a faster rate. However, eventually that scalability will reach a limit. Since the smaller DigitalOcean Droplets are less expensive, the cost does not scale proportionally to the size, so has the prototype been built with this future requirement in mind. By separating the tasks, as seen in Figure 13, and having the install service performing the preparation of new Droplets, the prototype is able to scale the install service’s functionality by having that service running on many Droplets. That provides the capability to scale the Install service horizontally.

The products that run on each individual Droplet is however confined to the vertical scaling capabilities of the Droplet. At present time the only way to alleviate the product’s load, is by placing the database on a separate Droplet. This separate requirement forces the implementa-
tion to provide the means for a multi-Droplet install feature. Currently there is no such feature. But after the initial tests, with the product Entize Planning, running a locally installed database, this feature will be enabled. This means that for each Entize Planning product, two Droplets need to be created, one with the web server running, and one with its database.

Depending on requirements from future products’ services running in the cloud service, the platform must very likely be able to support other kinds of Droplet setups and database setups. Some of these setups have been discussed in the 3.5 - Persistence section. Sharding is the way that MongoDB allows for horizontal scaling of persistent data. That is, the database is spread among many instances that are running on many different Droplets. By adding a new virtual machine with a MongoDB functioning as a part, a shard, of the whole database, the load distributed is even further.

3.10 Security concerns

During data transmissions and storage of persistent data, the data will be protected both in transit and in storage: certificates, one way encryption, hashes, data minimisation and isolation; are security techniques that need to be handled with care, since the techniques have the potential to be used in an insecure manner. Just recently, a weakness in the popular OpenSSL library was found and patched. The bug was named the Heartbleed bug, and was present in the same OpenSSL library that was previously distributed with the Ubuntu 12.04 operating system. This operating system is used for the Droplets that serve the Entize Planning product. An upgrade that fixed this security leak was distributed just a few days later. The fact that this security leak could exist for so long, and expose so much services as a result of its wide spread use, without being found earlier, is a testament to the complex nature of security. An example of the precarious nature of security technologies could be made with hashing. If we omit the fact that hash functions should be cryptographically strong, and focuses on performance, an inexperienced developer would probably opt for the faster and with only performance in mind that would be the appropriate choice. However, when factoring in the security aspect, then a fast hash function becomes a liability, since it allows an attacker to perform a brute force attack faster. So the use of a slow hash function is the correct choice when using hash functions. The prototype uses the slow bcrypt hash function with a salt, which is quite popular, for the hashing of passwords. Apart from perspectives that contradict each other so might others be obfuscated. The fact that as a developer all security concerns must be accounted for, even from within, so are always the issues with local access to the database, which means that some data, like passwords, will always need to be handled in one way, from plaintext (unencrypted) to cipher text (encrypted), and consequently should not be able to be transformed in the opposite direction.

The prototype uses sessions that are encrypted with a session key, to protect all the data fields that are sent within a request. On top of the encryption of the session fields, all traffic between the server and the client is encrypted by the use of Secure Sockets Layer (SSL) [74]. The SSL protocol is used by issuing a certificate that can be used by the client and server to
setup or establish encrypted sessions. There are different types of certificates. Certificates that are issued by a Certificate Authority (CA) [75] are actively maintained and are also bound to a domain and are called Trusted Certificates. Another type is the Self-signed certificate [76]. The main difference between the types are that the Trusted certificate is validated by a trusted third party. The self-signed certificate can thus not claim its validity, and anyone trusting such a certificate must essentially trust the issuer of the certificate. The prototype’s services uses a self-signed certificate currently, mainly since a trusted certificate is expensive, and even more so, when it is a wildcard enabled certificate. The use of a wildcard is needed when the subdomains also need to use signed certificates. Since the new Droplets also need the capability of using certificates for their running web servers, a trusted certificate, that can be distributed freely within the subdomain, will be used in a final product. The use of a self-signed certificate results in a warning notice issued by the web browser when accessing the service, and furthermore serves to lower the public trust for any offers presented by the service.

When all the prototype’s services uses the database and establishes its connections, login credentials are needed, since the MongoDB is configured to only allow authenticated requests (equivalent to a login procedure). Besides the use of authentication SSL can also be used to encrypt the communication. This is an option that has to be set, since it is not the default setting for authenticated sessions when using a MongoDB [77] database. It does not matter if the web server has SSL enabled for the communication between the web client and the server if the server and database do not enable encryption for their sessions. When deploying new Droplets into the cloud and enabling products running on those Droplets, those services must also use secure connections to their databases. The need to encrypt database transmissions also requires that the settings needed for this setup are carried out separately from the product, so that these settings can be included by the install service during deployment of the product’s services. Since the cloud service does not enforce the use of SSL between the product’s web client and web server, since a product may be serving static [78] content only, the use of SSL between the client and the server has to be an option. When SSL is used with MongoDB, then self-signed certificates can be used. However these certificates need to be different from the client-server certificate, since the client-server certificate is stored in the clients web browser.

Between subsequent web client requests, sessions (cookies) used are to store authentication details, usually an authentication token [79]. Since the details required to interact with the web server is available and stored in the web browser, the sessions will be invalidated. By invalidating or expiring the session after a certain period of time, and thus forcing the client to re-authenticate, some security violations can be prevented. The expiration will limit the time frame, that an unauthorised user, with local access, may utilise to take over the account, if a user forgot to use the logout feature. When using Express without explicitly setting the session expiration, the session lives two weeks on the server side, and is during this period available, with the same client side session details, to establish a logged in session. The prototype’s services limits its sessions’ life span to 15 minutes for all services. The session’s cookie, that store’s the data in the client are also configured to allow its use only in conjunction with SSL connections.
Another issue that has not been fully resolved is the consumers’ access to the product. There are two options considered. Either the product need to implement an authentication strategy - like the module Passport [80], provides - the same strategy implemented for the platform’s own services. Or a product wrapper, that handles the product’s restricted access, need to be implemented. Such a wrapper could contain additional functionality such as recurring payments or public messages to consumers of the platform’s services, etc.

3.11 The showcase

To provide a visual aid and to better picture what was constructed during the development of the prototype and its services, a few selected photos has been prepared. All pictures can be found in the Appendices section.

The first photo, see Figure 14, shows the interface that manages the virtual machines. The current status of the machines is also displayed. Functionality to deploy new products for testing can be found in the menu to the right. The menu also includes the removal utility for domains, should this be required. Later, the functionality to add private keys will be added to aid the management of new services as they are taken into operation, among other features.

Next follows, see Figure 15, a view of a product and the data used mainly by the installer service to retrieve and deploy the product on a virtual machine, a client. The functionality to add new products can be found in the menu. The private SSH key is not the real key.

In the Order service, see Figure 16, one of the forms look and feel can be seen. All forms consists of the same design elements throughout the site, which allows changes to style and appearance an easy task. The layout is a grid consisting of 12 columns that can be distributed per choice. The grid makes it easy to create separations by setting one section to one column with empty content.

The interface for install, see Figure 17, includes the order and the information required to determine the status of the installation coupled with that order. The functionality to issue a redeployment of a product or the retransmission of the e-mail containing the details, for the use of the product, is here.

Lastly, see Figure 18, that shows the interface for the request logs for the services. Later, functionality to filter requests will be added here - to display all requests with a 404 (File not found) response, for instance. To filter the bad responses could help during the process of revealing bugs.
4 Results and evaluation

4.1 The effects of modularity

It might be a common trait among developers, the tendency to focus and spend a lot of energy, and thus resources, on small things that we can master rather than the grand scheme. Even thought small things such as which technology to use in order to promote readability or what mechanisms to use for the pull of database updates, are important, so must the general perspective always be maintained. That general perspective can easily be forgotten in a one man development project, the result being that some functionality is tuned and developed upon too much. However, seeing the cloud service as building blocks or modules helps to break down the general idea into parts, that can be more easily evaluated and has helped to keep the general structure intact and the development time for new functionality even, among the services in the platform. Since each part has its own tasks and uses its own resources and manipulates its own database records uniquely, the building blocks or services do provide compartmentalisation [81] and autonomy [82]. Autonomy since each service handles its own setup and adjustment of resources. Compartmentalisation since no service has all the functionality and all the data available at all times, only the exact resources needed to complete its task. Node.js modules, and the way that those modules are currently built and used, do inspire to structure projects as modules, by choosing to abide to such a structuring, as a result comes traits, such as:

- Compartmentalisation
- Autonomy
- Mobility
- Readability
- Scalability
- Easy refactoring
- Program logic separation
- Per service development
- No single point of failure
- Code duplicity
- Low complexity
- General distribution mechanism

Even though the functionality to spawn multiple install services to distribute the load - will not work - since the installer lacks a working load-balancing feature, this feature easily can
be added later when the load balancing feature has been completed. The ability to put less
important features, on hold, is refreshing. Further down in a finer grained perspective - if a fea-
ture, like the manipulation of a log entry object, is isolated properly, this feature can easily be
transported to other services that also need logging capability. By choosing to build most of the
features isolated the last service, the one in charge of installation, was built in under 15 hours,
mostly as a result of reused code. The isolation of features also results in a clean separation
of program logic and resource use, making the refactoring or substitution of any functionality
- simple. Some features have been changed to rely upon a different module several times. By
lifting in a new dependency, making the necessary changes and the corresponding substitution
for the other services, the functionality has been replaced. The feature isolation and the modu-
larisation also generally promotes readability throughout the platform.

By beginning with the build of a database, on a Droplet in the cloud, so had, all the services,
a need to be configured to use a non local database. Also, since the services themselves were
deployed into the same cloud, their own resources needed to be configured with regard to the
subdomain that the services was linked to. These two circumstances made each service highly
mobile. So whether one service was running in the cloud or locally on the home computer did
not really matter.

By modularising the platform into services, so could the services also be developed with a
minimum of Droplets used at all times, which did cut the cost for development considerably.
During the four months period, in which, the Droplets has been used, so has the total cost for
creation and continuous run of Droplets accumulated to $26.

Another result of building the cloud service as collection of services is that the platform
has no single point of failure (SPOF) [35]. If one service breaks, the overall functionality loss
is limited. However, since some features are common, such as logging, database manipulation
certain model data and access features etc., a certain level of code duplicity exists at the
moment. A future version of the platform will most likely have a common code repository from
which all the common code fragments can be shared.

Currently, no part of the cloud service prototype use any real complex constructions. Since
Node.js does promote non-blocking I/O (input/output) [60] as discussed in section 3.4 - The pro-
totype in action, most tasks are done incrementally. When an order has been placed in the Order
service, the Install service consumes the order and starts by setting up a new virtual machine,
pushes scripts, installs, starts and notifies consumer, so all parts of the job of setting up a new
client, is done individually, as responses to different asynchronous requests. The request flow is
linear but made concurrently for all install jobs. The linear request flow, and the fact that jobs
are continued upon by different services when possible, results in an overall lower complexity
level, all together.
Since the concept of modularity also in a functional sense was adopted from the very beginning of this project, the product, Entize Planning, also were seen and handled as a module in itself. A side effect of the product being handled as a modular part, is a platform that is general and capable of the distribution of other Node.js projects as well. Although time has not allowed for the complete set of planned features, the platform have the capability to distribute, install and start any Node.js based product that does conform to the general project structure discussed in the section 3.3 - *An Express project*, and that uses the local MongoDB database option. The platform also supports the installation of the key-value store Redis that is used frequently in Entize Planning, by Derby.

### 4.2 Same environment

Common problems during development is a lack of the proper environment. Although setting up resources locally, effectively reducing all latency introduced by the network and also utilising the superior resources of a modern development machine, in contrast to the Droplet’s limited resources, speeds up the development process, the application’s intended environment can not be neglected. By using Unix both for the development, Mac OS X, as well for the Droplets, Ubuntu 12.04, so has the use of native Unix resources, binaries, in all cases worked successfully. When pushing the service applications to Droplets for testing, there has been no lack of support. The only difference has been the network latency which is surprisingly low for the chosen DigitalOcean’s cluster. Comparing a single request done locally, in respect to the same request done, in the cloud environment, only introduced a 104 millisecond latency. Had the PaaS layer been the one provided by Heroku [20] so had there been a program available to simulate the cloud environment, the Dynamos environment, locally, as explained in section 3.1 - *The lay of the land*, which emphasises the importance of choosing an appropriate environment from the start.

Even though the DigitalOcean Droplet with the smallest size configuration provides limited resources, as presented in the section 3.1 - *The lay of the land*, so can these environmental resources be scaled up when needed, as a result of the easy scaling provided by the DigitalOcean framework. Initially, the lowest size configuration works with Entize Planning if the database is setup on a different Droplet. This is a result of the Entize Planning application eating up all of the 512 MB worth of RAM for its execution, leaving nothing behind for the database’s service. This behaviour can be configured so that the database’s service receives system resources by priority, but is most easily avoided by the relocation of the database to an individual Droplet. The database size after running the Entize Planning continuously for 9 months with an average of five machines in the production chain has accumulated roughly 800MB of data. The environment on the smallest Droplet can access 20 GB of storage which should suggest that the database’s Droplet should be able to run for a long time before any scaling upwards is necessary.
4.3 Time consumption and deployment

From the start of the development, one particular feature has been central. The ability to deploy a functioning product to the consumer as fast as possible. However, there are other concerns rather than speed. If speed was the most important requirement, each product that is launched from the platform would be using a prepared virtual image with the product pre-installed, only waiting for the configuration to be set. The prepared image, in conjunction with a Droplet already setup, waiting for a claiming consumer and a corresponding subdomain already created for that specific Droplet, would result in a complete delivery, in the same time, it took to send the email containing the account details, and the link to the product. But that solution would also result in a subdomain without the consumers’ branding, a Droplet that would have to be prepared again for any small change made to the product and a Droplet always preallocated and not allocated on demand. The current solution uses the product name chosen by the consumer in order to provide a product that feels like the consumers and also has an address that bears some meaning to the consumer. Apart from the subdomain, the installer currently fetch the product’s latest version from the Git repository associated with the product, which means that any new product always uses up to date code. Any update of a specific product could be done by registering a Unix cron [83] job that shutdown and pulls a new update during the night or any appropriate time decided by the consumer since all the resources are prepared. Any way this solution is viewed so is the creation of the subdomain, the main bottleneck, time wise.

When setting up the new subdomain, the newly registered subdomain have to be distributed among the name servers, resulting in a 10 minute delay before the newly deployed Droplet is reachable by the use of the subdomain name. Since the subdomain is linked to the IP address given to a Droplet during creation, the time spent on creating a Droplet adds to the total time for complete deployment. As an alternative, the IP address to a Droplet can be delivered as a substitute for the domain name, until the subdomain is properly distributed among the name servers. To get an accurate picture of the times involved in the entire process from order to welcome email, so has data been collected for 10 installations of a product that is of size 35 MB and has five dependencies (including Express):

- 0.01 min - Response to order
- 0.98 min - Droplet creation
- 0.01 min - Creation of scripts and transfer
- 0.31 min - Clone of product’s Git repository (relative)
- 0.28 min - Installation of product (relative)
- 0.15 min - Prepare and send email

These time entries are relative to the size of the product, and also to the number of dependencies that has to be solved during the installation of the product. By sending the IP address
to use until the domain name is ready to use, a small product can be deployed and used in under two minutes. There are some ways to streamline the deployment, and cut the total time even further, like having Droplets pre-allocated, but such a feature could add unwanted complexity, since the installers then need to keep a reference to the pool of those unbound Droplets.

To provide some measure of performance, tests have been done on the installer. The installer can without any problem run 20 concurrent installations, the top has not been measured. The average installation time for those 20 installations, from the first placed order to the last welcome e-mail, was 3.42 minutes. The difference between one installation and 20 concurrent installation is roughly two minutes, so each new Droplet adds another 6.2 seconds of processing time when there is 20 installations running concurrently. As reference, one test made with 10 installations which took 2.55 minutes to complete, where every new Droplet added 7.8 seconds per Droplet, only a bit more than the test involving 20 installations. This may be the result of the software in charge of sending welcome e-mails. This software sets up a connection pool and tries again and again to send the e-mails. The time until a successful transmission occurs varies greatly, and could be the reason for the inconsistency in the presented result. The overall deployment time varies depending on other factors as well, the time to setup the socket to the virtual machine, how long time it takes to get a response from the repository for the product, the overall network load etc.

4.4 Cost

By choosing the DigitalOcean PaaS layer and indirectly the IaaS layer, as discussed in section 3.2 - The PaaS layer, which provides cheap virtual machines to its consumers, the development cost has been kept low while the potential earning for a product delivered by the platform has increased. The total cost of the project so far is $40.7, and can be summarised by these posts:

- $26.0 - Continous run and test setup of Droplets from DigitalOcean
- $7.70 - Droplet costs for install tests ($0.07 per Droplet)
- $7.00 - 1 year cost for .se domain

This represents the small part of the overall costs, since the time spent on development is likely to be the biggest cost. Four month, 40% part time would sum up to roughly 270 hours. With a salary of $4,600, the development cost would be $7,360. With social fees, which is 31% of the gross salary in Sweden for year 2014, $2,282, and 5% insurance and pension fees, $368, so would the total sum for one developer be $10,010 (ca. 65000 SEK). There is also one more cost that has not been included yet, the cost for maintaining a wildcard certificate for the entire domain. There is a cheap, probably not the best, PositiveSSL wildcard certificate that can be issued for $96 per year. Estimating further development, that might be needed for the prototype to go into product, to half the time already spent, so would the total cost from idea to production
be $15,050.70 plus an annual cost of $103 for maintaining the .se domain and the certificate. There is also the annual cost of running the platform’s services on Droplets. Having a MongoDB with a replica set configuration and backup enabled would include:

- 3 DB Droplets
- 1 Maintenance Droplet (could be spawn on demand)
- 1 Logs Droplet (could be spawn on demand)
- 1 Order Droplet
- 1 Install Droplet

Calculated with 7 Droplets with a $5 cost per Droplet, per month, and three Droplets with enabled backup, so would this configuration add an additional annual cost of $456 per year, resulting in a total annual cost for maintaining the platform to $559. Of course there are bound to be issues that needs support of some kind, but such a cost is pure speculation at the moment and can only be evaluated after the platform has been taken into production. Marketing of the platform will also add to the cost at some point.

Currently, the platform process orders but can not charge the consumer. The charging will require some payment service to be bound to an account, like Paypal [84] is. Opening a business account for online transactions is free, however a percentage fee is taken for each transaction based on the amount transferred. Basically, if Paypal is used, there would be no additional cost involved in charging the consumers.

4.5 Has the goal been satisfied

Before the start of the prototype’s development, see the section 2.7 - The prototype, a number of goals was setup for the final prototype as support for the evaluation phase. Lets review each, and see if the majority of them can be considered as done. Here follows the compiled list again:

- Choose what service mode to provide the consumer
- Choose when, what type, and how much hardware and software resources to utilise
- Sufficiently stable open source software might be preferable in order to cut cost
- Security technologies must be able to protect non-local data integrity and transmission
- Upgrade and capacity changes must be transparent to consumer
- Cloud services must be sentient to some extent and be able to recover from bad states
• If possible, avoid unnecessary complexity

• Consumers may need to be able to create new templates on demand

• Must support personalised protected resources for each consumer

• Automatic deployment for new consumers

• Support for initial data (settings) retrieval for new consumers

• Software should acts as a connecting layer between cloud environment and application to separate the application logic from the cloud architecture logic

In sections 2.2 - Cloud Service and 3.1 - The lay of the land, the various service levels are discussed and one specific level are chosen. It was decided that the service level to rely upon is a PaaS service and also that the prototype’s platform delivers the level Software as a Service (SaaS) for its consumers. So the first goal can be declared as done.

The recommended size for each product is preset but can be optionally increased by the consumer. Anytime in the future this option will most likely disappear from a consumer’s stand-point, and the size will be regulated automatically depending on the need. Currently, the support personnel are the only ones that can change the size of a consumer’s Droplet. Each product can use different software packages as a base for the Droplet, as discussed in section 3.2 - The PaaS layer, so can the image be prepared with software to support the product. This image option is preset - by the support personnel - when a new product is added to the platform. Thus, hardware resources and software resources can be regulated, making the second goal also fulfilled.

The entire platform is built using open source resources, something which has resulted in no software costs for the project so far. Since most of Node.js collection of modules has a free open source counterpart to proprietary equivalents, and many of the non proprietary modules’ have widespread use and are well documented, reliable products can be built without spending resources on software. Some special features, such as an invoice-printer, might not be free since they might use additional services to support the functionality. As for the third goal, to cut cost, it is safe to say that this goal has been fulfilled as well.

In section 3.10 - Security concerns, the security aspects have been covered. SSL with a self-signed certificate is currently employed throughout all services. The certificate will be replaced with a wildcard certificate in time for production, since the self-signed will produce a warning to the user, issued by the browser, which generally lowers the trustworthiness of the site. The certificate added to the product, via the applied configuration that is pushed to the product during the installation phase, might be one of those cases that transgresses the boundary, between product and cloud environment. This security feature is however required. To ensure that traffic is actually encrypted, tools can be used to inspect the traffic, Wireshark [85] is one example of such a software. All SSH communication uses pre-distributed keys and the database
connections uses authentication. All the passwords are one-way encrypted, and all the sessions and form data are also encrypted. The products are retrieved while using SSH keys distributed in a secure manner. The only topic missing is the use of CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) [86], for the order service to prevent bots, other computers, from loading the server by repeatedly posting to the server by using a form present in the web page. The payment module does often include CAPTCHA and can be placed before and form that posts to the server to achieve the same effect. A security technology that the platform lacks for the moment is the use of HTTP Strict Transport Security (HSTS) [87], which allows for the server to require that all traffic is made while using SSL. HSTS protects against SSL-stripping [88]. Apart from HSTS, a couple of headers remain to be set - to prevent Cross-site request forgery (CSRF) [89], where an attacker produces a web page that looks the same, or a manipulated link that is trusted. Another header that will be set is a header to prevent Cross-site scripting (XSS) [90], which is when an attacker plants malicious Javascripts. These last mentioned security features can be applied using a Node.js middleware called Helmet [91] which will be applied before the platform goes into production. All in all, when the last security features has been applied, this requirement have been fulfilled and can be seen as completed.

The upgrade and capacity changes for the product, and the resources of the Droplet are features that are still not implemented. They can be done transparently, without bothering the consumer, but a solution with using the Unix tool cron [83] has been proposed in section 4.3 - Time consumption and deployment. The cron job cloud pull new product updates from the online repository, and also determine available resources and request additional resources if necessary. Even though these actions can be performed, both activities require manual attention. Still, this functionality is highly prioritised. Setting up the cron job is an easy task which makes the completion concerning this goal - close at hand.

The cloud services has the capability to restart themselves upon service interruptions and also when the Droplet restarts. The only functionality that is sensitive to service interruption at the moment is the installation of software on a new Droplet. All the tools for a redeployment on error exists, but have not been taken into use yet. The re-deployment can be done manually, and although it would be satisfactory to have this feature automatic, this goal has been met.

Regarding the goal to avoid unnecessary complexity, a list of strategies was compiled to combat complexity, see sections 2.5 - Aspects, 3.8 - Maintenance and 4.1 - The effects of modularity. One key point was modularity. By separating the different tasks into services, and using the database as a nerve centre, all connections between individual services have been avoided, and the services can drop due to errors and restart without affecting the other services. The overall complexity is also reduced in this way since no service requires communication with the other services. None of the services or their internal features are timing sensitive since they only respond to events. Basically, this means that new data is available or exists for consumption. Another strategy to combat complexity was to reuse code whenever possible. Since the use of the exports function allows for functionality to be broken out and imported, common features
has been broken out and reused. Since this introduces duplication of code, it will most likely be augmented with the use of a common code repository later. Currently, no code blocks or interactions within the platform exhibits any complex traits, which was the goal.

The ability to create templates was one of the functionalities that might have been attractive for the Entize Planning product. But as the development has progressed, the distinct boundary between the cloud service’s platform and the product has become more clear. The functionality to use templates in the Entize Planning application to model production steps belongs to the product and as such should be provided by the Entize Planning itself. The goal to provide consumers with templates is therefore shifted away - from the cloud platform’s responsibilities.

The current version of the platform can store and handle personalised data for each consumer, but the only use for such data currently is for the product wrapper, that is planned to handle restricted access for the use of the product’s application, as discussed in section 3.10 - Security concerns. Currently so are the product deployed and can be used without restriction. All the details concerning this wrapper are not decided yet, but it is conceivable that the wrapper could provide some additional base functionality that can be used by the consumer. Such functionality could include public messages to the consumers or consumer feedback. This goal is partially fulfilled.

The next goal was to provide a service that is automatically deployed. That is no human interaction should be needed for the creation and deployment of a new product to a consumer. After an order has been placed, all the succeeding steps are automatically carried out and ends in an email being sent containing the details for the use of the product. Except for the planned wrapper, everything work automatically, and thus also this goal is satisfied.

When creating an order for a product so are the required data for the product collected and stored in the order until the order has been handled. So this goal has also been fulfilled.

The last goal mentions a layer between the product’s environment and the product itself. This layer is the same wrapper that were discussed in one of the previous goals. This wrapper has only been partially realised.

Overall, most of the goals has been met. Since the main objective was to produce a prototype, and not a completed and finalised product, it’s expected that some of the goals has not been fully realised. However, all in all so are the main problems solved and the work left consists mostly of filling in the blanks.

4.6 User interface capabilities

All services in the cloud service prototype expose its functionalities via a user interface with restricted access. The maintenance service Nexus, see Figure 13 for an overview of the services, plays a special role. Since the service in charge of the maintenance, needs to be able to
support the entire platform, it has more capabilities including install capabilities and more. By utilising the Nexus interface, the support personnel is able to add, remove and change any state of the Droplets or clients. The Nexus interface also exposes functionality to handle the products that are offered in the Order service. Nexus provides the administrators with the tools to add, remove and change state of the products. There are options for the support personnel that can be used - to add products without displaying the products, to the Order service. Thus, a new product can be tested safely within the environment and easily deployed when the tests have been concluded. The subdomains within the DigitalOcean cluster can also be maintained via the Nexus Web service. To handle the users for all the restricted services within the cloud, the Nexus application has the capability to add and remove users. Database tools will most likely appear to support the Nexus service in the future.

The order service exposes the only public page in the platform. Via the interface presented by Order, the products are showcased and promoted. The ability to place orders that will be consumed by the installer also resides here. In the final product, the protected part of this service will be handling display, add, remove, costs, payments, accumulated income and refunding of orders. To add, remove and display orders is currently implemented.

When orders appear in the database, the Install service process the order and setup of the product. Currently, the deployment is the only capability that the installer has. However, in the near future, other features are to be implemented including statistics, such as deployment time, total number of deployments and current load etc. - statistics that can be used to determine if an addition of another installer service is necessary. An expansion of the use of statistics could also be a system for notifying when an additional installer is needed and perhaps an automated deployment.

The last service currently deployed in the platform is the Logs service which presents all the logs for all the services in the platform. The only currently logged information is the requests to all the services. Later on, it will be possible to filter out certain traffic from the logged information. This functionality can be useful to track down any errors that might have happened, and to help service consumers with complaints. Later functionality might be available to include resources for detection of some common attack-signatures.

Lastly, a Monitor service is planned. A Monitor service in charge of common usage statistics such as health, uptime, load, available resources and consumed traffic across the platform. This service most likely include the use of charts to present overall statistics in a more human readable form. A Node.js module with some interesting functionality already exists, that might be nice to extend upon, called monitor [92].

The interfaces exposed to the users of the platform’s services reuses the same components, styles and libraries. By spending time in the beginning of the project on building up a component stack of basic components with attached features, new functionality is easily added to the
backend [93]. Basic components such as lists, buttons, menus, overlays, notifications, layouts, separators, texts and input elements can be acquired from online resources. When working with backend functionality that needs front-end support [93], the need to stop the backend development in favour of building front-end components can be interruptive, to copy paste some code and continue, can in view of this, be desirable. The components do not need to be very pretty, since style and cosmetics can be promoted later. At some stage the need to support many languages will be needed, internationalisation [94]. To be able to choose language or make the application process the consumers’ browsers locale [95] setting to provide services in a consumable way, is common in modern Web services. This interface support will be added to the Order service before the project goes into production.
5 Conclusion

When taking a step back and reviewing the assignment relevance, its execution, choices made and the possibility for improvements, a few topics do come to mind.

5.1 In time

The concept of secure storage and processing of private data, while using hired hardware, online, has grown in recent years. Being able to change allocated resources instantly and at the same time avoid the employment of the necessary experts for hosting, makes the market lucrative for software development companies. A documentation of the process, to evolve or transform a Web service to be delivered, by the cloud, is therefore relevant.

5.2 Framework and external services

Although, the Node.js framework supports the development of custom-made applications for the web and also provides the freedom to choose which software to use, so could some functionality be included optionally from start, to lead the way for new web developers. The freedom comes at the expense of that a higher understanding and knowledge, of how to develop secure, reliable, scalable and fast Web services, is needed. Other full stack solutions include some basic functionality, such as session handling and internationalisation, which reduces the workload and emphasises the need. The framework has support for pre-bundled software packages (DigitalOcean, Heroku). To avoid problems with, and to keep down the development, so can an initial investigation into what kind of support the cloud services, relied upon, possesses, be rewarding. For example, DigitalOcean requires communication with its support in order to lift, the 10 Droplet limit, placed on new accounts - what if - the limit could not have been adjusted? Such limitations should be known from the start.

5.3 Prototype for cloud services

In the cloud environment, security is a requirement, and its importance can not be emphasised enough. Therefore encrypted transactions, restricted and authenticated access and secure storage are crucial. Still, the prototype has a few vulnerabilities that need to be patched before production.

By separating the prototype’s main tasks, and building standalone applications for the tasks, the overall complexity has been kept low. The separation of tasks has also resulted in highly mobile and scalable implementation. The prototype’s main tasks do not need to compete for the same system resources. The drawback of building a cluster of services is that all common functionality essentially results in code duplicity and requires that functionality is built modular and distributed to avoid multiple version of the same functionality.
Even though the prototype is not finished, the functionality to deploy products works, and the major goals have been fulfilled. The fact that the prototype is built on Node.js and does not require hardware maintenance makes the product fulfil the requirements from the company in charge of Entize Planning. The platform is more general than was initially expected and is the result of keeping a distinct separation between the product, and the cloud platform. The platform can be used to deploy its own parts, and will therefore be used for horizontal scaling.

The cloud service still needs to be expanded to support cluster configurations for products. Since the initial goal was to launch a product into the cloud with its resources located locally on the same virtual machine, so is this support an extension of that goal.

5.4 Generally

The development has evolved naturally and many problems have been avoided through the use of a good design. Another effect of design choices made during the development is that the platform is not restricted to distribute only Node.js projects. The process of adding products to the platform gives the developer the tools to allow deployment of other software as well. The drawback of treating the product as an independent part of the platform is that all functionality must be generic and generic functionality often comes with a higher price tag, due to the additional development that is often required. The platform has the potential to evolve into a full scale product delivery platform in the future.

Apart from this thesis the project’s code base lack some documentation. It might be worth noting that the platform is restricted in some respect to DigitalOcean’s Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).
6 References


62
Figure 14: The Nexus interface for the clients (virtual machines)
Figure 15: The Nexus interface for the products
Figure 16: The Order interface for the ordering
Figure 17: The Install interface for managing installations
Figure 18: The Logs interface for displaying service logs

<table>
<thead>
<tr>
<th>URL</th>
<th>Status</th>
<th>Time</th>
<th>Date</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>/images/logo.png</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/public/images/logo.png</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/fonts/reconciliation.ttf</td>
<td>200</td>
<td>3ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/images/download.gif</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/images/logo.png</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/jquery.min.js</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/bootstrap/js/bootstrap.min.js</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/index.html</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/magnetic.js</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/reaped.js</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/bootstrap/js/bootstrap.min.js</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/stylesheets/style.css</td>
<td>200</td>
<td>17ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>9.95KB</td>
</tr>
<tr>
<td>/bootstrap/css/bootstrap-lm-large.css</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/bootstrap/css/bootstrap.css</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/logic</td>
<td>200</td>
<td>49ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>2.16KB</td>
</tr>
<tr>
<td>/</td>
<td>200</td>
<td>14ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>6KB</td>
</tr>
<tr>
<td>/fonts/reconciliation.ttf</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/download.js</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/images/logo.png</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/bootstrap/js/bootstrap.min.js</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/index.html</td>
<td>200</td>
<td>2ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/magnetic.js</td>
<td>200</td>
<td>1ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/javascripts/reaped.js</td>
<td>200</td>
<td>3ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
<tr>
<td>/stylesheets/style.css</td>
<td>200</td>
<td>8ms</td>
<td>Thu, Apr 25 2014</td>
<td>20:23:20 UTC</td>
<td>1.34KB</td>
</tr>
</tbody>
</table>