VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Anass Shekhamis

Multi-Instance Quotation System (SaaS) Based on Docker Containerizing Platform

Technology and Communication
2016

FOREWORD

This is the final paper of my thesis at Vaasa University of Applied Science, Vaasan Ammattikorkeakoulu, in Information Technology Degree Programme.

I want to thank my supervisor, Jukka Matila, for his guidance and help that he provided to accomplish this thesis. His professional notes, knowledge, and mentoring methods have assisted writing this thesis and delivering high quality end results.

I also want to express my appreciation and thanks to the sponsor of this thesis work, Pool Interactive Oy, for providing this opportunity, and to the teachers and staffs who have helped me and provided their knowledge during my study period.

Finally, I would extend my gratitude to my family and friends who supported and up-held me through every step.

Anass Shekhamis
Vaasa, Finland
05.04.2017

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Programme of Information Technology

ABSTRACT

Author	Anass Shekhamis
Title	Multi-Instance Quotation System (SaaS) Based on Docker Containerizing Platform.
Year	2016
Language	English
Pages	133
Name of Supervisor	Jukka Matila

This thesis covers the development of a quotation system that is built as a multi-instance SaaS. Quotation systems usually come as part in customer relationship management systems, but not necessarily included. They also tend to have invoicing alongside the original functionality; creating quotations for customers.

The system uses Microservices Architecture were each service is a replaceable and upgradeable component that achieve certain functionality and easily integrate with other third-party applications such as invoicing and team management systems through their API.

The thesis discusses the purpose of building such system which is directed specifically towards in-house construction and maintenance companies. Then, the difference between multi-tenant and multi-instance when building SaaS in the cloud and the practice of building a REST API that could be easily integrated with external web service and third-party software and APIs. After that, the web security principles and software containerization concepts were introduced. Next, the process of collecting the system requirements, analyzing them, and design the solution supported by the UML diagrams and the system architecture description. Then, the solution structure and the technology stack were introduced followed by the system implementation of the back-end as API and the front-end web application as a single-page application that consumes that API. Furthermore, the implementation of authorization access on the user and the API level was discussed. Finally, the deployment using Docker and Kubernetes was explained briefly.

It can be concluded that the implemented has a well-designed architecture, and met the expectation of the current customers and the sponsor. Moreover, the system could be improved and optimized, and more features could be added.

Keywords SaaS, Quotation System, Multi-Instance, Containerizing, Clustering

CONTENTS

LIST OF APPRECIATIONS

SaaS	Software as a Service
IT	Information Technology
REST	Representational State Transfer
API	Application Programming Interface
SPA	Single Page Application
OWASP	Open Web Application Security Project
DBMS	Database Management System
CRM	Customer Relationship Management
MIS	Management Information System
ERP	Enterprise Resource Planning
HRM	Human Resource Management
HTTP	Hypertext Transfer Protocol
CORBA	Common Object Request Broker Architecture
RPC	Remote Procedure Call
WSDL	Web Service Definition Language
SOAP	Simple Object Access Protocol
CRUD	Create, Read, Update, and Delete
SPA	Single Page Application
SPI	Single Page Interface
UI	User Interface
XSS	Cross-Site Scripting

CSRF	Cross-Site Request Forgery
LXC	LinuX Containers
DBA	Database Administrator
DevOps	Software Development and Information Technology Operations
UML	Unified Modelling Language
ACID	Atomicity, Consistency, Isolation, Durability
RDBMS	Relational Database Management System
LAMP	Linux, Apache, MySQL, Perl/PHP/Python
MVC	Model-View-Controller
ORM	Object-Relational Mapping
CLI	Command-Line Interface
SSH	Secure Shell
IoC	Inversion of Control
JWT	JSON Web Token
CORS	Cross-Origin Resource Sharing
SWT	Simple Web Token
SAML	Security Assertion Markup Language Token
MVVM	Model-View-ViewModel
IIFE	Immediately Invoked Function Expression
MVVM	Model-View-ViewModel
PHP-FIG	PHP Framework Interop Group
npm	node package manager

UX	User Experience
GPG	GNU Privacy Guard

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 **Purpose**

This thesis covers building a Software as a Service (SaaS) driven system that is intended for companies working in in-house constructions and maintenance field. The application automates the process of creating quotes for the company's customers, taking the measurements and the dimensions of a located area, estimate the required material for each maintenance or construction service, calculate the final expenses of that quote, and finally, invoice the customer.

Since the whole system has been built as a multi-instance SaaS, and each business will have its own application, the need to manage these sets of applications has been introduced. Thus, a clustering application has been created for that reason. This clustering application will take care of registering a new company, automating the process of creating a new application for the registered company, and provide an access point to that request through a subdomain.

1.2 **Overall structure of the thesis**

The main ten chapters of this thesis provide a broad understanding of how the system is built, the chosen software architecture and the reason behind it, and the technology stack. Chapter one provides an introduction and overall view of the scheme. Chapter two covers the theoretical background and some concepts like REST APIs and SPAs, the solution structure and the relevant technologies including but not limited to data persistence, and third-party APIs integration. Chapter three covers some relevant concepts about information security, and software containerizing. Chapter four and five discuss the requirements, system analysis, system architecture, and the used technology stack. Chapter six and seven provide detailed information about parts of the system implementation at the API and the web application level. Chapter eight shows how the security parts are implemented in the code and the communication level. Chapter nine is about using Docker containerizing platform to structure the application for deployment and managing containers with orchestration tools. Chapter ten wraps up and summarizes the content of the thesis.

1.3 **Background**

Pool Interactive, which is the sponsor of this thesis work, saw the necessity of such a system and the amount of time and effort it will save if implemented. The requirements and the implementation are being placed according to lengthy discussions and meetings to reach the gratification the company craving.

2 SOFTWARE DESIGN AND ARCHITECTURE

2.1 **Quotation system**

A quotation is an agreement that the provider/supplier will deliver or offer to the buyer/consumer which consists of the services at a stated price, under specified conditions. Those quotations are documents/forms that are used to inform the consumer how much the service will cost before committing to it. They also tend to commit the provider to a specific fee which is being calculated by providing parameters such as labor, time, raw materials and extra costs. /1/

The outcome serves as a contract containing the cost of the total service in details, and a scheduled time for a delivery that both parties agree upon. Any change, later on, will affect that outcome, and the customer should be informed beforehand. A quotation could contain payment and invoice. /1/

According to the definition above, an idea for a quotation system that aids to automate the process of creating a quote for in-house construction and maintenance businesses was born.

2.2 **Software as a Service**

Software as a Service is a software delivery model in which the software is accredited on a subscription basis and is centrally hosted. It became a common delivery model for many business applications like management information system (MIS), enterprise resource planning (ERP), customer relationship management (CRM), database management system (DBMS) software, and human resource management (HRM). SaaS, usually is browser-based web application and accessed through the internet. /2/

2.3 **Cloud Architecture**

SaaS service model is one of the four foremost categories of cloud computing, along with Data as a Service (DaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Cloud architecture requires components to form the cloud computing. These components are consist of front-end platforms (fat client, thin client, or mobile device), back-end platforms (services, and storage), a cloud-based delivery, and a network (internet). /3/

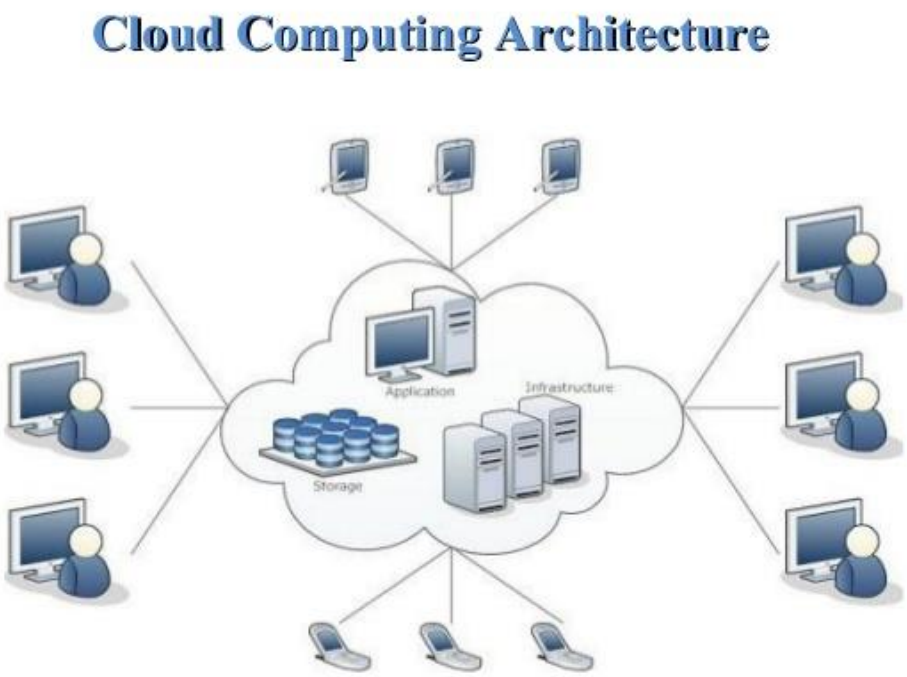


Figure 1. Cloud computing architecture. /7/

2.4 **Multi-instance VS Multi-tenant**

SaaSs are being built by IT upon four architectures mainly; those are multi-instance, multi-tenant, single instance, and flex tenancy. The result will not differ to the end user much, but it will differ regarding the architecture of the system, data and its access, the configuration, and user management. This thesis will focus on multi-tenant and multi-instance, and in their simplest form, they both have a tendency to solve the same problem, however, multitenancy adverse to multi-instance.

In a multi-instance architecture, multiple companies will run their own separate instance of the application, with their own separate database. Also, it could run on a different operating system (OS), and a different hardware platform, so it is very flexible. Each business will have access to its data separately from the other, which means that each company will have its users, customers, services, materials, categories, packages, templates, and quotes. /4/

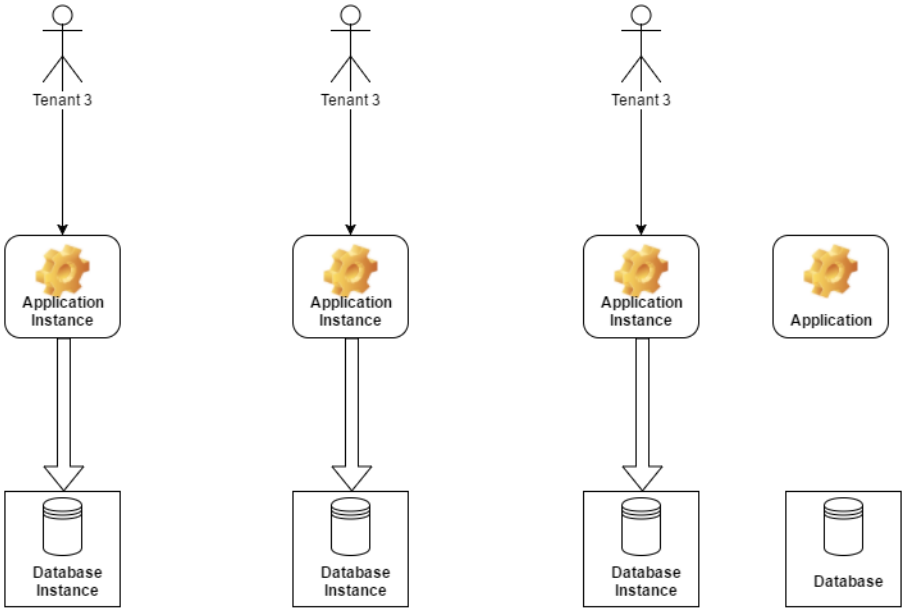


Figure 2. Multi-instance architecture

However, in a multi-tenant architecture, multiple companies will be using a single instance of the application, with a single database, running on the same hardware and same OS. This architecture does not give much flexibility but simplifies the process of adding features and fixing code bugs. /4/

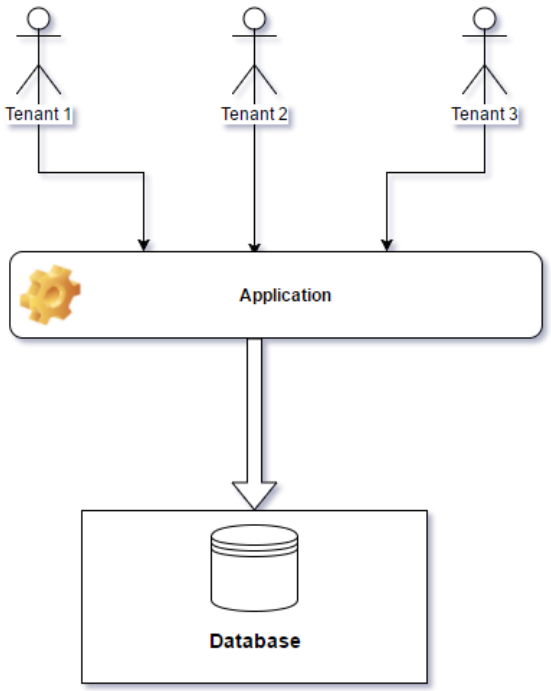


Figure 3. Multi-tenant architecture

Many argued that the multi-instance be better than multi-tenant approach when it comes to cloud architecture, and according to many, this is how the enterprises run their mission-critical applications. /5/ /6/

The following are two lists of the advantages and disadvantages of multi-tenant approach.

Pros:

- Cost effective. Using the same infrastructure and resources.
- One shared cloud. The cloud provider sustains for all users.
- Time effective. It requires less time and resources for updates/upgrades a large number of users at a given instance.
- Always on the latest version. Changes take place in the whole environment and happen once for the system/application.

Cons:

- Shared database. Any action that affects the multi-tenant database will affect all shared customers.
- Preventing exposure of data happens in the application layer. The developer team will be responsible for preventing data exposure from one client to another, which will sufficiently increase in terms of complexity.
- Any security breach has a massive effect as it will affect every tenant in the system.
- Not customizable. Tenants cannot customize the application to fit all their requirements.

Now, for multi-instance, here are the two lists.

Pros:

- Data isolation. Each consumer has its database and infrastructure.
- Great flexibility and control of configuration and customization.
- High availability. If one instance is down due to infrastructure issues, it will not affect others.
- High scalability. Let it be in the case of an individual server, virtual machine, or container per consumer; it is always easy to add more resources including, but no limited to, memory, CPU, or cache.
- Ability to move consumers individually.

- Highly customizable. Updates and upgrades can be performed on individual customer instances where it fits the requirements and the needs of the user.

Cons:

- It is harder to deploy changes to multiple instances.
- Not cost effective when it comes to creating and configuring the environment such as the database or the application.

2.5 **RESTful APIs**

RESTful or, or shortly REST, stands for Representational Stateless Transfer, is a web service and a hybrid architecture style evolved from various network-based architectural styles, incorporated with restraints which define a uniform connector interface for designing networked applications which rely on stateless, client-server cacheable communications protocol (HTTP). It is a way to provide interoperability between computer systems on the internet. The idea behind it is that rather than using complex mechanisms such as CORBA, RPC, SOAP or WSDL, simple HTTP requests to create, update, delete, or read (CRUD) data are used to make calls between machines. /8/

REST relies on HTTP verbs and statuses to form the communication with the API. Defining routes that a machine can make calls to, and either request or manipulate the data, in case it has the authorization to perform that.

The following is a general example of a REST API, where resource could be any factual resource i.e. users, products, or similar.

URL	HTTP verb	HTTP body	Response
/api/resource	GET or HEAD	Empty	List of resource
/api/resource/:id	GET or HEAD	Empty	Single resource by its id
/api/resource	POST	JSON, XML or String	Create new resource and return a reference to it

/api/resource/:id	PUT	JSON, XML, or String	Update a specific resource by its id
/api/resource/:id	DELETE	Empty	Delete an existing resource by its id

Table 1. REST API general example

2.6 **Single Page Application (SPA)**

Single Page Application, also known as Single Page Interface (SPI), is a web application that fits on single web page providing a native-like user experience as if using a native mobile or desktop application. Usually, what is seen on the browser while viewing a web page is a combination HTML, CSS, and JavaScript, being loaded from the server. While requesting another web page from the same domain in a traditional technique, it will end up in a situation where the browser will have to reload those CSS and JavaScript files again, unless the browser cached them. SPA concept will load those files once, and make calls to a thin, thick stateful, or thick stateless server architecture loading the data from an API; using AJAX. /9/

The most notable characteristic of the single page applications is the ability to redraw single part of the user interface without making a server round trip to retrieve HTML. Multiple JavaScript implemented this idea, both partially and comprehensively, including but not limited to AngularJS and Ember.js. /10/ /11/

SPAs faced some issues in the early stage concerning search engine optimization, which was due to the absence of JavaScript execution on crawlers of few search engines. A work around this issue was that the application would handle rendering the first page on the server, and updates the subsequent on the client, which was challenging since rendering code needs to be written in a different language on the server and the client. Some JavaScript full-stack frameworks did solve this problem where the same JavaScript code runs on the server and the client. An example of such framework was Meteor.js. /12/

Google search engine crawls URLs containing fragments starting with "#!" which is called hash bang. However, SPA site needs to implement some behavior to allow extraction or relevant metadata by search engine's crawlers.

The following figure shows how a modern single page application is structured.

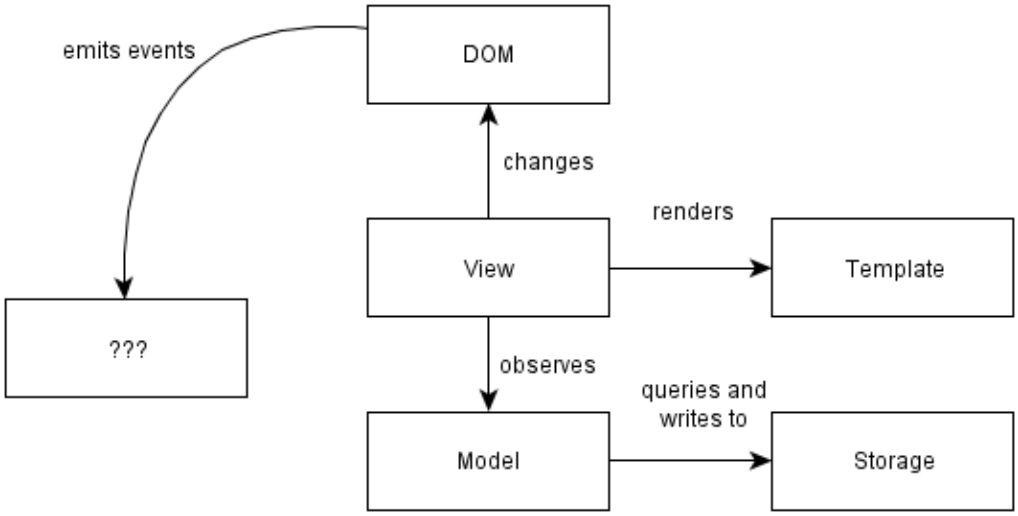


Figure 4. Modern single page applications structure. /13/

2.7 External Web Services and Third-Party APIs Integration

Building large systems requires time that could be saved if the same functionality could be achieved by using components from other applications and web services. Such services could be only API-centric or standalone applications with API built around them to provide an ability to interact with them. This popular approach has an enormous impact on saving development time and reusing systems.

The same concept applies to a system that is being built with an idea of exposing functionality to external systems and applications. However, such process is not easy and requires following best practices even for enterprises.

2.8 Web Application Performance

In general, performance could be defined as the achievement of a specified duty measured against predetermined known standards of precision, completeness, cost, and speediness, and is based on perception. Web application performance does agree with the previous definition. However, good performance is a relative term where it varies depending on the type of the application and the usage, and what seems to be good enough today might not be in the future. /19/

Typically, optimizing web application performance goes through three steps: measuring, diagnosing, and fixing issues. Developers most of the time tend to overdo performance fixes and sacrifice usability. Rich web applications tend to have a higher threshold of page load traded for reach user experience.

3 WEB SECURITY AND SOFTWARE CONTAINNARIZING

3.1 Information Security

Information security, also shortened InfoSec, is a set of strategies, methodologies, and practices for managing the tools, policies, and processes to prevent, detect, and document unauthorized access, use, modification, inspection, and disruption of digital, physical or any other form of confidential, sensitive, and private information. /14/ /15/

Confidentiality, integrity, and availability together form the core principles and the key concepts of information security. Confidentiality is a set of rules promise that limits and restricts access to particular types of information to unauthorized entities and processes. Integrity is the assurance of accuracy and inclusiveness that the information is accessible over its entire life-cycle. Some practices would be network management procedures to guarantee data integrity, including monitoring authorization levels for all users, recording system administration actions, parameters, and maintenance activities, and creating disaster recovery plans for occurrences such as power outages, server failure, and virus attacks. Availability means that data should be available whenever it is needed.

3.1.1 Defense in Depth

Defense in depth, also known as “Castle Approach,” is a multi-layer of security controls in an information technology system that assures its protection from the initial creation to the final disposal. This information assurance concept can be divided into three zones: physical, technical, and administrative.

Physical controls are the physical limits that prevent access to the IT system like guards, fences, CCTV systems. Technical controls are the hardware and the software that protects the system and its resources like disk encryption and fingerprints readers. Administrative controls are the policies and procedures that ensure there is a proper guidance available to security.

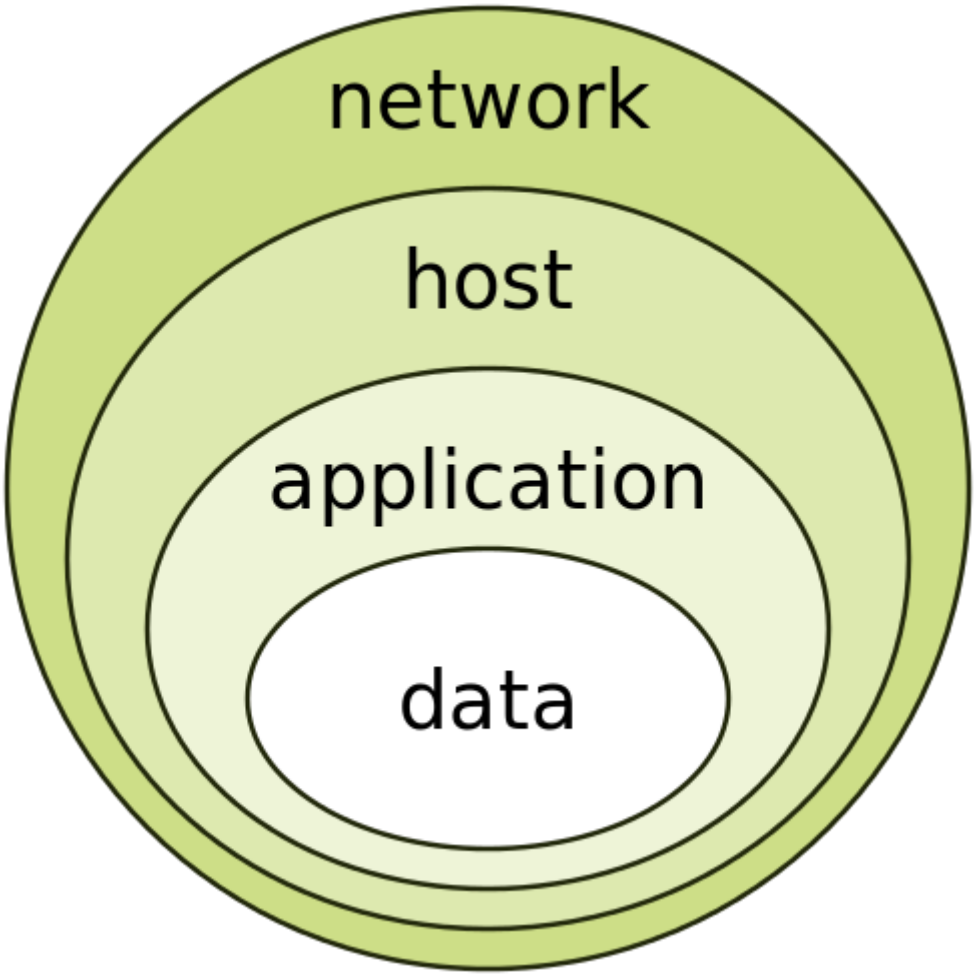


Figure 5. Defense in Depth: Onion Model /16/

3.1.2 Web Application Security and OWASP

Web application security is part of the information security that takes care of the security of web applications, websites, and web services.

OWASP stands for Open Web Application Security Project, which is a foundation that came online on 2001, to become a non-for-profit charitable organization in 2004 to guarantee the continuing obtainability and support for its work. The core values of OWASP are open, innovation, global, and integrity, which ensure transparency, and honesty to anyone around the world. /17/

The most recognizable work from OWASP is the OWASP Top 10 project, which is an important awareness document that provides a list of the ten most critical web application security risks. /18/

According to the OWASP Top 10 for 2013, the top security risks are in the following order.

1. Injection: Injection flaws, such as SQL injection happens when untrusted data is directed to an interpreter as part of a command or query. The invader's contentious data can trick the interpreter into executing unintended commands or retrieving data without authorization.
2. Broken Authentication and Session Management: application authentication and session management are not implemented properly, allowing attackers to compromise passwords, keys, or session tokens.
3. Cross-Site Scripting (XSS): XSS flaws arise when an application receives untrusted data and sends it to a web browser without validation or escaping, allowing attackers to execute scripts in the target's browser hijacking user sessions, or redirecting the user to malicious sites.
4. Insecure Direct Object References: direct object reference arises when a developer leaks a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
5. Security Misconfiguration: good security entails having a secure configuration defined and deployed for the application, application server, web server, and database server. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Furthermore, software should be kept up to date.
6. Sensitive Data Exposure: many web applications do not correctly protect sensitive data, such as credit cards, and authentication credentials. Invaders may snip or alter such unprotected data to conduct credit card fraud or identity theft.
7. Missing Function Level Access Control: Usually, applications authenticate function level access rights before allowing that functionality visible in the UI. Applications have to accomplish the same access control checks on the server before each function is accessed. If requests are not confirmed, attackers will be able to forge requests to access functionality without authorization.
8. Cross-Site Request Forgery (CSRF): such attack forces a victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

9. Using Components with Known Vulnerabilities: such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is demoralised, such an outbreak can ease serious data loss or server occupation.
10. Unvalidated Redirects and Forwards: web applications normally send and redirect users to other websites, and use untrusted data to regulate the destination pages. Lacking such validation, attackers can send victims to phishing or malware sites or use forwards to access unauthorized pages.

Applying security techniques to prevent such vulnerability should be taking place in the earlier stages of development and continue throughout the software development lifecycle, and never been left out until the end.

3.1.3 Access Control

In simple form, access control is the guarantee that accessing information must be restricted to the users who are authorized to have such privileges. Access control consists of three steps: Identification, Authentication, and Authorization.

- Identification: Who someone or something is. In software development, the claim could be in the form of username or email.
- Authentication: Verifying the previous identity claim. Before granting access to a user that is claiming an identity, verifying the user must be performed like checking ID or fingerprints. In software development, this could be a PIN or password.
- Authorization: the process of which a machine or a user are being determined the resources that they are allowed to access and the actions they can perform. Usually, authorization has two forms in software development: role- and permission-based access control.

3.2 Software Containerizing

Software containerizing is an alternative to machine virtualization which involves encapsulating an application with its operating environment in a container that could run on any suitable physical machine like computers, virtual machines, bare-metal servers, and cloud clusters, without taking care of the dependencies of that application. /20/

Docker is a lightweight containerizing solution that runs on the same OS kernel. Its standards enable containers to run on all the main Linux distributions and recently on Microsoft Windows. It is secure by default as the containers isolate the applications from one another while providing an additional layer of protection for the application. Docker is currently the world's leading software containerizing platform.

The major features of Docker are that accelerates developers work where they can take copies of their live environment and run them on any new endpoint running a Docker engine, eliminating environment inconsistencies, and include only the application with its dependencies; sharing the kernel with other containers while virtual machines include the application and all the binaries, libraries, and the entire guest operating system.

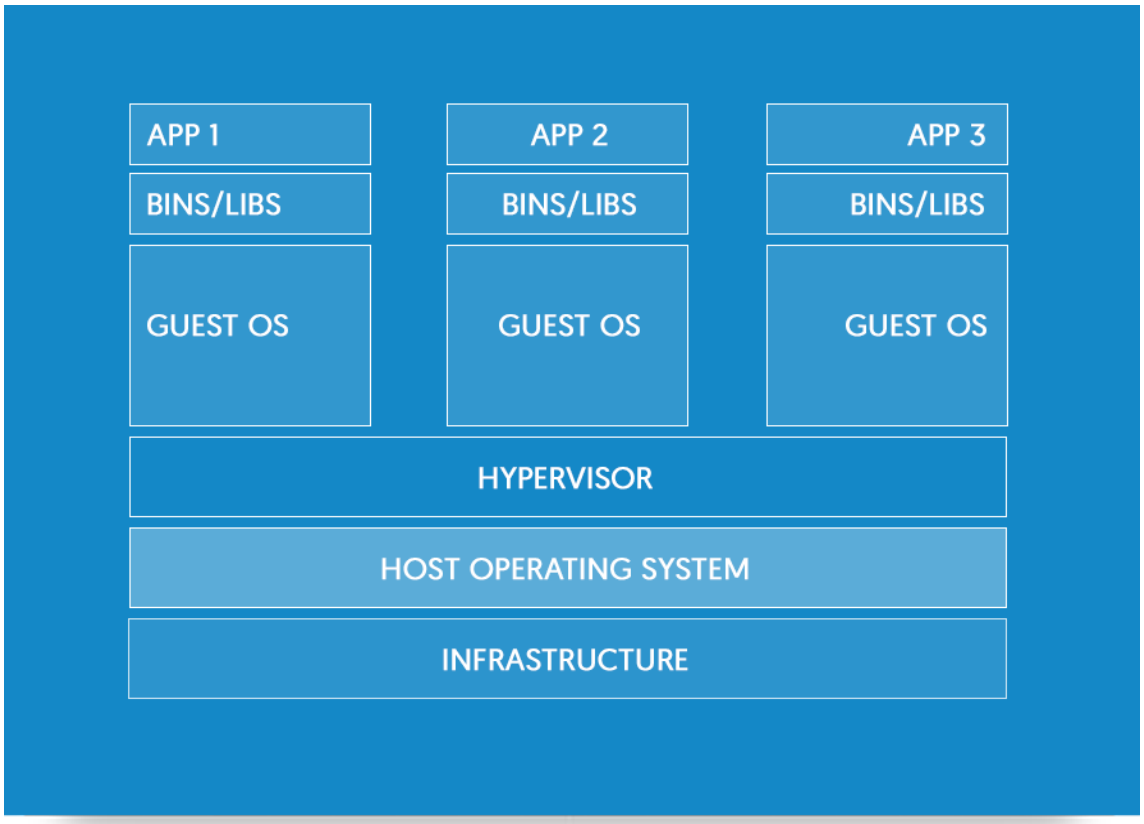


Figure 6. Virtual Machine Structure

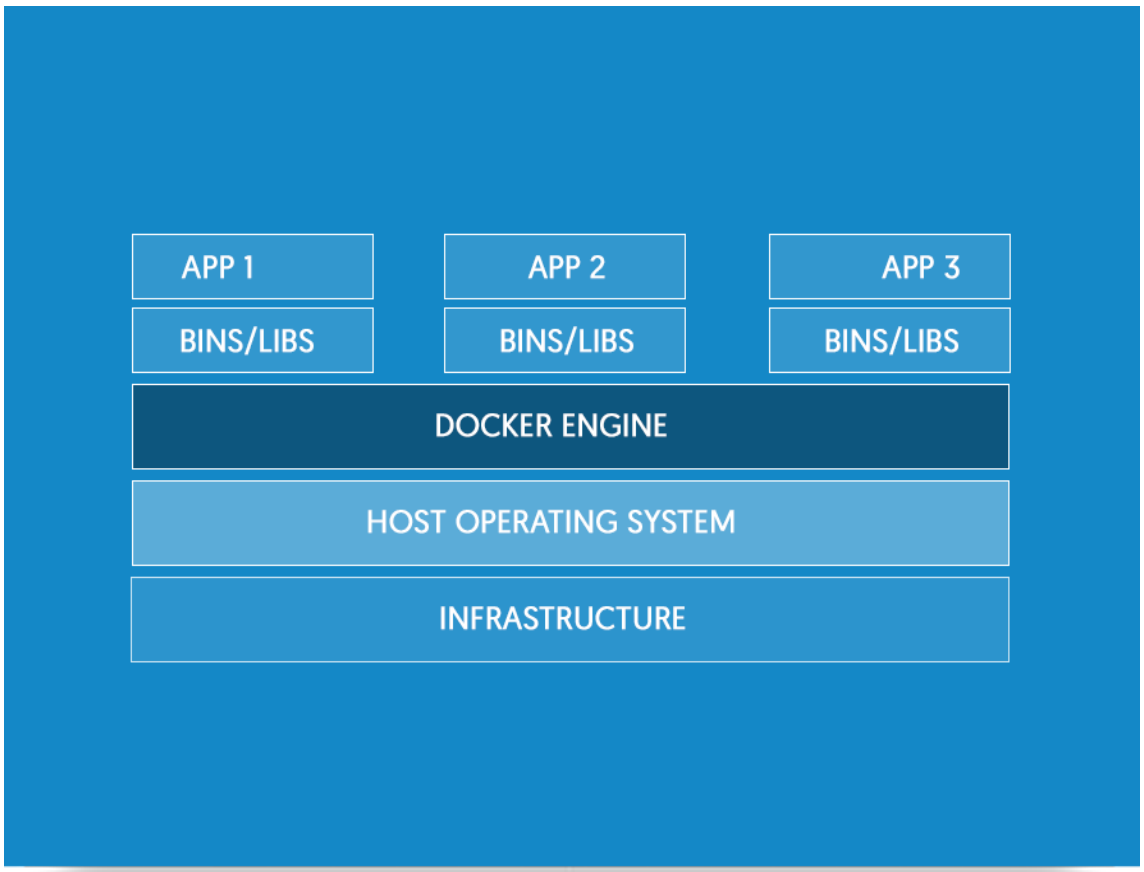


Figure 7. Docker Containers Structure

Docker is not the first software containerizing platform, the foundation for containerization lies in the LinuX Containers (LXC) format, which is a user space interface for the Linux kernel containment features. Other containerizing solutions would be CoreOS's rkt, Microsoft Drawbridge, and Ubuntu's LXD.

4 REQUIREMENTS AND SYSTEM ANALYSIS

In this chapter, a detailed description of the requirements and system architecture options are discussed.

4.1 System Description

Golv1 is a flooring, tiling, and painting small in-house construction company that found a need to automate their work and did not find a potential solution in the market that solve their problems. The idea of building a customized solution came up at the early stage, however, as the need of such system and the lack of an existing quotation system that is intended for those businesses appeared, Pool Interactive saw an opportunity to build it as SaaS. /22/

Each company will have its own subdomain and its data, including users, customers, services, materials, categories, packages, templates, and quotes, separated from the others. The users of the system will be able to save the company's customers information, the type of services they provide, and the materials they use. After that, they will be able to create quotes for a client depending on the previous information, where each quote will have the measurements, the active date, the services, the materials, and the total calculated automatically. After that, the administrators will be able to invoice the customers depending on the information that the quote contains. The system works like that in its basic form. However, more details and functionalities needed to be handled later on during collecting requirements and development stage.

4.2 Collecting Requirements

Understanding the problem is the first step to solving it, and to achieve that, all the possible necessary information about the problem needs to be collected. In most of the agile software development methodologies, the lifecycle of the development process is relatively short and happens frequently, keeping the potential user of the outcome software, application, or system in the circle and informed of each added feature as early as possible. Such practice of developing software has proved its effectiveness and efficiency.

The following are the results of several meetings and gathering requirements, which gave a good enough understanding to outline the main features of the system and to analyze probable accompaniments in the future.

- Roles and permissions.

  Three levels of roles are being discussed at this stage; super admin, admin, and operator or maintainer, and the access control will be determined by the roles where the permissions are fixed for the whole role.
- Management functionalities.

  The super admin role holds the responsibility of these functionalities.

  A company's details such as business name, the company's representative name, address, and phone number need to be collected for further use, in addition to managing users with their roles.
- Administration functionalities.

  Two roles will be responsible for these functionalities, which are admin and super admin.

  Handling customers' information such as name, email, phone number, address. Managing services and materials information like title and price. Dealing with templates, which represent the skeleton of the quote, as well as supervision of quotes and invoices.
- Quoting functionalities.

  All three roles have access to these functionalities.

  Consisting of creating quotes for customers immediately, and have them pushed to the admins to confirm and proceed.

Further discussions revealed more features that considered general and edge cases.

- Starting tools and plans costs: Some additional costs were not fitting as services that could be added to the general service components. Thus, a starting tool or plan component became a substitute for such purpose.
- Packages: Typically, a service needs material. This need formed a necessity to have some packing component that wraps a service and material together. A list of packages is what put a template together.
- Types and categories: When dealing with an area of work; for instance, floor or ceiling, services tend to vary. Therefore, package types and categories are necessary to distinguish and locate a package while creating a quote.
- Extra costs: Some quotes will have additional costs that are not part of a template or a starting plan which is being handled by adding them directly to the quote itself.

4.3 **Analyzing the Requirements**

At this stage, a formal understanding of what exactly needs to be implemented has been reached. Analyzing the requirements and giving examples will help more regarding how to put all the previous components together to form the whole functioning system. Some clarification will be dealt with to help understand each element separately, and how combining two or more will create another.

The following figure will contribute to illustrate the whole idea of the components.

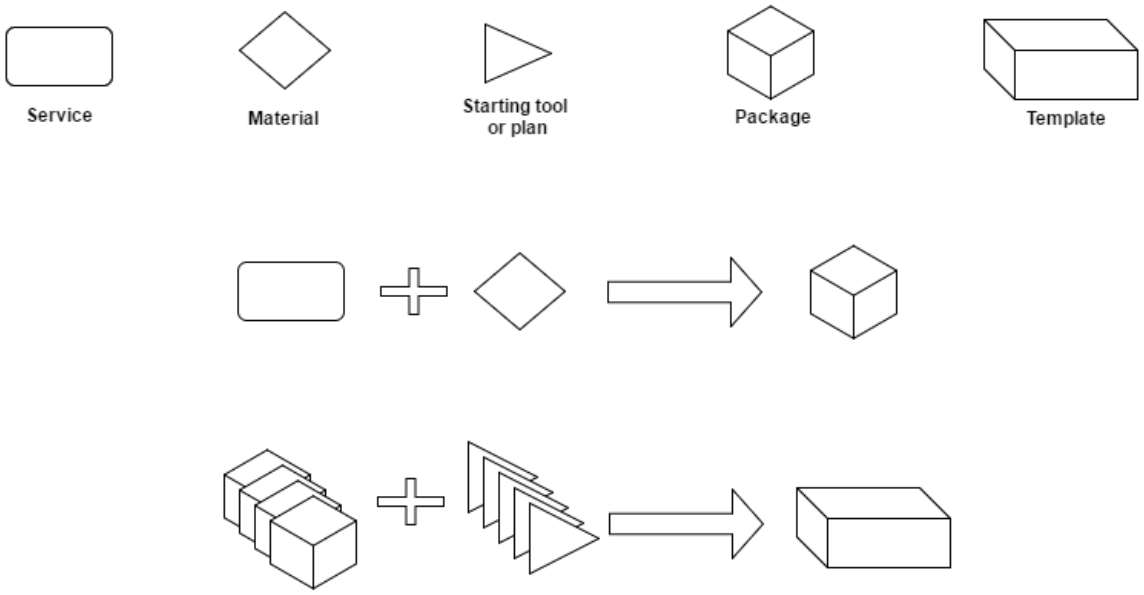


Figure 8. The system's components

- Service: Any act, or deed that the company offers as part of its services. For instance, installing ceramic and porcelain floor tiles is a multi-steps process where one needs to plan the tile layout, mix the thin-set and apply it, test the mortar and begin laying tile, and comb the thin-set. Each step represents a service.
- Material: Any physical substance that is used by the company to perform a service. For instance, while painting a wall could be a service, paint is a material that is employed in this process represents a material.
- Starting tool or plan: Any extra cost that is not combined with a single service and cannot be created as material. For example, a roll that is used as a tool for painting services is considered a starting tool, which cannot serve as a material because it is being used for multiple times. Costs for driving to the customer’s construction site is a valid starting plan.

- Package: Each service, to fulfill its needs, requires a material. This combination of a service and a material called a package. Polishing walls is an example of a package. However, some services might not require a material to be performed, and that is valid when creating a package. Taking measurements serves as an example. Every package must be assigned an area (floor, wall, or ceiling) that it will be performed upon it.
- Template: These are the base that the quote will be created upon. Floor tiling is an instance of a template. Templates contain packages and starting tools and plans.
- Quote: The main component of the system that will be the contract between the company and the customer to deliver its services. “Refurbishing the kitchen” is an instance of a quote.

4.4 System Architecture and Microservices

After having a whole idea of the system requirements, and analyzing them to form and understanding of how the system should work, the architecture procedure is the next phase. Microservice architecture approach, which is an approach for architecting large applications as modular services each has a specific business goal, seems to be a good fit for such system for many reasons.

Monolithic application means that an application has various capabilities, and all those capabilities are put together in a single application that runs in a single process. Microservices approach tends to take each of these capabilities and place them in their own processes where instead of having one process, have a network of communicating process. A good example of this is the Unix command line, where if a list of all the files in a directory sorted, multiple different commands will be placed in a pipeline to do so.

It also has a consequence for distribution. In a monolith approach, scaling would be by cluttering the application on multiple machines, while in a microservices approach, a this is more flexible where different services could be placed on various machines, and in case a service has more load than others, another copy of them could be added.

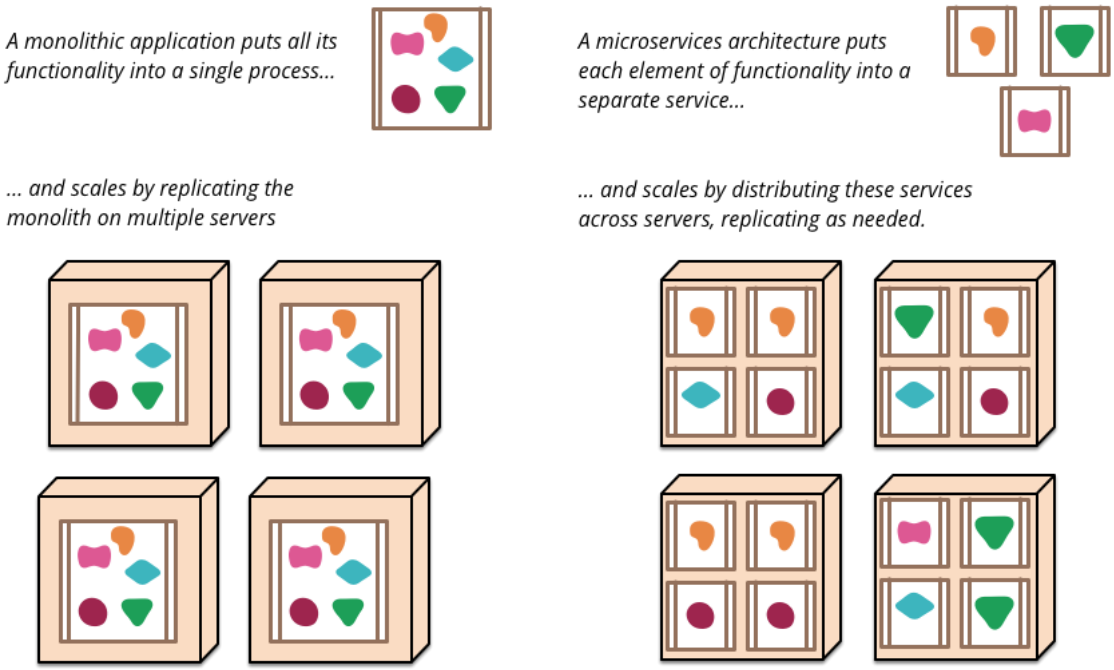


Figure 9. Monolith and microservices /23/

Services in microservices architecture are small, independent applications, where each is running in its own process, and communicate with other services with a lightweight mechanism, usually APIs, to form a single system or application. Each service is a language-agnostic, highly decoupled module that has a single responsibility and does a small task. /23/

The following is a list of some of the common characteristics of microservices architecture.

- Componentization via services: A service is behaving as an independently replaceable and upgradeable component, where if a single service needed to be changed, none of the others needs to be tampered with.
- Organization around business capabilities: Most of the software development organizations organize themselves around the technology, where one can find DBAs, DevOps, back-end or server developers, and front-end or UI designers. In microservices architecture, the key is that the organization should happen around business capabilities and focus on the end users.
- Decentralized data management: In the monolithic approach, the whole data is setting in one big database and often commits to right across the entire company teams, while microservices architecture suggests that each service is responsible

for its data and its persistence. A service can never talk to another service persistence layer directly, but only through its API.

- Infrastructure automation: Techniques like continues delivery and blue/green deployment that allow the business to put changes alive with zero downtime are mandatory to make microservices architecture works.

There are many factors to consider when to use monolith or microservice approach.

- Monolith is more straightforward and familiar approach to use, while microservices approach tends to have more complexity because it introduces distributed computing and asynchronous communications.
- Microservices approach has the ability to deploy various parts independently, while this is hard while dealing with the monolith.
- Microservice gives a high availability to the whole system. For instance, if a service is down for some reason, that does not affect the other services. However, this will make consistency much harder to maintain.
- Microservice helps preserve modularity and keeps module boundaries solid which is a good practice in software development.
- Microservice allows using multiple platforms. For example, a service using a programming language while another service using different programming language.

Considering the characteristics of the microservices architecture, and the factors above, this approach seems to be a good fit for the quotation system. The system will consist of a service for registering new companies and initiate a new instance of the application, a service for dealing with subscription plans and payments, a service of the quoting functionalities, and a service for invoicing.

4.5 **UML Diagrams**

In software engineering, UML is a modeling language that is used to visualize the design of the system. /24/

The following is the use case diagram of the quoting application.

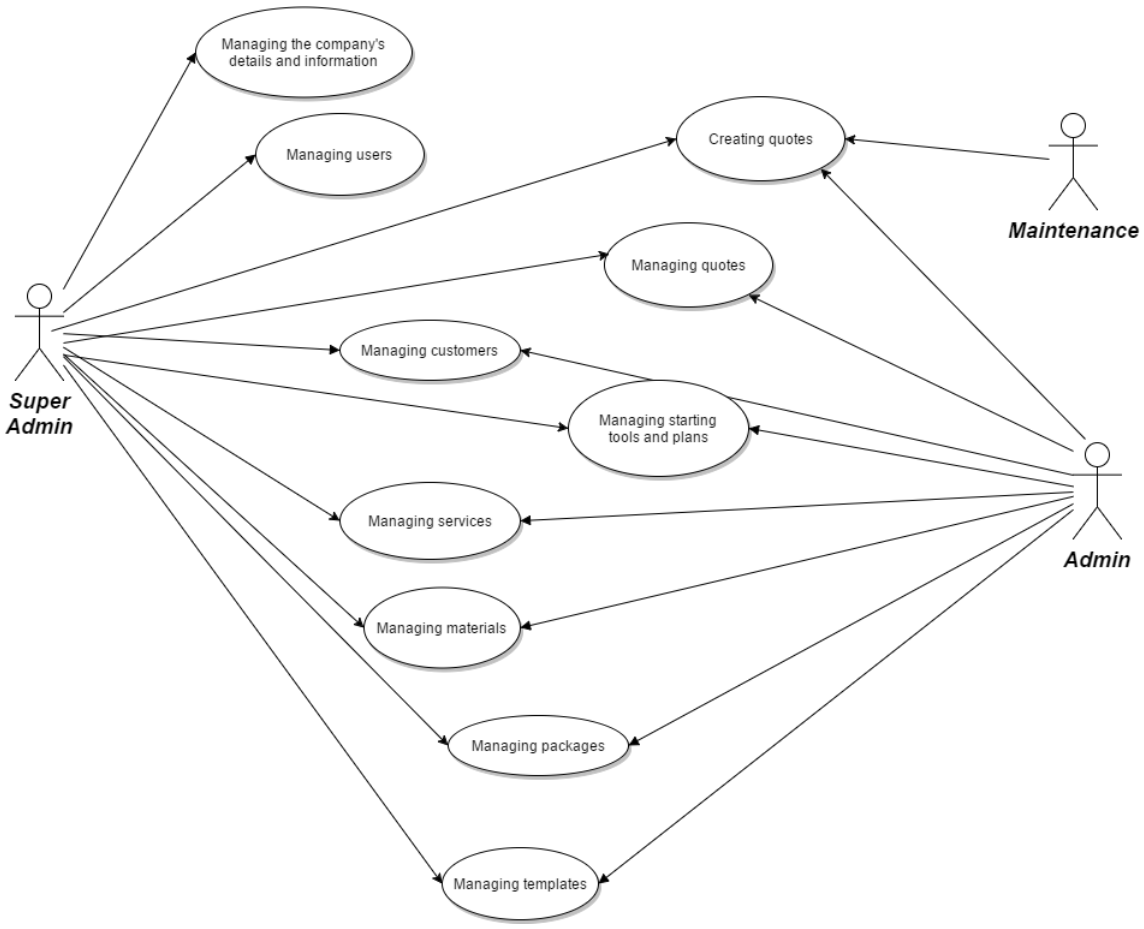


Figure 10. The use case diagram of the quotation application

The diagram shows the user cases and the actors in the application. Each use case represents a functionality or a set of functionalities toward a component in the system and which role can perform this use case. For example, admin can manage the customers, create a new client, update his/her data, and delete a client, but a maintenance actor cannot perform these functionalities. In the same way, the super admin can access and manage the company's details and information while an admin and maintenance role cannot accomplish such action.

The following is a partial entity relationship diagram that represents the relationship between some components of the application.

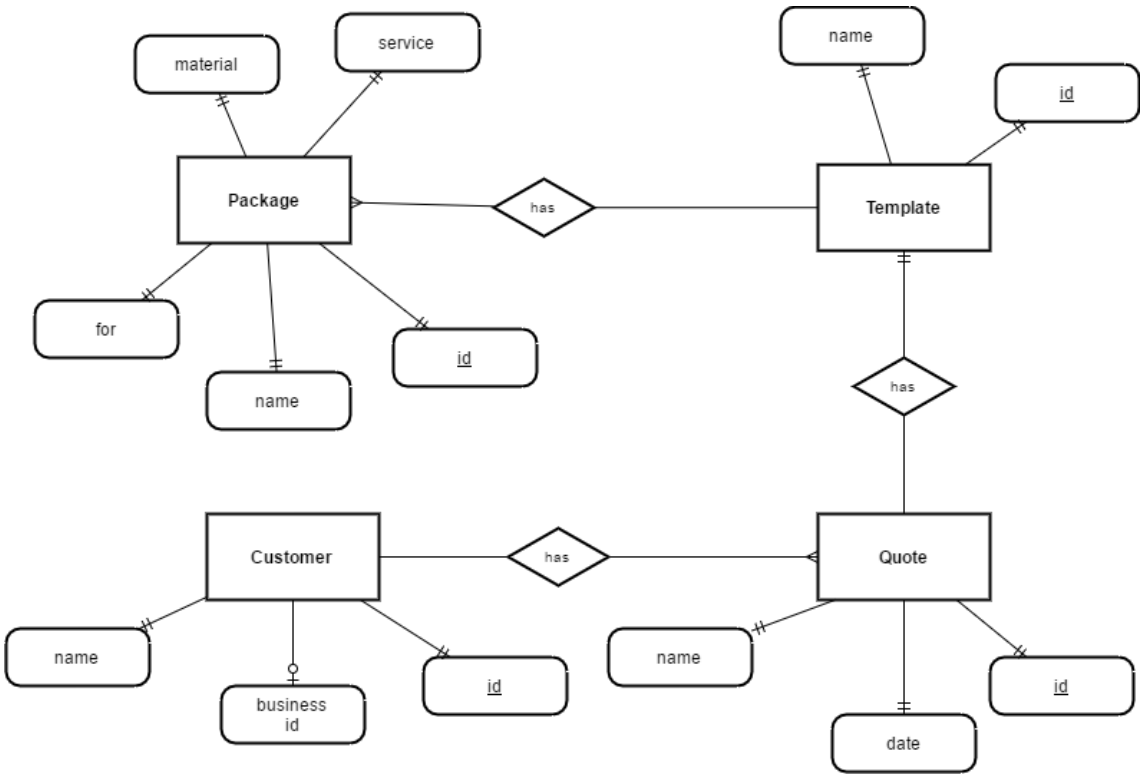


Figure 11. Entity relationship diagram of some components

From the entity relationship diagram, one can notice that each customer has a quote or multiple quotes (one-to-many), where each quote has a template (one-to-one), and each template consists of multiple packages (one-to-many).

A class diagram representation of the previous relationship is shown below. Explicitly, the relationship between the classes is presented. The “1” represents that the class of this side of the relation must be presented once. However, “0..1” embodies that the class might not occur, but if it did, it is one occurrence.

Similarly, “0..*” embodies that the class might be present once or multiple time, or might not. Finally, “1..*” expresses that the class should be present at least once.

For instance, each customer can have zero or many quotes, where the quote must belong to one customer only. A quote must have one template associated with it only, while it must have at least one package.

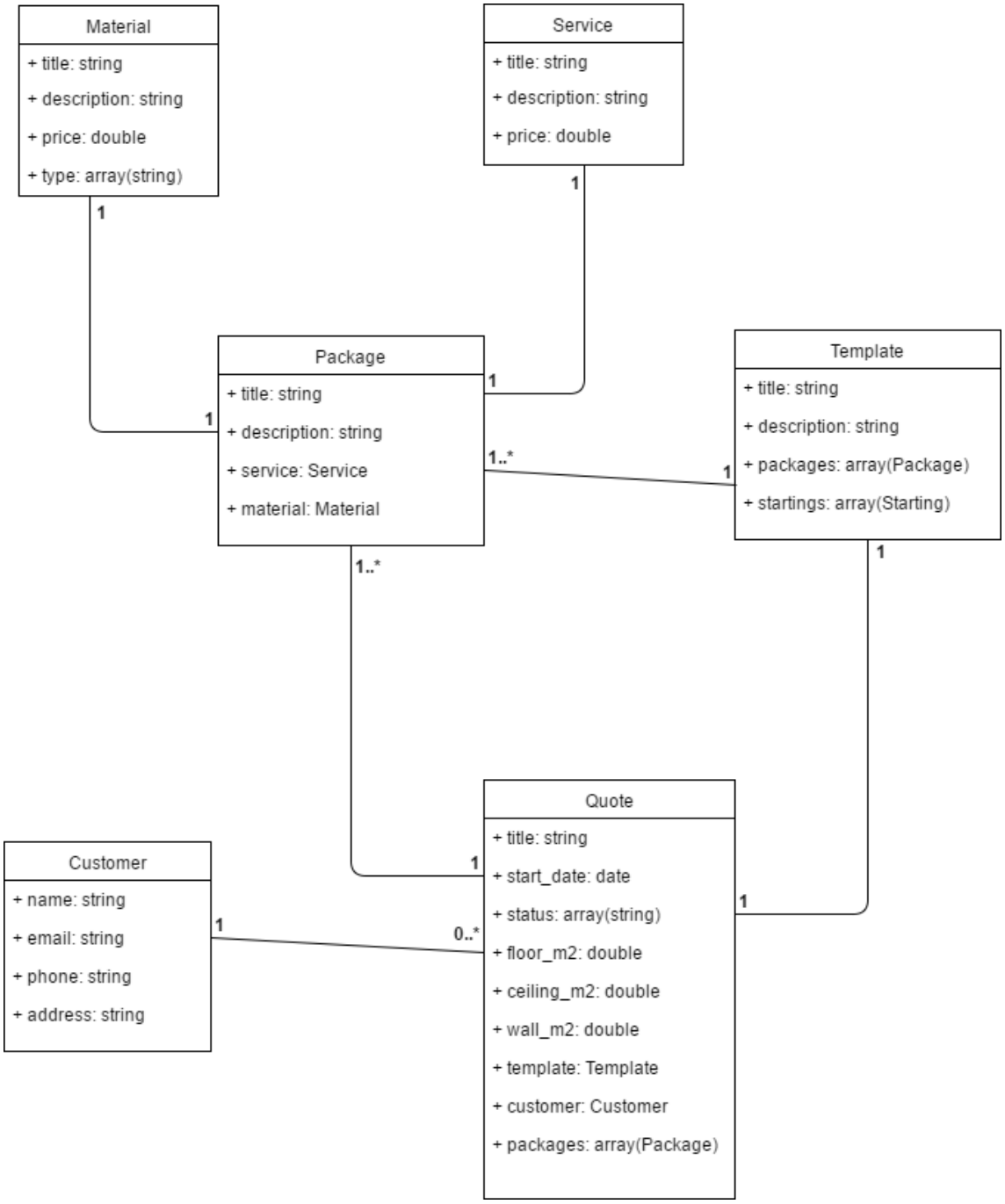


Figure 12. Class diagram of some of the system components

The previous UML diagrams are preferred in every software design stage, but not limited to them only. For instance, package, sequence, activity, and component diagrams are significant decomposing or aggregating system parts visually to provide a better understanding.

5 SOLUTION STRUCTURE AND RELEVANT TECHNOLOGIES

In this chapter, the solution structure and the technology stack will be discussed.

5.1 **Data Persistence**

Data persistence is where the data are being stored. There are different options for persisting data, mainly relational databases, and non-relational databases, and for those, there are also various options. For instance, some examples of the non-relational databases are JSON-based storage like MongoDB, key-value data store or in-memory data structure store like Redis, or graph database like neo4j. /25/ /26/ /27/

Non-relational databases tend to have faster access and can handle a larger amount of data than the relational databases. However, relational databases have the ability to manage relations and transactions reliably. Most of the non-relation databases lack transactions where atomic modifiers, the A in ACID, can only work against a single entity.

The quotation system relies highly on transactions and relations between its components, which made relational database management systems more convenient and suitable solution. MySQL is an open source RDBMS, and it is part of the LAMP stack. Many content management systems use MySQL including WordPress and Drupal, and it is also used in several prominent websites and web applications including Facebook, Twitter, Flickr, and YouTube. /28/

Depending on the entity relationship diagram, a database diagram has been created.

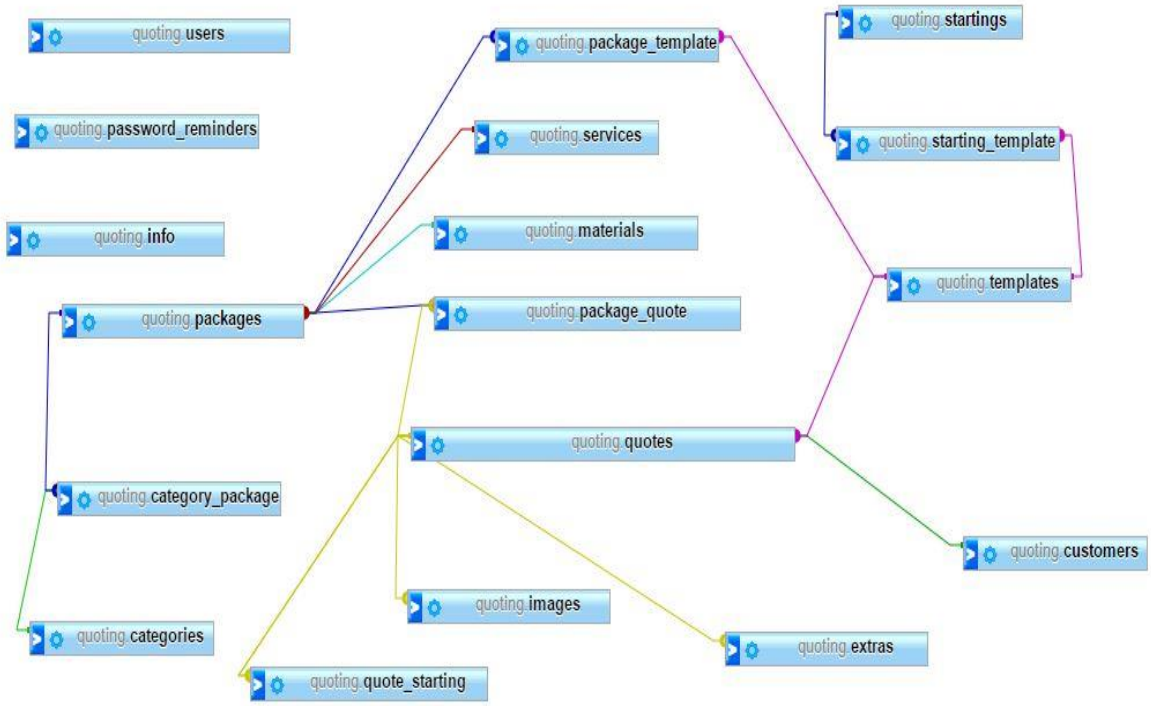


Figure 13. The database diagram of the quotation application

The diagram shows the relationship between all of the components in the quoting application.

5.2 **Back-end**

MVC design pattern was chosen to be the architectural pattern for the back-end. Even though this pattern was created for desktop applications, it became commonly used as an architecture for web applications in many programming languages. Many web frameworks impose the pattern and divide the responsibilities between the client and server. /29/

Laravel is an open source MVC framework built using PHP. It has many features that make it one of the most popular MVC frameworks. The following are some of these features. /30/

- Routing: Laravel provides an interface for defining routes of the application. The simplest route accepts a URI and closure.
- Middleware: It provides a mechanism for filtering HTTP requests entering the application. For instance, Laravel comes with a middleware that verifies the au-

thenticated user. If the user is not authenticated, it will be redirected to a login page or other pages that have been defined by the developer. Middleware can be global, which runs on every HTTP request to the web application, assigned to a specific route, or grouped with several others.

- Controllers: A controller class organizes request handling and business logic of the web application. Restful or resource controllers is a way of assigning a CRUD route to a controller with a single line of code.
- Validation: Laravel provides multiple approaches to validate incoming data.
- Eloquent ORM: Eloquent is an object-relational mapper that implements Active Record design pattern for working with the database. It presents database tables as classes, with their object instances tied to single table rows.
- Query builder: It provides classes and methods to create and run database queries without letting the developer writing them manually and uses PDO parameter binding to protect the application against SQL injections attacks. It also provides a caching mechanism for the results of executed queries.
- Migrations: It provides an easy way to build the database schema of the application using Laravel's schema builder, and act like version control for the database. This makes the deployment and updating the application simple.
- Seeding: By using seed classes, Laravel allows seeding the application's database with test data, or initial data for the application setup.
- IoC Container: Inversion of Control Container is a design pattern for managing class dependencies and performing dependency injection. The whole Laravel framework idea revolves around IoC Container, and understanding that is vital to building a large, well-organized applications.
- Artisan command: Artisan is a CLI that comes with Laravel which provides several useful commands to assist while building a web application. Some of the commands would be for quickly creating controllers, caching and clear cache, migrating and seeding, dealing with the environment file, and running the application on the PHP development server. Artisan commands are not limited to these features, and Laravel allows developers to extend the list of commands by providing an interface to create new ones.
- Autoloading classes: It provides an automatic loading of the classes without the need for manual maintenance of inclusion paths. That will load only the used components.

- Dependencies: Laravel uses composer as a dependency manager to add framework-agnostic and Laravel-specific packages from Packagist repository. /31/

The following are some of the Laravel-specific packages that are ready to use.

- Cashier: For managing subscriptions billing services, handling coupons, and generating invoices.
- SSH: For executing CLI commands programmatically on a remote server using SSH protocol.

The back-end has the following structure, which is based on Laravel 4.2.

Figure 14. Back-end application structure

The app directory contains the application logic that contains:

- “classes” directory: It has been added to the application for customized classes that don’t follow the framework definitions.
- “config” directory: It handles the configurations of the application including the application API keys, timezone, authentication, cache, database, mail, and remote connections.

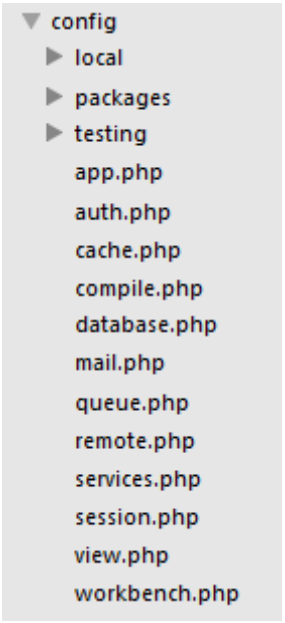


Figure 15. The config directory

- “controllers” directory: It is the directory that contains all the controllers.
- “database” directory: It contains the migrations and seeds.

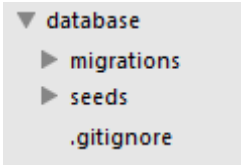


Figure 16. The database directory

- “models” directory: Containing all the models of the application.
- “start” directory: Which contains all the initiations to help in the application set-up.
- “storage” directory: Contains the log, session, and cache files.
- “tests” directory: Contains the test cases for the application.
- “views” directory: It contains the HTML and PHP templates that are used to view the output. In this case, it is used only to render the outcome for emails and PDFs templates.

- “filter.php” file: It has the application and route filters which are events that are used to do any work before or after a request to the application.
- “routes.php” file: It contains the routes to the application.

At the same level of the app directory is the vendor directory which contains all the packages that are being installed by composer including the Laravel code base. Also, the public directory which usually has the JavaScript and CSS files. The bootstrap directory which contains the autoloading file that register the composer autoloader, the paths to the app directory, public directory, and storage directory, and the start file that will create a new Laravel application instance which serves as the "glue" for all the components of Laravel, and is the IoC container for the system binding all of the various parts.

Laravel makes it easy to have multiple different configurations for different environments in the same application. For instance, the application could have development, production, and testing environments and easily switch between the three as needed. That is useful for situation where running tests should be on testing stage, and the application should have configurations that suit the case. Same while sending emails, where in the development stage, the application should not actually send any email but mock the process of doing so.

5.3 **Authentication and Authorization**

Because the back-end is being built as a REST API, there are no sessions to be tracked while dealing with authentication. That produced a need to find another mechanism to handle the authentication. One solution is to use a token that will be sent in the header of each request to the API.

JSON Web Token (JWT) is an open standard JSON-based object defined in RFC 7519 that assert claims for securely transmitting a digitally signed set of information using a secret or a public/private key pair.

JWTs are small in size which makes the data transmission between the server and the client faster and could be transmitted through HTTP header, URL, or even POST parameter.

JWTs could be used in different scenarios. However, the most widely used one is in authentication. Each request to a route or resource that is permitted with a JWT will include the token allowing a logged-in user to access that route or resource.

The process usually follows this scenario. The user logs in with his/her credentials and a JWT will be returned and should be saved locally, typically in local storage, session storage, or a cookie. Then whenever the user agent sends a request to a protected route or resource, the JWT should be attached to the "Authorization" header using "Bearer" schema. It looks like the following.

```
Authorization: Bearer <token>
```

The server will get the JWT from the "Authorization" header, and allow or decline the access depending on its validity. The following figure presents the last process.

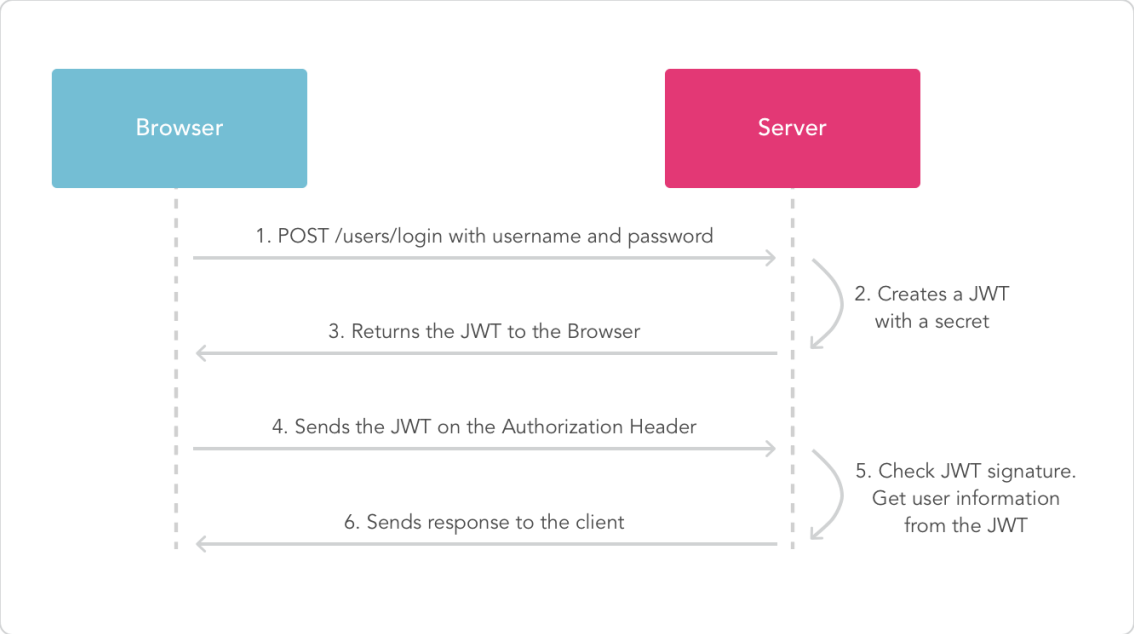


Figure 17. The working process of the JSON Web Tokens. /32/

The statelessness of the JWT makes it a great fit for stateless data APIs, not mentioning that dealing with Cross-Origin Resource Sharing (CORS) is not an issue as the cookies are not used. Also, because JWTs are self-contained, the required information for authentication are in the token which eliminates the need to query the database.

The token is composed of a header, a payload, and a signature separated by dots ".", therefore, it looks like the following: "xxxx.yyyy.zzzz" as "header.payload.signature" structure.

The header entails two parts: the used hashing algorithm, and the type of the token. "HMAC SHA256" and "RSA" are some of the examples of the hashing algorithms, and the type of the token is JWT. The header will be "Base64Url" encoded forming the first part of the JSON Web Token.

The following is an example of the header.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The payload consists of the claims. Claims are the data of the entity, which is usually the user, and other metadata. They are divided into three types: public, private, and reserved. The public claims are defined by the users of the JWTs and should be defined as a URI that has a collision resistant namespace or identified in the IANA JSON Web Token Registry. The private claims are the customized claims that are created for sharing the information between the server and the client agent. The reserved claims are the recommended predefined claims, which their names are three characters long. The following is a list of some reserved claims.

- "iss": Is the issuer of the JWT.
- "sub": Is the subject of the JWT.
- "aud": Is the audience of the JWT. The audience represents the intended recipients of the JWT.
- "exp": Is the expiration time of the JWT.
- "iat": Is the "issued at" claim that identifies the issue time of the JWT.
- "jti": Is a unique, case sensitive JWT ID.

The payload will be "Base64Url" encoded forming the second part the JWT. The following is an example of the payload.

```
{
  "sub": "0987654321",
  "name": "John Doe",
  "id": 98743,
  "admin": true
}
```

The signature is the last part of the token which consists of the sign of the encoded header, the encoded payload, a secret string, and the algorithm the has been specified in the header. It is used for verifying the sender of the JWT and determining that the message has not been tampered with while it has been sent. The following shows how to create a signature using the HMAC SHA256 algorithm.

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret )
```

Adding all the previous parts together forms the token. The following is an example of a signed JWT.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Figure 18. An example of the header, payload, and signature parts forming the JWT.

JSON Web Tokens are not the only solution when it comes to stateless authentication or sharing information securely, some other options exist, like Security Assertion Markup Language Tokens (SAML), and Simple Web Tokens (SWT) and both options use XML to transfer data. /33/ /34/

However, JWTs have the prosperity of using JSON instead of XML where the prior is less verbose and smaller in size when encoded than the last. Usage-wise, JWT is suitable for multiple platforms (like the web, and mobile) due to the easiness of the processing it. For example, JSON parsers map to objects directly in most of the programming languages, while XML does not have this natural mapping. Security-wise, SWT can use HMAC algorithm to symmetrically sign a token with a secret, while JWT can use public/private key pairs. However, SAML can use the last approach, but signing

XML with XML Digital Signature is very difficult comparing it to signing JSON. The following figure shows the length of an encoded JWT compared to an encoded SAML.

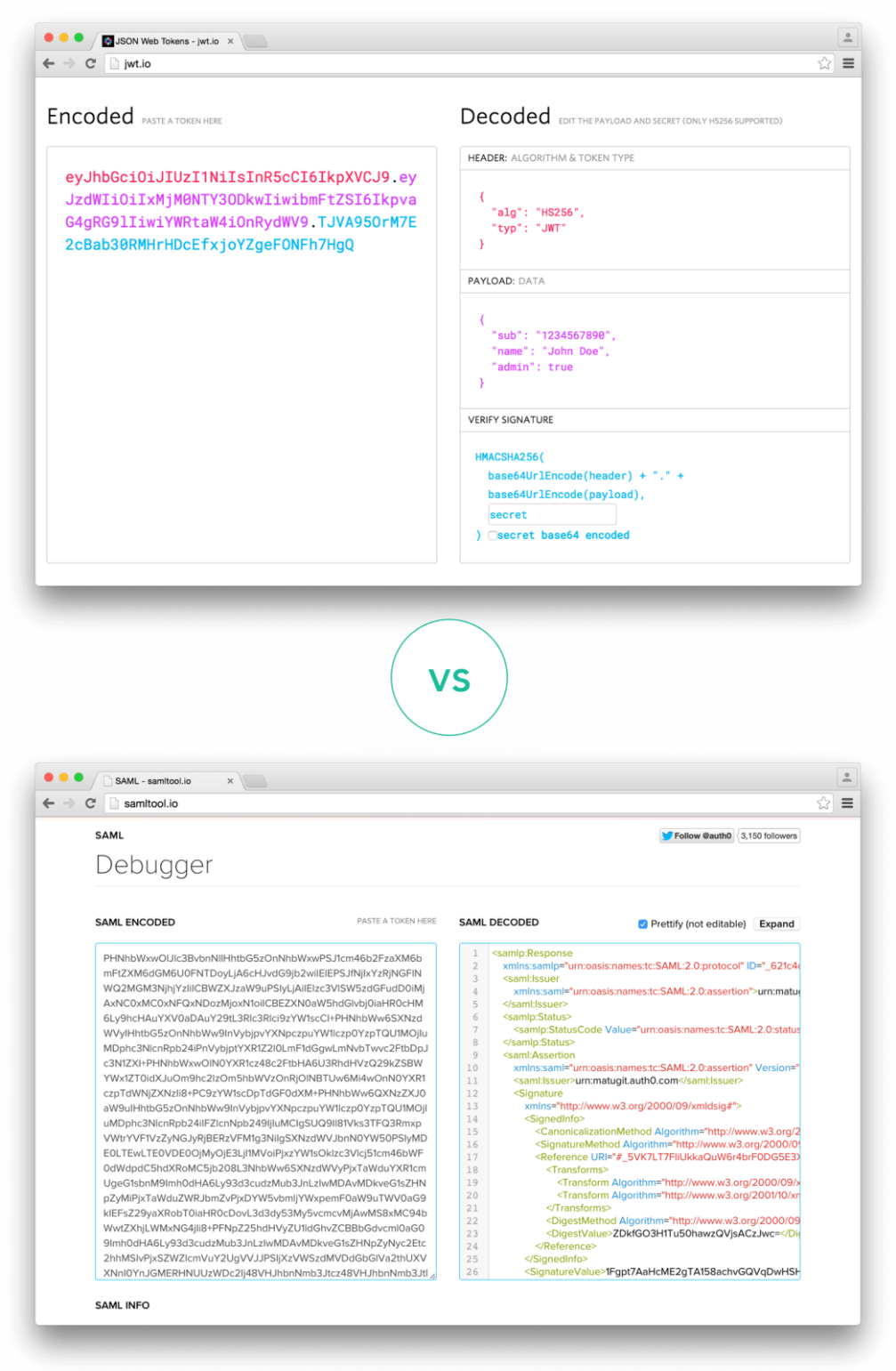


Figure 19. The length of an encoded JWT compared to an encoded SAML. /32/

5.4 **Front-end**

The web application was built as a Single Page Application using AngularJS MVW framework, where MVW stands for Model-View-Whatever. AngularJS was identified as MVC pattern, but due to many changes in the API itself and refactoring big amount of its code, it is now more like MVVM. That created a split in the AngularJS community and created inconsistency in the paradigm, which made the development team change it to MV* or MVW where the developers can use whichever paradigm that suits their needs, MVC or MVVM. /10/

Some of the features that AngularJS provides are the following.

- Data binding: Also known as two-way data binding which is a way where if the view gets updated, the model will change as well and vice versa.
- Deep linking: AJAX driven applications do not have the ability where the link echoes where the user is in the application. AngularJS provides this benefit making the web application feel more like a desktop application.
- Form validation: Is a mandatory part while dealing with applications that want to provide a good user experience to its users. Thus, AngularJS provides a mechanism to maintain form validation without the need to write the logic in JavaScript.
- HTTP Requests: AngularJS has services that are built on top of the XHR JavaScript API and simplifying the asynchronous calls by using promises.
- Reusable components: A component is a way to package an HTML, CSS, and behavior using directives to hide the complexity and reuse that component.
- Dependency injection: Where a part of the application will use only the needed component and functionality and leave the rest.
- Testability: AngularJS was built to be testable assuring the business logic separation and providing mocks.
- Localization: AngularJS has filters, modules, and directives making the application available in multiple locales.

Those features and more made AngularJS a good fit for the system. Working with AngularJS requires an understanding of some JavaScript concepts and design patterns. The following is a list of some concepts that need to be explained in order to be efficient with JavaScript and AngularJS. These concepts are made while dealing with functions.

- Function as an abstraction: Using this approach when dealing with functions that are written by a third-party developer and the maintainer of the application wants to sanitize those functions or simply seeks to add some routine to it.

```
var output = function () {
        console.log('this is an output');
}
var printOutput = function (fn) {
        console.log('starting');
        try {
                fn();
        } catch (ex) {
                console.log(ex);
        }
        console.log('ending');
}
printOutput(output);
```

  The example shows how to define functions as an abstraction. The "output" function expression represents the third-party code, while the "printOutput" function expression represents the abstraction layer. "printOutput" accepts a function as a parameter, wrap it in a try-catch statement, and adding a routine of simply logging "starting" and "ending" correspondingly before and after the function execution.

- Building modules: A module, in computer science and software development, is a collection of data and features/methods that could be packaged together to perform useful work. The main aim of the modules is to hide the information and the data from the outside access providing an API to the rest of the application to interact with it. The following example is a design pattern called "Revealing Module Pattern" showing how to use functions to create a module.

```
var createWorker = function () {
        var workCount = 0;
        var task1 = function () {
                workCount++;
                console.log('task1' + workCount);
        }
        var task2 = function () {
                workCount++;
                console.log('task2' + workCount);
        }
        return {
                job1: task1,
                job2: task2
        }
}

var worker = createWorker();
worker.job1();
worker.job2();
```

The module is the “createWorker” function expression. It has “workCount” which is a local variable, and “task1” and “task2” as local function expressions. The return statement is what makes the revealing module pattern because it is providing an object with “job1” and “job2” properties revealing the functionality of the module “task1” and “task2” consecutively. However, the “workCount” variable remains local to the module as it is not exposed in the return statement. The “worker” variable in the global scope creates an instance of the module and can call the methods by the property key name like “job1” and “job2”.

This pattern is helpful while dealing with AngularJS services and factories. For instance, while dealing with customers CRUD operations API calls, a service or a factory could wrap these functionalities in a function and reveal it using the last pattern.

- Immediately Invoked Function Expression (IIFE): In software development, global variables are well known for being bad practice and should be avoided. They are a source confusion and bugs, especially in dynamically typed languages like JavaScript because it is easy for override a global variable that was defined before. This behavior is dangerous, and while the module pattern helps in this by reducing the global variables to one, IIFE contributes to reducing that number to zero by defining a function and invoking it on the spot. The following shows how to set an IIFE declaring an anonymous function and invoking that function immediately.

```
(function() {
        // Code
}());
```

The front-end web application follows a modular approach and single responsibility principle for structuring where each component is placed according to the business expression that it implies. For example, a user profile is a component that has its own directory which contains the controller, the view, and the services that are needed to make this component works. While the other approach of defining the structure depending on functionality, like a directory for controllers, services, and others, is more suitable for small applications, it does not seem to useful approach for this system.

The following figure shows how the front-end application is structured.

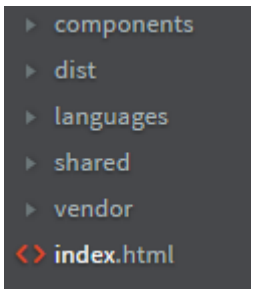


Figure 20. Front-end web application structure

- "index.html" file: Is the starting point of the application.
- "components" directory: Contains the application components, each as a directory which has the controller, the view, the services, and any other specific functionality for that component like filters.

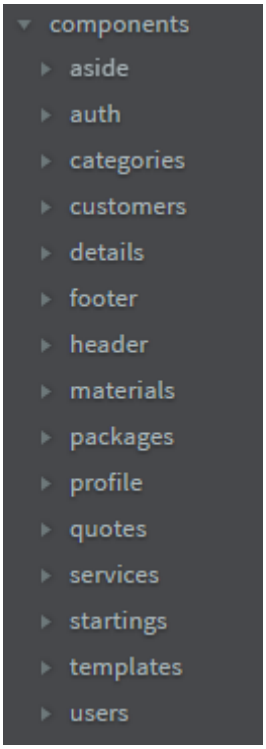


Figure 21. Front-end's component directory

- "dist" directory: Contains the customized CSS, images, and JavaScript files that are general to the application.
- "languages" directory: Contains the JSON files that are used for as human languages translations that the system provides.
- "shared" directory: Contains the common functionalities that are used across the application like filters, directives, and views.
- "vendor" directory: Contains the third-party frameworks, libraries, plugins, modules, and directives that are external dependencies for the application. Also,

"app.js" which contains a definition of the father application module, app configuration, and routing functionalities.

The last structure helps to apply AngularJS best practices like the following.

- Rule of one: This states that a file should contain one and only one component. For instance, defining a module, a controller, and a service/factory each should be in its file.
- Using "controllerAs" syntax: It is a syntactic sugar over "$scope" to help avoid using the "$scope" methods and consider them only when needed.
- Define a controller for a view: This will keep the controller focused and non-reusable where the reusable functionalities should be moved to the services and factories.
- Single responsibility for services and factories: A service or a factory should have a single responsibility, and once the service starts to surpass its determination, a new one should be created.
- Separating API calls: Some tend to forget the thin controller concept which states that the controllers should not deal much with the business logic, but directing the logic to the right place. In AngularJS, controllers are for presentation and collecting data for the view knowing whom it should as for the data and not how to get it. Therefore, creating data services that are responsible for API calls, local storage, and other data operations is a good practice. It also helps while testing controller by mocking those calls.

6 SYSTEM IMPLEMENTATION: BACK-END AS API

The following chapter will discuss the process of building the back-end API-centric application including installing Laravel, configuring the environment and the application, defining models, creating database migrations and seeding, declaring routing, handling the business logic, and tweaking the performance among other tidbits.

6.1 **Installing Development Requirement**

Ubuntu 14.04 is the OS that has been used for production and development servers, and LAMP is the technology stack that has been used for developing this application. The production server requires multiple steps of configuration, which is why the installation configurations of the software were minimalistic on the development server.

First, starting with installing Apache that would serve as the web server using the following command.

```
sudo apt-get install apache2
```

Second, installing MySQL as a database server using the following command.

```
sudo apt-get install mysql-server
```

Third, installing PHP with “php5-mcrypt”, “libapache2-mod-php5”, and “php5-json” meta-packages using the following command.

```
sudo apt-get install php5 php5-mcrypt libapache2-mod-php5 php5-json
```

Finally, installing Git version control system with the following command. /35/

```
sudo apt-get install git-all
```

The last four commands will install the latest stable version of each software that is available in the repositories used by Ubuntu. The server has Apache 2.4, MySQL 5.5, PHP 5.6, and Git 2.7 now installed and ready.

6.2 **Installing Laravel**

There are a few requirements before installing Laravel, and the following shows those requirements.

- PHP 5.4 and above.
- MCrypt PHP extension.

- Composer PHP package manager for managing Laravel's dependencies.

From the previous step, PHP 5.6 and MCrypt PHP extension are installed. However, the extension needs to be enabled using the following command.

```
sudo php5enmod mcrypt
```

After that Apache needs to be restarted using this command.

```
sudo service apache2 restart
```

The last requirement is installing Composer package manager. The following commands should be executed from the home directory. The commands will download the latest composer installer and name it "composer-setup.php", verify the installer SHA-384, run it with specifying the name as "composer", and lastly remove it. /36/

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') === '55d6ead61b29c7bdee5cccfb50076874187bd9f21f65d8991d46ec5cc90518f447387fb9f76ebae1fbbacf329e583e30') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php --filename=composer
php -r "unlink('composer-setup.php');"
```

Now composer is installed and ready.

There are multiple ways to install Laravel: using Laravel installer, using composer, using Git, or simply downloading it. However, the easiest and most efficient way is via composer using the `create-project` command.

```
composer create-project laravel/laravel {directory} 4.2 --prefer-dist
```

Running the previous command in the `\var\www\html` Apache directory replacing `{directory}` by the desired name will install and create a Laravel 4.2 project.

6.3 **Environment Configuration**

Laravel based projects must have a random string application key that is sat after the installation. However, because composer was used to install Laravel, this key was already sat. The application key ensures that the encrypted data and the user sessions are secure. Also, the "storage" directory requires a set of permissions to so that Laravel can gain access to its content like caches, sessions, and logs.

Laravel comes with ".htaccess" file for defining rules and configurations used by Apache specifically for this project directory. The following configurations are used to allow pretty URLs where the file name and extension are dropped.

```
<IfModule mod_rewrite.c>
  <IfModule mod_negotiation.c>
    Options -MultiViews
  </IfModule>
  RewriteEngine On
  # Redirect Trailing Slashes...
  RewriteRule ^(.*)/$ /$1 [L,R=301]
  # Handle Front Controller...
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^ index.php [L]
</IfModule>
```

However, Apache's "mod_rewrite" module should be enabled, and Apache server must be restarted using the following commands.

```
sudo a2enmod rewrite
sudo service apache2 restart
```

6.4 **Application Configuration**

As mentioned in the application structure, the application's configurations of Laravel 4.2 exist in the "config" directory which is located in the "app" directory. Each configuration file returns an array.

In the "app.php", the debug mode is sat to true which is helpful for debugging while the application is in the development phase where a more detailed error messages with stack trace will be displayed when an error occurs. The rest of the configurations like "timezone", "locale", "fallback_locale", and "cipher" will remain the same as preconfigured by the framework. Service providers can be autoloaded by referencing them in an array associated with "providers", and Laravel already has multiple service providers initiated.

```
'providers' => array(

        'Illuminate\Foundation\Providers\ArtisanServiceProvider',
        'Illuminate\Auth\AuthServiceProvider',
        'Illuminate\Cache\CacheServiceProvider',
        'Illuminate\Session\CommandsServiceProvider',
        'Illuminate\Foundation\Providers\ConsoleSupportServiceProvider',
        'Illuminate\Routing\ControllerServiceProvider',
        'Illuminate\Cookie\CookieServiceProvider',
        'Illuminate\Database\DatabaseServiceProvider',
        'Illuminate\Encryption\EncryptionServiceProvider',
        'Illuminate\Filesystem\FilesystemServiceProvider',
        'Illuminate\Hashing\HashServiceProvider',
        'Illuminate\Html\HtmlServiceProvider',
        'Illuminate\Log\LogServiceProvider',
        'Illuminate\Mail\MailServiceProvider',
        'Illuminate\Database\MigrationServiceProvider',
        'Illuminate\Pagination\PaginationServiceProvider',
        'Illuminate\Queue\QueueServiceProvider',
        'Illuminate\Redis\RedisServiceProvider',
        'Illuminate\Remote\RemoteServiceProvider',
        'Illuminate\Auth\Reminders\ReminderServiceProvider',
        'Illuminate\Database\SeedServiceProvider',
        'Illuminate\Session\SessionServiceProvider',
        'Illuminate\Translation\TranslationServiceProvider',
        'Illuminate\Validation\ValidationServiceProvider',
        'Illuminate\View\ViewServiceProvider',
        'Illuminate\Workbench\WorkbenchServiceProvider',

),
```

This array is extensible, and any added Laravel package could be included in this array, and the framework can have access to it on every request to the application.

There is also an array with the aliases of the registered classes that come with Laravel. This array is also extensible where any package that extends a Laravel's façade class can have an alias. The following shows the original array that comes with the framework.

```
'aliases' => array(

        'App'               => 'Illuminate\Support\Facades\App',
        'Artisan'           => 'Illuminate\Support\Facades\Artisan',
        'Auth'              => 'Illuminate\Support\Facades\Auth',
        'Blade'             => 'Illuminate\Support\Facades\Blade',
        'Cache'             => 'Illuminate\Support\Facades\Cache',
        'ClassLoader'       => 'Illuminate\Support\ClassLoader',
        'Config'            => 'Illuminate\Support\Facades\Config',
        'Controller'        => 'Illuminate\Routing\Controller',
        'Cookie'            => 'Illuminate\Support\Facades\Cookie',
        'Crypt'             => 'Illuminate\Support\Facades\Crypt',
        'DB'                => 'Illuminate\Support\Facades\DB',
        'Eloquent'          => 'Illuminate\Database\Eloquent\Model',
        'Event'             => 'Illuminate\Support\Facades\Event',
        'File'              => 'Illuminate\Support\Facades\File',
        'Form'              => 'Illuminate\Support\Facades\Form',
        'Hash'              => 'Illuminate\Support\Facades\Hash',
        'HTML'              => 'Illuminate\Support\Facades\HTML',
        'Input'             => 'Illuminate\Support\Facades\Input',
        'Lang'              => 'Illuminate\Support\Facades\Lang',
        'Log'               => 'Illuminate\Support\Facades\Log',
        'Mail'              => 'Illuminate\Support\Facades\Mail',
        'Paginator'         => 'Illuminate\Support\Facades\Paginator',
        'Password'          => 'Illuminate\Support\Facades\Password',
        'Queue'             => 'Illuminate\Support\Facades\Queue',
        'Redirect'          => 'Illuminate\Support\Facades\Redirect',
        'Redis'             => 'Illuminate\Support\Facades\Redis',
        'Request'           => 'Illuminate\Support\Facades\Request',
        'Response'          => 'Illuminate\Support\Facades\Response',
        'Route'             => 'Illuminate\Support\Facades\Route',
        'Schema'            => 'Illuminate\Support\Facades\Schema',
        'Seeder'            => 'Illuminate\Database\Seeder',
        'Session'           => 'Illuminate\Support\Facades\Session',
        'SoftDeletingTrait' => 'Illuminate\Database\Eloquent\SoftDeletingTrait',
        'SSH'               => 'Illuminate\Support\Facades\SSH',
        'Str'               => 'Illuminate\Support\Str',
        'URL'               => 'Illuminate\Support\Facades\URL',
        'Validator'         => 'Illuminate\Support\Facades\Validator',
        'View'              => 'Illuminate\Support\Facades\View',

),
```

After that, the database connection should be configured. The database configurations could be found in the “database.php” file in the “config” directory.

The “default” represents the default database connection name which is one of the connections that are specified in the “connections” array. The following shows the default sat to “mysql” and setting the connection host, username, database name, password, charset, and collection for the “mysql”.

```
'default' => 'mysql',

'connections' => array(

        'sqlite' => array(
                'driver'   => 'sqlite',
                'database' => __DIR__.'/../database/production.sqlite',
                'prefix'   => '',
        ),

        'mysql' => array(
                'driver'    => 'mysql',
                'host'      => 'localhost',
                'database'  => 'quotation',
                'username'  => 'root',
                'password'  => 'root',
                'charset'   => 'utf8',
                'collation' => 'utf8_unicode_ci',
                'prefix'    => '',
        ),

        'pgsql' => array(
                'driver'   => 'pgsql',
                'host'     => 'localhost',
                'database' => 'forge',
                'username' => 'forge',
                'password' => '',
                'charset'  => 'utf8',
                'prefix'   => '',
                'schema'   => 'public',
        ),

        'sqlsrv' => array(
                'driver'   => 'sqlsrv',
                'host'     => 'localhost',
                'database' => 'database',
                'username' => 'root',
                'password' => '',
                'prefix'   => '',
        ),

),
```

6.5 Installing External Packages

This section will discuss some packages that are used while developing this application, and how to install them.

6.5.1 Laravel 4 Generators

Laravel generators is a package that provides additional commands to Laravel's Artisan CLI to help developing applications faster. Instead of creating models, controllers, migrations, or seeds directly, this package contributes to ease the scaffolding process and to initiate the created component with data immediately. /37/

To install this package, the following line should be added to the “required-dev” or “require” in “composer.json” file.

```
"way/generators": "~2.0"
```

Moreover, run the following command to update composer’s dependencies.

```
composer update --dev
```

Finally, the following line needs to be added to the service providers array in the “app.php” file.

```
'Way\Generators\GeneratorsServiceProvider'
```

6.5.2 Faker

Faker is a PHP library that is used to produce fake data. It is useful for seeding database with data, stress-testing the database, or populate any persistence technology like XML, or JSON files with data. /38/

Installing Faker is similar to installing any library with composer, including the last package. The only step is to require that library by composer either by using `composer require` command or by adding it directly to “composer.json” file.

```
composer require fzaninotto/faker
```

6.5.3 JWT Authentication for Laravel

JWT-Auth is a Laravel package for simplifying dealing with JSON Web Token from the server-side perspective. The “0.4.*” version of the package should be installed as the Laravel version is 4.2. /39/

First, the package should be added directly to “composer.json” because of the specific version of it.

```
"tymon/jwt-auth": "0.4.*"
```

After that, composer needs to update its dependencies which will check for all the packages, update the existing ones if there is any new available version, and install the packages that have not been installed.

```
composer update
```

Then, the service provider of this Laravel package needs to be added to the “providers” array in the “app.php” configuration file.

```
'Tymon\JWTAuth\Providers\JWTAuthServiceProvider',
```

Optionally, classes aliases for the façades of this package could be included in the “aliases” array in the “app.php” configuration file.

```
'JWTAuth'    => 'Tymon\JWTAuth\Facades\JWTAuth',
'JWTFactory' => 'Tymon\JWTAuth\Facades\JWTFactory',
```

This package needs to be configured before using it, and running the following “artisan” command will create a configuration in the “app/config/packages” directory.

```
php artisan config:publish tymon/jwt-auth
```

The configuration file that has been set up is named “config.php” which exists in the “tymon/jwt-auth” directory in the “packages” directory. The configurations are represented as a PHP associated array with the following key-value pairs.

- “secret”: This is the string that is used to sign the token. For security reasons, the string is recommended to be generated by a cryptographically secure pseudo-random string generator. However, the package comes with a helper command to generate a new key.

```
php artisan jwt:generate
```

- “ttl”: Is the time that the token is valid for in minutes. Sixty minutes is the default.
- “refresh_ttl”: Is the time that the token can be refreshed within and it is in minutes as well. 20160 minutes is the default.
- “algo”: Is the hashing algorithm. “HS256” is the default.
- “user”: Specifies the full PHP namespace of the User model. “User” is the default value.
- “identifier”: Specifies the unique property of the user as the “sub” claim in the token payload. The default value is “id”.

- “reuired_claims”: Represents an array of the JWT required claims. The default value is the resulting array.

```
['iss', 'iat', 'exp', 'nbf', 'sub', 'jti']
```

- “blacklist_enabled”: Is used to invalidate tokens. The default is “true”.
- “providers”: Is an array of the user, JWT, authentication, and storage service providers. The default array looks like the following.

```
'providers' => [
    'user' => 'Tymon\JWTAuth\Providers\User\EloquentUserAdapter',
    'jwt' => 'Tymon\JWTAuth\Providers\JWT\NamshiAdapter',
    'auth' => 'Tymon\JWTAuth\Providers\Auth\IlluminateAuthAdapter',
    'storage' => 'Tymon\JWTAuth\Providers\Storage\IlluminateCacheAdapter'
]
```

6.5.4 Laravel DOMPDF Wrapper

This package allows the developer to write HTML, and it will parse the HTML DOM into a PDF that could be saved in a file, streamed for viewing, or downloadable. /40/

Similar to the last packages, this one does also follow these steps. Include it in the “composer.json” file, and update composer.

```
"barryvdh/laravel-dompdf": "0.4.*"
```

Add the service provider to the “providers” array.

```
'Barryvdh\DomPDF\ServiceProvider',
```

Moreover, optionally use the façade by adding an alias to the “aliases” array.

```
'PDF' => 'Barryvdh\DomPDF\Facade',
```

6.5.5 Intervention Image

This library helps to handle and to manipulate images in PHP by easing the process of creating, editing, and composing them. Even though this library is a standalone library,

it ships with service providers and façades for Laravel integration, which makes it a better compared to other image processing and manipulation PHP libraries. /41/

Installing this library is similar to the previous packages, include the following in "require" a section of the "composer.json" file, and run `composer update`.

```
"intervention/image": "^2.3"
```

Then, the service provider should be added to the "providers" array.

```
'Intervention\Image\ImageServiceProvider',
```

Optionally, an alias could be included in the "aliases" array.

```
'Image' => 'Intervention\Image\Facades\Image'
```

Although this package does not require any configurations, it is a good practice to create a configuration file for it. The following artisan command will create one in the "app/config/packages" as "intervention/image/config.php".

```
php artisan config:publish intervention/image
```

The configuration file should return an associated array with one key named "driver" which represents the used image driver that is used by the library. Two drivers could be chosen from, the "GD library" and "Imagick", and both are open source helper libraries to process images.

6.6 **ORM and Models**

Laravel comes with an ORM called Eloquent, which is and implementation of the Active Record design pattern to deal with database operations. It is a powerful tool as it maps a class to a data relation and saves much time having the outcome be an object that could be converted easily to any data structure needed.

Models classes extend the eloquent class and lives in "app/models" directory, which is not mandatory but recommended. They could be created manually, or using the generator package that has been installed before. The following is the artisan command that is used to create a new model called "customer".

```
php artisan generate:model Customer
```

It will create a new model called "Customer" and placed it in "Customer.php". The following is the content of the model.

```
<?php

class Customer extends \Eloquent {}
```

This is a valid model and does not require anything to work with it. Eloquent will assume that the table name of the User model is the lowered case plural name of the model which is “users” in this case. However, by defining a protected property named “table”, one can explicitly mention the name of the table that should be used.

Also, Eloquent will assume that the model table has a primary key column named “id”. However, this convention can also be overridden by defining a property named “primarykey”.

Laravel records the time of when the model has been saved or updated into the database table by defining “created_at” and “updated_at” columns. Those timestamps can be disabled by creating a property named “timestampes” and set it to false. It also provides a soft delete ability by using a “SoftDeltetingTrait” PHP trait. After that, “deleted_at” must be added to “dates” array, which is defined as a property of the model and represents the timestamp of when the deletion of the record happened.

Models that extend Eloquent will accept an array of attributes through its constructor. However, this could be a security issue if the user input passed into the model because it will make the model’s attribute modifiable. This security concern is called mass assignment, and Eloquent helps to protect against it by defining two array properties in the model called “fillable”, and “guarded”. The “fillable” property contains the attributes that can be mass assigned, while “guarded” is the opposite.

Laravel uses a validation API that helps to validate user input and provide meaningful and elegant error messages. Rules could be assigned directly to the “make” method that the “validator” class offers. However, a better approach is to define those rules in the model as they are actually part of it, which is considered a good practice.

The following is an associated array named “rules” which is showing the defined validation rules for the “Customer”.

```
public static $rules = [
    'name' => 'required',
    'email' => "unique:customers",
    'phone' => "unique:customers",
    'mobile' => "unique:customers",
    'address' => 'required',
];
```

The keys of the array are all or part of the “fillable” array values. The “name” and “address” keys are required, which means that the validator will reject the input if it does not contain a name and address. The “email”, “phone”, and “mobile” keys must be unique values in the “customers” table, and if not, the validator will reject the input.

These rules are basics for that specific model. However, other models could have more complex validation rules, and each key could have multiple rules. For example, the “Quote” model has the following rules.

```
public static $rules = [
    'title' => 'required',
    'floor' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'wall' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'ceiling' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'calculated_price' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'calculated_price_tax' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'final_price' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'final_price_tax' => ['regex:/^([1-9][0-9]*|0)(\.[0-9]{1,2})?$/'],
    'status' => 'required|in:done,created,canceled,processing',
    'start_date' => 'required|date',
    'template_id' => 'integer',
    'customer_id' => 'integer',
];
```

As observed, the “floor”, “wall”, “ceiling”, “calculated_price”, “calculated_price_tax”, “final_price”, and “final_price_tax” values must have a form of number with a maximum of two numbers after the decimal point according to the regular expression. The “status”, and “start_date” keys require more than one rule. The “status” should be present, and its value is one of the following: done, created, canceled, or processing. The “start_date” on the other hand is required and must be of a type date.

Relationships between models could also be created, and this feature is supportive while a model requests other model related data to be retrieved. These relationships represent the relationship between the database tables like one-to-one, one-to-many, and many-to-many relationships. For instance, the previous “Customer” model could have many “Quote” model. This relationship could be defined in a function that has the plural name of the “Quote” model representing the “many” relationship.

```php
public function quotes()
{
    return $this->hasMany('Quote');
}
```

The “hasMany” method is a defined method in the Eloquent class which indicate the relationship with the mode “Quote”, namely one-to-many. However, the reversed relationship could also be defined, and that relationship is defined in the target model, which is, in this case, the “Quote” model.

```php
public function customer()
{
    return $this->belongsTo('Customer');
}
```

The “belongsTo” method expresses the reversed relationship with the customer, namely many-to-one.

The result of the “Customer” model is the following class.

```php
<?php
use Illuminate\Database\Eloquent\SoftDeletingTrait;

class Customer extends \Eloquent {

    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];

    protected $fillable = ['name', 'email', 'phone', 'mobile', 'address', 'note'];

    protected $guarded = ['id'];

    public static $rules = [
        'name' => 'required',
        'email' => "unique:customers",
        'phone' => "unique:customers",
        'mobile' => "unique:customers",
        'address' => 'required',
    ];

    public function quotes()
    {
        return $this->hasMany('Quote');
    }

}
```

The following is the defined models that are used in the quotation application: Category, Customer, Extra, Image, Info, Material, Package, Quote, Service, Starting, Template, and User.

6.7 **Migrations and Seeding**

Laravel's migrations help versioning the database schema, which keeps the development team up to date with the latest schema. Those migrations use a schema builder API to help to build the application's schema.

Migrations live in "app/database/migrations" directory, and each one will have a timestamp bundled in the file name. That timestamp is created automatically by the framework to help to detect the order that the migration API should run or revert. Laravel's Artisan CLI has a command to create a migration file which will create a barebone migration class.

```
php artisan migrate:make create_users_table
```

The migration file name, in this case, will be a timestamp followed by "create_users_table". For instance, this is a valid migration name created by running the last command "2015_12_29_105218_create_users_table". Additional attributes could be added to the command to indicate the name of the database table.

```
php artisan migrate:make create_users_table --create=users
```

The command will take care of some of the schema builder boilerplate. However, Laravel generators package comes with more efficient commands when dealing with migrations. The command will detect the schema method that it should choose depending on migration name. For example, running the following command will apply the "create" method of the schema builder API.

```
php artisan generate:migration create_users_table
```

Running the previous command will create the migration file such as running the artisan command. However, it will help initiating the "create" method with "id" and timestamps for each created table. The following is the outcome of running the previous command.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateUsersTable extends Migration {

	/**
	 * Run the migrations.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('users', function(Blueprint $table)
		{
			$table->increments('id');
			$table->timestamps();
		});
	}


	/**
	 * Reverse the migrations.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('users');
	}

}
```

The “up” function is used by the migration API to detect its job when the migration runs, while the down is for reversing or rolling back the migration. The “Schema” API has a “create” method which will create a new table, a “drop” method which will drop a specified table, and “table” method which is used when adding or deleting columns from a table. “Blueprint $table” is mandatory, and it is called the table builder. It contains a variety of methods that represent column types. For example, boolean, double, integer, medium integer, small integer, string, text, timestamp, null, unsigned integers and more. Some of these methods are specific and have special meaning. For instance, “increments” is a method that will create a column that is an auto incremented integer that will act as a primary key, and “timestamps” will add “created_at” and “updated_at” columns.

The “generate:migration” command has an option called “fields” which helps to define the columns names and types. The following is the command that is used to create the “customers” table with its columns.

php artisan generate:migration create_customers_table --fields="name:string, email:string:nullable:unique, phone:string:nullable:unique, mobile:string:nullable:unique, address:text, note:text:nullable"

Resulting in the next migration.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateCustomersTable extends Migration {

	/**
	 * Run the migrations.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('customers', function(Blueprint $table)
		{
			$table->increments('id');
			$table->string('name');
			$table->string('email')->nullable()->unique();
			$table->string('phone')->nullable()->unique();
			$table->string('mobile')->nullable()->unique();
			$table->text('address');
			$table->text('note')->nullable();
			$table->timestamps();
			$table->softDeletes();
		});
	}


	/**
	 * Reverse the migrations.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('customers');
	}

}
```

However, the “softDeletes” is added manually which is also a special method that will add the “delete_at” column to the table for soft deletes timestamp.

For foreign keys, the table builder also provides special methods that will add references. For instance, the quotes table has “customer_id”, and “template_id” columns that represent foreign keys. The following is the “create_quotes_table” migration.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateQuotesTable extends Migration {

	/**
	 * Run the migrations.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('quotes', function(Blueprint $table)
		{
			$table->increments('id');
			$table->string('title');
			$table->text('note')->nullable();
			$table->decimal('floor', 12, 2)->nullable();
			$table->decimal('wall', 12, 2)->nullable();
			$table->decimal('ceiling', 12, 2)->nullable();
			$table->decimal('calculated_price', 12, 2);
			$table->decimal('calculated_price_tax', 12, 2);
			$table->decimal('final_price', 12, 2)->nullable();
			$table->decimal('final_price_tax', 12, 2)->nullable();
			$table->enum('status', array('done', 'created', 'canceled',
'processing'))->nullable();
			$table->date('start_date');
			$table->integer('template_id')->unsigned()->index();
			$table->foreign('template_id')->references('id')->on('templates')-
>onDelete('cascade');
			$table->integer('customer_id')->unsigned()->index();
			$table->foreign('customer_id')->references('id')->on('customers')-
>onDelete('cascade');
			$table->timestamps();
			$table->softDeletes();
		});
	}


	/**
	 * Reverse the migrations.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('quotes');
	}

}
```

To run the migration and create the table, the “migrate” Artisan CLI command should be executed.

```
php artisan migrate
```

Laravel provides an easy way for seeding a database with test data. Seed classes are stored in “app/database/seeds” and can have any desired name. However, logical names

are recommended. Each seed class must have a “run” function. Finally, the database seeder class should call the seeder classes using “call” method.

Seed classes use models or database query builder to populate the database with the test data. Those classes could be created manually, but the Laravel generators package could help easing this process by via the “generate:seed” artisan command. Creating a seed class for the “customers” table could be achieved executing the following command.

```
php artisan generate:seed customers
```

The generated class has the name “CustomersTableSeeder” which is the convention that the package use.

```
<?php

// Composer: "fzaninotto/faker": "v1.3.0"
use Faker\Factory as Faker;

class CustomersTableSeeder extends Seeder {

        public function run()
        {
                $faker = Faker::create();

                foreach(range(1, 10) as $index)
                {
                        Customer::create([

                        ]);
                }
        }

}
```

The Laravel generators package assumes that “Facker” is the library that is used to generate test data.

In the “run” function, a new instance of “Faker” is being created. After that, a “foreach” loop, which will run ten times, will execute the “create” method on the “Customer” model, which takes an associative array where keys are being the columns or the model fillable (mass assigned) properties, and the values are the test data of each one.

“Faker” provides a smart API that generates almost every type of data like names, addresses, phone numbers, digits, texts, paragraphs, sentences, words, dates, emails, URLs, IPs, files, images, barcodes and more. It also provides formatting options by defining formatter classes that ship with the library.

Using “Faker”, the array that is passed to the “create” method is the following.

```
                        Customer::create([
                                'name' => $faker->name,
                                'email' => $faker->email,
                                'phone' => $faker->phoneNumber,
                                'mobile' => $faker->phoneNumber,
                                'address' => $faker->address,
                                'note' => $faker->paragraph,
                        ]);
```

The “QuotesTableSeeder” is used to seed the “quotes” table.

```
<?php

// Composer: "fzaninotto/faker": "v1.3.0"
use Faker\Factory as Faker;

class QuotesTableSeeder extends Seeder {

        public function run()
        {
                $faker = Faker::create();

                foreach(range(1, 20) as $index)
                {
                        Quote::create([
                                'title' => $faker->word,
                                'note' => $faker->realText(100),
                                'floor' => $faker->randomFloat(2, 0, 100),
                                'wall' => $faker->randomFloat(2, 0, 200),
                                'ceiling' => $faker->randomFloat(2, 0, 100),
                                'calculated_price' => $faker->randomFloat(2, 0, 500),
                                'calculated_price_tax' => $faker->randomFloat(2, 0, 500),
                                'final_price' => $faker->randomFloat(2, 0, 500),
                                'final_price_tax' => $faker->randomFloat(2, 0, 500),
                                'status' => $faker->randomElement(['done', 'created', 'canceled',
'processing']),
                                'start_date' => $faker->date(),
                                'template_id' => rand(1, 10),
                                'customer_id' => rand(1, 20),
                        ]);
                }
        }

}
```

Some of the tables are created as pivot tables in the many-to-many which do not have models that represent mappers to them. For such case, the Laravel’s query builder is a suitable option. For instance, quotes have starting tools and plans values, and the next seeder class demonstrates creating test data for the “quote_starting” pivot table.

```php
<?php
use Faker\Factory as Faker;

class QuoteStartingTableSeeder extends Seeder {

        public function run()
        {
                $faker = Faker::create();

                foreach(range(1, 20) as $index)
                {
                        DB::table('quote_starting')->insert([
                                'quote_id' => rand(1, 20),
                                'starting_id' => rand(1, 10),
                                'starting_checked' => rand(0,1),
                                'starting_name' => $faker->sentence,
                                'starting_price' => $faker->randomFloat(2, 10, 100),
                        ]);
                }
        }

}
```

6.8 **Routing**

This section discusses the routing mechanism and the system's defined routes. Routes are specified mostly in the "app/routes.php" file. Using the "Route" API which has methods that define the HTTP method, the URI, and a closure or a directed method on a controller. There is a particular type of routes called the "resource" route which is a RESTful route for a resource, and it will create the following paths, each accompanied with its HTTP verb and route name.

Verb	Path	Action	Route Name
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{resource}	show	resource.show
GET	/resource/{resource}/edit	edit	resource.edit
PUT/PATCH	/resource/{resource}	update	resource.update
DELETE	/resource/{resource}	destroy	resource.destroy

Figure 22. Resource route

“/resource/create” and “/resource/{resource}/edit” are for viewing a form that will take care of creating or editing the resource consecutively. However, these methods are not needed as the front-end will take care of that. The routing API allows a subset of the previous methods to be defined. For example, the “customer” resource route is defined with index, store, update, destroy, and show end points.

```
Route::resource('customer', 'CustomersController', array('only' => array('index', 'store',
'update', 'destroy', 'show')));
```

The first parameter of the resource route is the name of the resource, the second is the controller that will handle the operations, and the last is optional and has the subset of the desired methods.

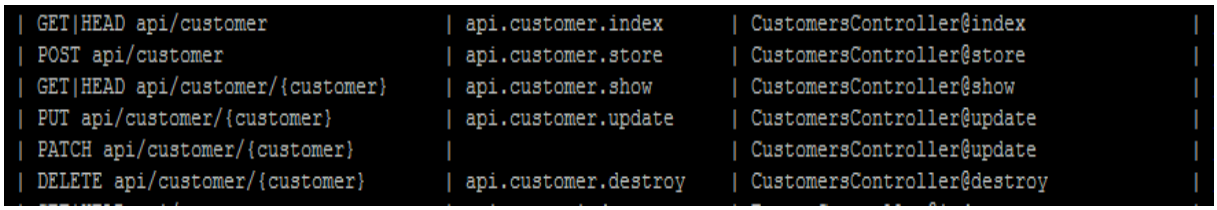


Figure 23. Customer resource routing

The “api” prefix is defined using the “group” method. This method groups a set of routes and applies specific rules to them. For instance, the prefix “api” is defined on almost all of the application routes.

```
Route::group(array('prefix' => 'api'), function () {
        // Defined routes
});
```

Some routes are defined by methods named get, post, put, or delete. These methods represent the HTTP verb that is applied or accepted on these routes. For instance, to get customer’s quotes, the following route is defined.

```
Route::get('customer/quotes/{id}', array('as' => 'api.customer.quotes', 'uses' =>
'CustomersController@quotes'));
```

This route accepts a GET HTTP request. The “{id}” is a route parameter that represents the customer id. The “get” method accept an array as a second parameter where the “as” is the route name, and “uses” is the method name that will handle the request on the controller.

6.9 **Filtering Requests**

Filtering requests is a mechanism for restricting access to specific or all routes. Laravel's ships with several sets of filters like authentication, and CSRF filters. Filters could be defined in the "app/routes.php" or "app/filters.php" file.

The "app/filters.php" file contains a "before" and "after" methods, where they are used to do any work before or after the request. As the back-end is being built as an API, the API should enable the CORS. This is done by placing some HTTP headers before accepting the request.

```
App::before(function($request)
{
	// Enable CORS
	header("Access-Control-Allow-Origin: *");
	header('Access-Control-Allow-Credentials: true');

	if (Request::getMethod() == "OPTIONS") {
		// The client-side application can set only headers allowed in Access-Control-
Allow-Headers
		$headers = [
			'Access-Control-Allow-Methods' => 'POST, GET, OPTIONS, PUT, DELETE',
			'Access-Control-Allow-Headers' => 'X-Requested-With, Content-Type, X-
Auth-Token, Origin, Authorization'
		];
		return Response::make('You are connected to the API', 200, $headers);
	}
});
```

However, when the application is in production, the "Access-Control-Allow-Origin" header should replace the "*" by the web application domain that is consuming this API.

6.10 **Controllers and Business Logic**

Controllers act like an orchestration level of the application. They assemble business logic in class and use patterns like dependency injection to form a much better-structured application that follows software design best practices.

Controllers live in "app/controllers" directory. However, this can be changed by registering another desired directory in the "classmap" of the "composer.json" file. Controllers can be created manually or using Artisan CLI, and all of them must extend "BaseController" class that lives in the same directory and extends "Controller" class by itself.

Using the Artisan command to create a new controller has an advantage of specifying the exact same set of methods that are allowed when creating the resource route.

```
php artisan controller:make CustomersController --only=index,store,update,show,destroy
```

This will create a controller class in “app/controllers” named “CustomersController”, and scaffold it with five methods: index, store, update, show, and destroy.

```
<?php

class CustomersController extends \BaseController {

	public function index()
	{

	}

	public function store()
	{

	}


	public function show($id)
	{

	}


	public function update($id)
	{

	}

	public function destroy($id)
	{

	}

}
```

The “index” method should retrieve a list of all the customers using the “Customer” model.

```
	/**
	 * Display a listing of customers
	 *
	 * @return Response
	 */
	public function index()
	{
		return Customer::all();
	}
```

Laravel’s Eloquent models are smart enough to explicitly return the response as JSON, which makes the process even faster.

Saving a new customer object is handled in the “store” method.

```
/**
 * Store a newly created customer in storage.
 *
 * @return Response
 */
public function store()
{
	$data = Input::all();

	$validator = Validator::make($data, Customer::$rules);

	if ($validator->fails())
	{
		return $validator->messages();
	}

	return Customer::create($data);
}
```

First, the “all” method on “Input” handles receiving the data while dealing with POST requests, which will be sent as JSON. These data are saved in a variable called “$data”. Then validating the data by using the “make” method on the “Validator” which takes two parameters, the inputs and the rules, and store than in an object called “$validator” which will be used to check if the data is valid. The “fails” method of “$validator” will return true in case the data is not valid. Otherwise, it will return false. “Validator” provides descriptive error messages that could be returned in case the validation failed. These messages are returned using “messages” method. Finally, and if the data is valid, the customer will be created using the “create” method on the “Customer” model which is an Eloquent method, and returned as JSON.

The “show” method is used to retrieve a customer object by its id.

```
/**
 * Display the specified customer.
 *
 * @param  int  $id
 * @return Response
 */
public function show($id)
{
	return Customer::findOrFail($id);
}
```

The “findOrFial” method is another Eloquent method which, as the name indicates, tries to find a row, in this case, a “Customer” row, by its id, and fails with “404 could not found” error if it could not.

The “update” method is used for updating a customer.

```
/**
 * Update the specified customer in storage.
 *
 * @param  int  $id
 * @return Response
 */
public function update($id)
{
	$customer = Customer::findOrFail($id);

	$data = Input::all();

	$validator = Validator::make($data, Customer::rules($id));

	if ($validator->fails())
	{
		return $validator->messages();
	}

	$customer->update($data);

	return Response::json(['response' => true]);
}
```

It will accept an id which is passed in the URL and tries to find the customer that is associated with it, and if it could not, it will return a “404 could not found” error. After that, the steps of getting the input data and validate it are the same while creating a new customer. Then, the customer’s object that is stored in “$customer” variable has an “update” method that will update the associated row in the database with the entered data. Finally, a customized JSON is created using the “json” method on the “Response” class and returned.

The last method is the “destroy” method which will delete a customer.

```
/**
 * Remove the specified customer from storage.
 *
 * @param  int  $id
 * @return Response
 */
public function destroy($id)
{
	Customer::destroy($id);

	return Response::json(['response' => true]);
}
```

The method will receive a customer id through the URL and runs the “destroy” method on the “Customer” model which will soft delete the customer row by setting the “deleted_at” to the corresponding timestamp of when the deletion happened. This behavior

occurs due to the soft delete operations on the model itself, but the “destroy” method usually deletes the desired resource permanently.

The controller is not restricted to these methods only; it can extend as it goes. For instance, the “CustomerController” class define a method calls “quotes” which will return a list of the quotes that are associated with a customer by his id.

```
        public function quotes($id)
        {
                return Customer::with('quotes')->findOrFail($id);
        }
```

The “with” method on the “Customer” method is a special one that helps reduce the number of queries that are needed to attain such result. It will include, depending on the model relationship with the “Quote” model that is defined in its class, the list of the quotes with the customer information as well.

Moreover, the relationship between models could be hard to achieve when attempting to save the object and its relational data. For instance, while storing a “template” object, “packages” and “startings” ids needs to be saved in their pivot tables.

```
    public function store()
    {
            $data['title'] = Input::get('title');
            $data['description'] = Input::get('description');
            $data['startings'] = Input::get('startings');
            $data['packages'] = Input::get('packages');

            $validator = Validator::make($data, Template::$rules);

            if ($validator->fails())
            {
                    return $validator->messages();
            }

            return DB::transaction(function () use ($data) {
                    $template = Template::create($data);
                    $template->startings()->sync($data['startings']);
                    $template->packages()->sync($data['packages']);
                    return Template::with('Packages')->with('Startings')-
>findOrFail($template->id);
            });
    }
```

This is possible due to the relational methods that have been defined in the “Template” model, and the “sync” method. However, this process could go wrong at some point, and the data would end up half-saved. For instance, the template information could be saved, but for some reason, the “sync” operations failed. This will produce inconsisten-

cy that is avoidable by using the “transition” method on the query builder API, which will wrap the process in a database transition that will guarantee that all or none of the operations will occur.

6.11 **Images as Base64**

When building APIs, the most efficient way of transferring files is by encoding and decoding. The quotation system uses Base64-encoded files between the API and the web application. The API will receive an image, check the type and the size, save a reference in the database, decode the image, and finally save it on the disk permanently using the Intervention Image library.

For instance, updating the logo of the company that is using the quotation system.

```
        /**
         * Update the logo.
         *
         * @return Response
         */
        public function logoUpload()
        {
                $info = Info::findOrFail(1);
                $data = Input::all();

                if (($data['filetype'] == 'image/png' || $data['filetype'] == 'image/jpeg') &&
($data['filesize'] <= 2000000)) {
                        if ($data['filetype'] == 'image/png') {
                                $logo = 'img/logo.png';
                        } else if ($data['filetype'] == 'image/jpeg') {
                                $logo = 'img/logo.jpeg';
                        }
                        Image::make((string) ($data['base64']))->save($logo);

                        $info->logo = $logo;
                        $info->update($data);

                        return Response::json(['response' => 'accepted']);
                }

                return Response::json(['response' => 'not-accepted']);
        }
```

`Image::make((string) ($data[‘base64’]))->save($logo)` is the part where the Intervention Image library decode and save the logo image on the desk in the “public/img” directory.

To retrieve the logo image, a reference to the image must be returned to identify the required image, encode the image, and send it in JSON.

```php
/**
 * get the logo.
 *
 * @return Response
 */
public function getLogo()
{
        $info = Info::findOrFail(1);

        $data = (string) Image::make($info->logo)->encode('data-url');

        return Response::json(['img' => $data]);
}
```

6.12 **Testing the API**

To test the API, Postman Google's Chrome Application, which is also a standalone desktop application, is used. Postman has a simple, yet powerful GUI platform which has testing, documenting, and monitoring features for API development. /42/

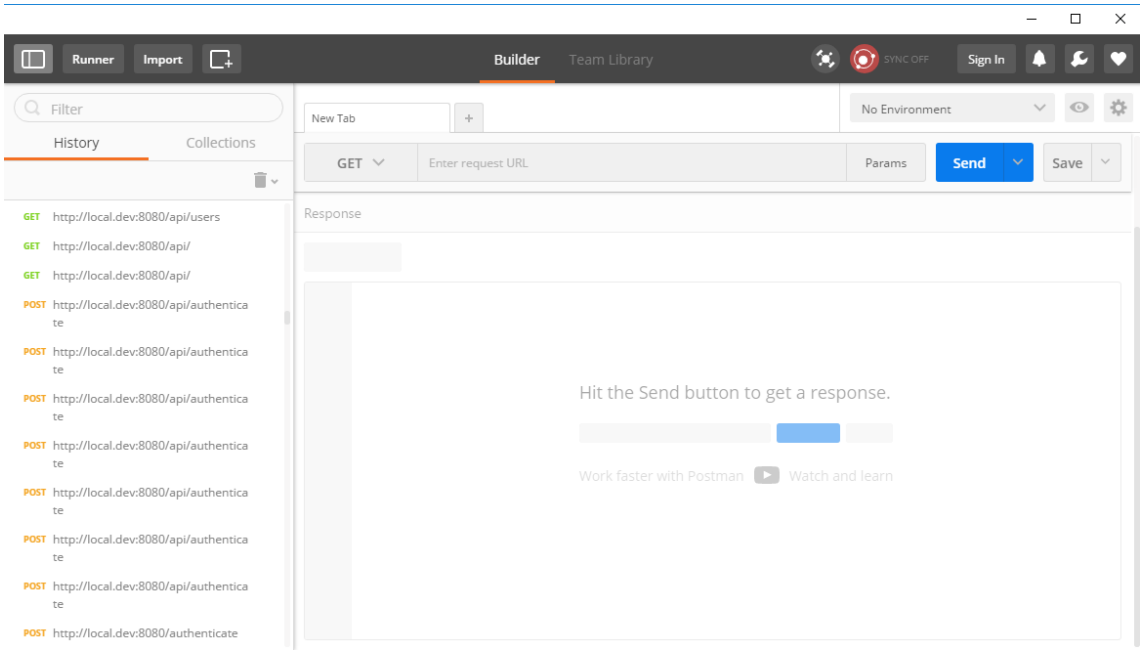


Figure 24. Postman Application GUI

The "customer" endpoints will be tested, including getting a list of all the customers or one by its id, creating, updating, and soft deleting a customer.

To get a list of customers, a GET request must be sent to `api/customer` end point. The response should be an array of customers' objects.

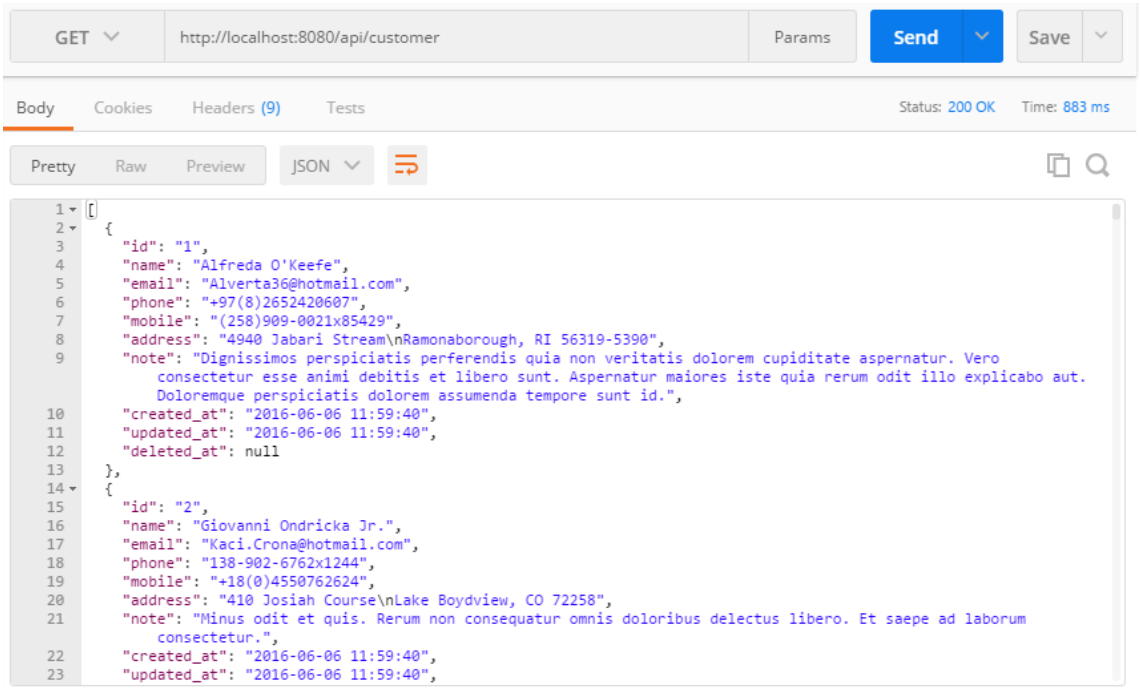


Figure 25. Array of customers' objects

Getting the customer's information which his id is one by sending a GET request to `api/customer/1`.

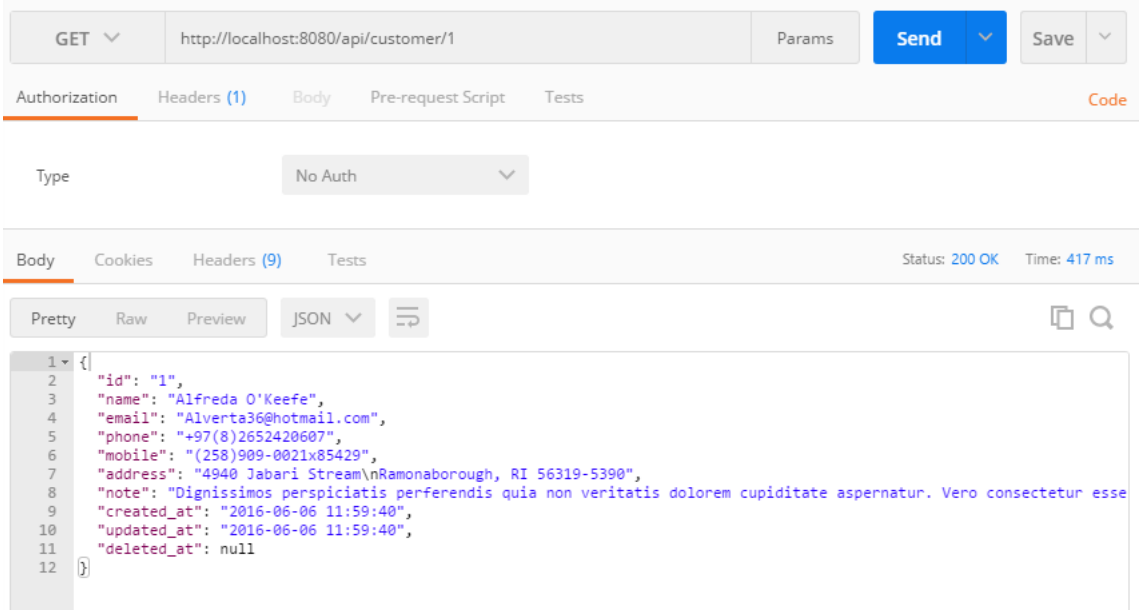


Figure 26. Customer's response object

To create a new customer, a POST request with JSON object containing the required data should be sent.

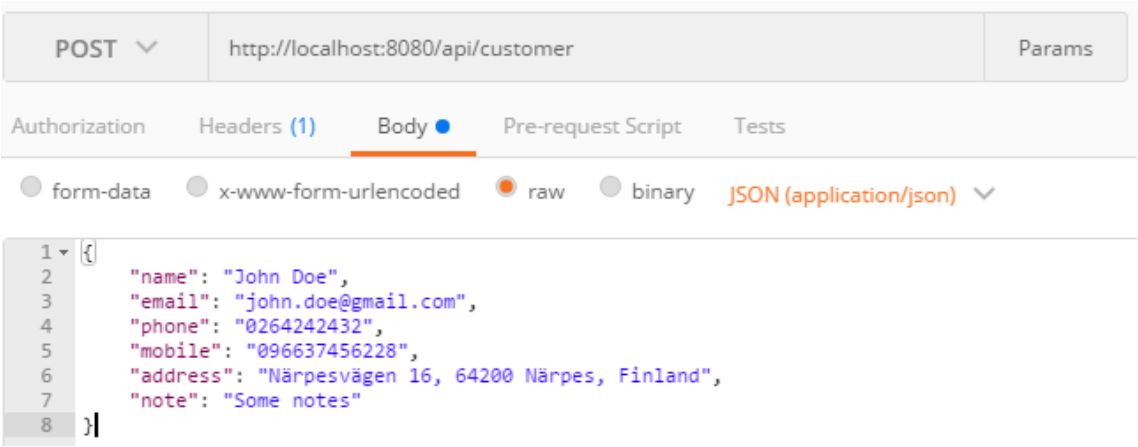


Figure 27. Create customer POST request

The response will be the created customer object.

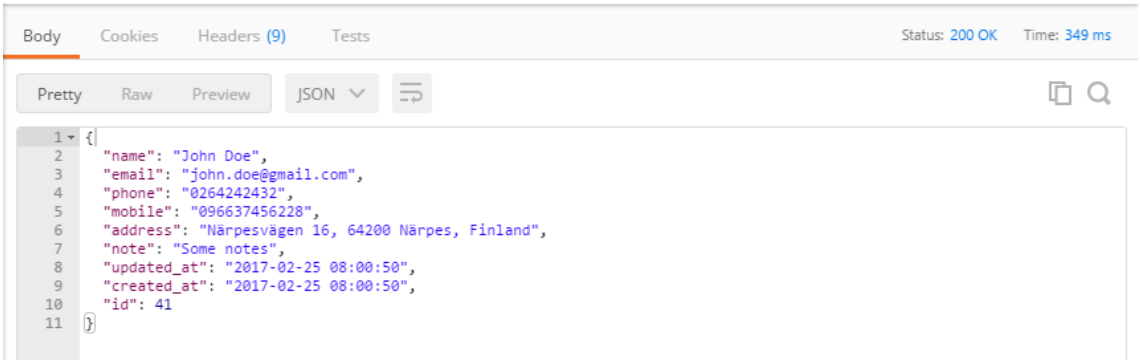


Figure 28. Created customer object response

To demonstrate what will be the response if a required field is missing, the “name” from the previous POST request is intentionally removed. The following expressed object was returned.

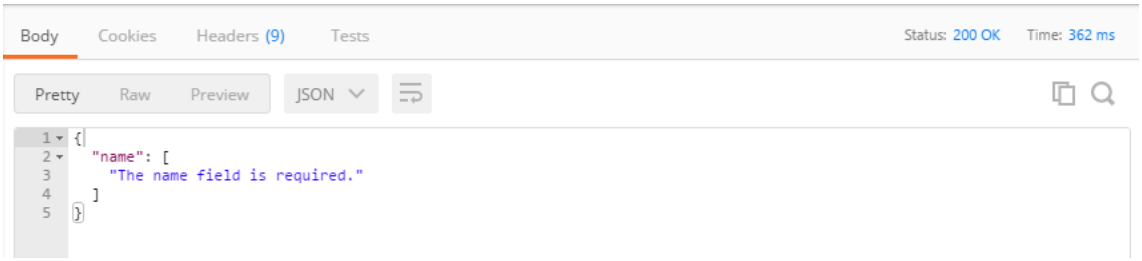


Figure 29. Response error message

To illustrate the update process, a PUT request is sent to `api/customer/41` to update the previously created customer, with the new data that are needed to be saved.

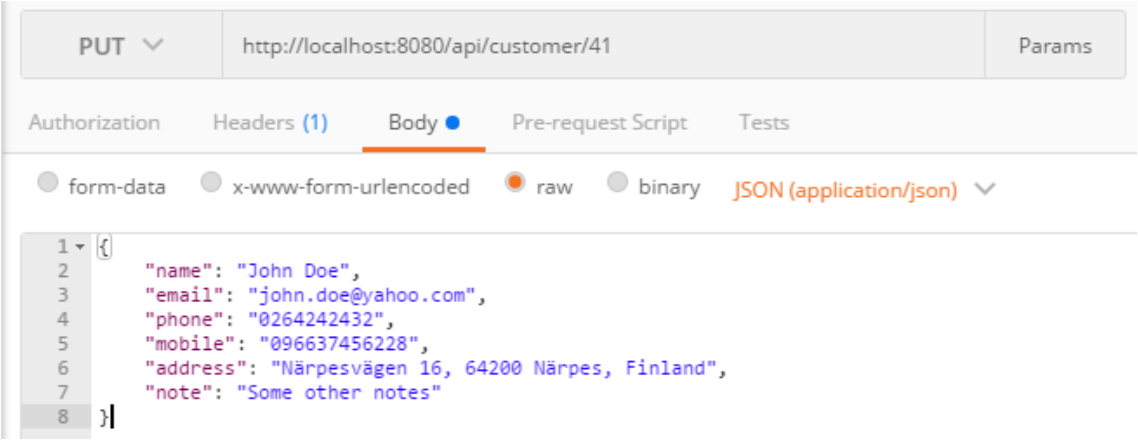


Figure 30. Customer's update PUT request

A JSON object with “response” key and “true” value is returned.

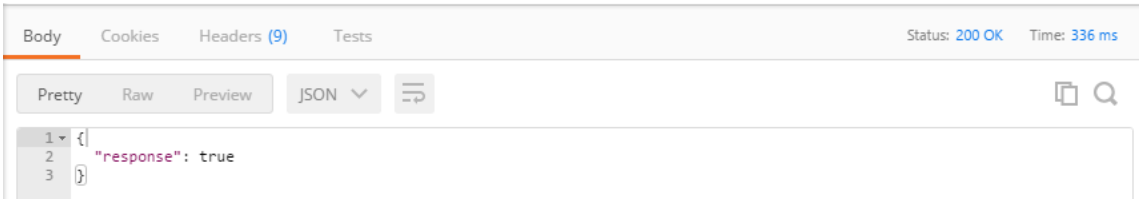


Figure 31. Update customer's information response object

Finally, to test the delete customer functionality, a DELETE request is sent to `api/customer/41` which will delete the previously created customer object that holds the id of 41. A JSON object with “response” key with “true” as a value is returned.

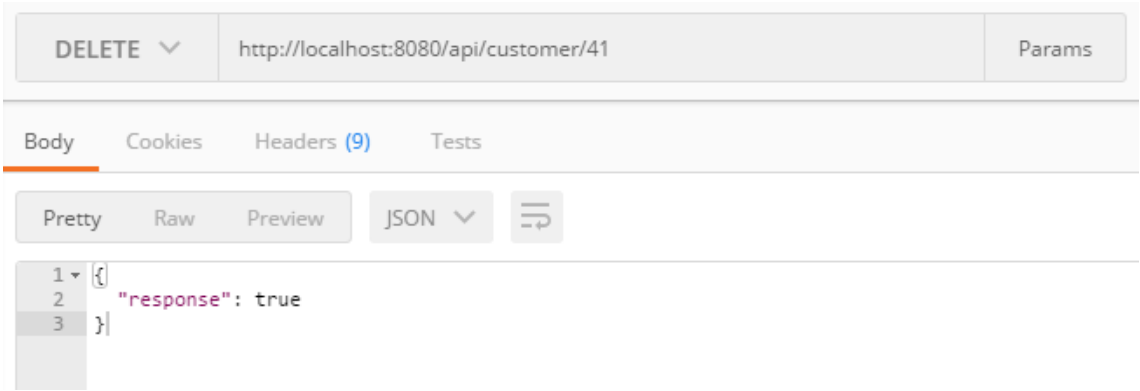
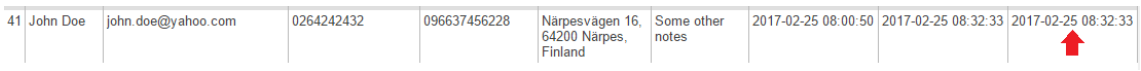


Figure 32. Delete request

As mentioned before, the customer is softly deleted by setting “deleted_at” column on that customer row to the timestamp of when the operation occurred. The following shows the value of the “deleted_at” timestamp was assigned after the previous DELETE request.



41	John Doe	john.doe@yahoo.com	0264242432	096637456228	Närpesvägen 16, 64200 Närpes, Finland	Some other notes	2017-02-25 08:00:50	2017-02-25 08:32:33	2017-02-25 08:32:33

Figure 33. Soft delete illustration

6.13 **Performance**

Caching and optimizing code are some options that are mostly considered while dealing with application performance. The process could be excessive and time-consuming, but some common practices should be applied at least, especially in production.

The following is a list of performance optimization practices that could be implemented by any back-end application or unique to Laravel.

- Database indexes: They are helpful when retrieving data using the SELECT statement, but might slow the code while inserting, updating, or deleting data.
- Database normalization: This should be by architecture, and keeping more frequent requested data at one table, while less demanded in another.
- Database Engine: Using the right engine that suits the application needs. In the case of the quotation application, MySQL InnoDB was the best fit.
- Composer's autoload: Composer is using PSR-4 Autoloader recommendation from PHP-FIG by the time of this paper was written. Optimizing the autoload mechanism could be achieved by running `composer dumpautoload -o`.
- Class map optimization: The back-end ended up with a few hundreds of files after all. One way to optimize performance is to combine the essential used files (such as filters and service providers) on each request into one file. Laravel's provide a "config/compile.php" file which accept the files that could be combined in its array, and by running `php artisan optimize`, the framework will combine and optimize these files.
- Caching: Laravel supports Memcached and Redis for back-end caching with little to no configurations where they are stored in "app/config/cache.php" file. Storing data that are desired to be cached is achievable using the following.

```
Cache::put('key', 'value', $minutes);
```

The key is a unique key that will be used to retrieve the value during the lifetime that is specified in minutes using "$minutes". For instance, when retrieving a list

of customers, it is recommended to save the data in cache for later use, and the next time a user requests the data, a check will run on the cache to retrieve it first. If the data does not exist or has expired, it will be retrieved again from the database.

```
if (Cache::has('key')) {
        $value = Cache::get('key');
}
```

Laravel provides an easier mechanism.

```
$value = Cache::remember('customers', $minutes, function()
                                                    {
        return DB::table('customers')->get();
});
```

- Code profiling: This is an obvious step, but many developers tend to forget it. There are always improved, and more performant algorithms, the most important is to choose one that will suit the situation better.

7 SYSTEM IMPLEMENTATION: FRONT-END AS SPA

This chapter discusses building the front-end web application that will consume the API and visualize what has been done so far.

7.1 **Installing Dependencies**

The front-end development is not a simple process like it used to be. Dependency managers, package managers, and automation systems are must have tools. Therefore, before starting the development, some of those tools need to be installed.

Node package manager (npm) is a dependency package manager that is used to install and update the application's dependencies like libraries and frameworks. It is installed with Node.js but can be mounted manually. However, npm should update to the latest version always before use. Ironically, npm is treated as just another package and maintained by itself, which is why npm is used to update npm. /43/

```
npm install npm@latest -g
```

Now npm could be used to install the other dependencies, including an automation and task runners. Automation tools help to automate repetitive tasks such as minifying and bundling JS and CSS files, running unit tests, running code analysis, real-time browser sync and refresh, and much more, saving lots of time. /44/

Gulp is an automation toolkit that uses JavaScript to do the previous tasks. Because of the Gulp configuration nature, it is an excellent fit and a better option than other toolkits like Grunt that uses JS objects for configurations. Installing gulp requires Gulp CLI to run globally.

```
npm install gulp-cli -g
```

Then gulp can be installed locally and as a development tool, and "gulpfile.js", which will contain gulp configurations, should be created.

```
npm install gulp -D
touch gulpfile.js
```

Now that npm and gulp are ready, the libraries and frameworks could be installed and minified easily for production.

The web application was built as a SPA, and AngularJS is the framework that is used to achieve this. Using npm, AngularJS could be installed in a directory called "node_modules" which is where all the dependencies are stored. In production, gulp is used to move these dependencies to "vendor" directory.

```
npm install angular
```

Installing other modules, directives, and libraries follow the same process. The following is a list of all the dependencies.

- Angular Route ("ngRoute"): For routing and linking URLs to views and controllers. /45/
- UI Bootstrap: Are Twitter's Bootstrap, which is an UI front-end framework, components written in AngulaJS. /46/
- Angular Messages ("ngMessages"): A directive that is used to show/hide messages based on a JS object's key/value. Mainly used to display messages for form fields. /47/
- Satellizer: It is a token-based authentication module. In the quotation system, it is used to while dealing with authentication when sending requests to the back-end. /48/
- ngStorage: This module helps while working with local and session storage. /49/
- Angular Translate: This module eases the process of creating multilingual web applications by using the data binding that AngularJS provides. /50/
- Angular Base64 Upload: Converts files into Base64 encoded models. /51/

Finally, to ease the process of building an UI, AdminLTE template theme was chosen as a starting point due to the features that it provides such as mobile first responsive design (using Twitter bootstrap), valid HTML and CSS syntax, lightweight, and support for major browsers. Not mentioning that most of the components and plugins have AngularJS equivalents or could be wrapped in directives to suit AngularJS needs.

7.2 **Application Setup and Configuration**

The application starting point is the "index.html" file in the root of the application directory. It includes the necessary CSS and JavaScript files for the application to perform, in addition to the container "div" with "ng-view" directive. That directive is used by the "ngRoute" module to render the views that associated with the visited URL.

The “vendor/app.js” file contains the father AngularJS module and the rest of the application configuration. First, a new AngularJS module called “quote” is instantiated with all the dependencies and assigned to a variable called “app” to be used as a reference to when defining constants, and configurations.

```
var app = angular.module('quote', [
          'ngRoute', 'ui.bootstrap', 'ngMessages',
          'angularUtils.directives.dirPagination',
          'satellizer', '720kb.datepicker',
          'ngStorage', 'angular-intro', 'pascalprecht.translate']),
```

Then, an AngularJS constant named “API” is defined, which holds the URL of the back-end API that the web application will send requests to. AngularJS constants life scope is through the whole module which makes them accessible by all the services.

```
app.constant('API', 'http://localhost:8080/public/api/');
```

The configuration in AngularJS application is defined in the “config”, which takes a function like the following.

```
app.config(function ($routeProvider, $translateProvider, $authProvider) {});
```

Currently, the translation/multilingual and the routing functionalities configurations are handled. For the translation/multilingual settings, a method to determine from where the translations will be loaded should be added, and another one for sanitizing the language specific texts.

Language translations could be defined as an object to the “translations” method on the translation service provider “$translateProvider”, which is a good choice for small applications. However, this is not helpful when dealing with medium to large applications. Therefore, the “useStaticFilesLoader” method was chosen, which will load the translations from static JSON files. To determine where the module should load these files, the method accepts a JavaScript object that represents the options.

```
// Translation
$translateProvider.useStaticFilesLoader({
    prefix: 'languages/',
    suffix: '.json'
});
```

The “prefix” is the directory that contains the translation files in perspective to the loader HTML file; “index.html”. The “suffix” is how the file name ends and could be anything, but must end with “.json”.

For security reasons, the translation module must provide a sanitizing strategy that will be used to prevent any injected HTML, or JavaScript from being executed. To specify the strategy, the “useSanitizeValueStrategy” method is used.

```
$translateProvider.useSanitizeValueStrategy('escape');
```

The routes are configured using the “$routeProvider” and chained “when” methods. Each “when” appearance describes an URL, with the associated view and controller.

```
.when('/customers', {
    templateUrl: components + 'customers/index.html',
    controller: 'CustomersController',
    controllerAs: 'customers'
})
```

The ‘/customers’ is the partial URL, the “templateUrl” is the path to the view, and the “controller” is the used controller. The “controllerAs” is optional, and it is an approach to replace the “$scope” object that keeps the view, and the model synced (data binding). This makes the controllers clean and “$scope”-free while maintaining almost every feature available.

The URL in “when” accepts parameter as well. Parameters are helpful for when passing information between views for example. They can be defined by prepending colon “:” to the parameter name.

```
.when('/customer/:id', {
    templateUrl: components + 'customers/customer.html',
    controller: 'CustomerController',
    controllerAs: 'customer'
})
```

To set a default URL which will handle redirecting in the case of hitting non-existing ones, the “otherwise” method on the “$routeProvider” is used.

```
.otherwise({
   redirectTo: '/auth'
});
```

For Satellizer’s configuration, only the API route that the JWT should be retrieved from is asked.

```
$authProvider.loginUrl = 'http://46.101.120.7/quoting-system/public/api/authenticate';
```

7.3 Application Components

As discussed in Chapter 5, the front-end web application is structured as components. This is applied to business components like “customers”, and visual or UI components like the user menu. Each component has at least one view and one controller associated with it. However, some might have more than one view and controller. For instance, the “packages” component has two views and two controllers associated with these views.

Even though each component might have its services file, some components could use the other ones’ services. Each service is an AngularJS factory returns a JavaScript object, which consists of API calls and other functionalities unique to that component.

There are three UI components: “header” for the header menu, “aside” for a side menu, and “footer” for the page footer. The header menu contains the user-specific menu, such as a link to user’s profile, log out, and changing the language.

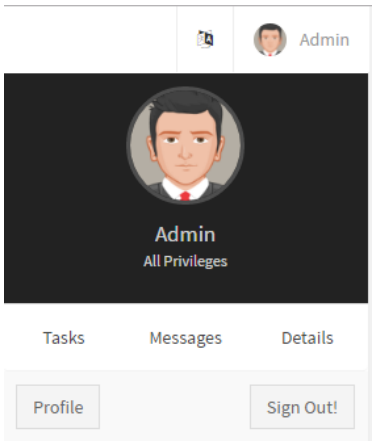


Figure 34. User specific menu

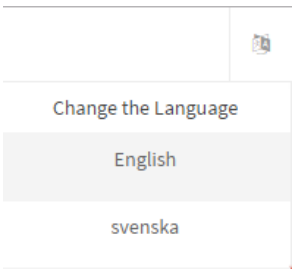


Figure 35. Language menu

While the header menu is user specific, the side menu is application specific. The side menu represents a navigation menu to the application components.

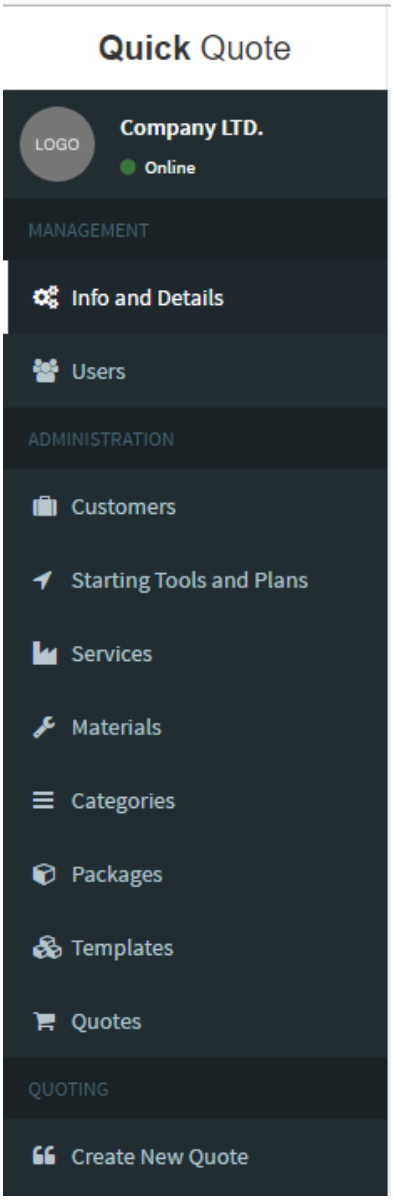


Figure 36. Side navigation menu

The view “aside.html” of the side menu is associated with a controller “AsideController”. The view contains an “aside” HTML element has a “ng-controller” attribute directive that defines a controller-as syntax.

```
<aside class="main-sidebar" ng-controller="AsideController as vm">
...
</aside>
```

In the “aside”, a “section” with a “div” that represents the logo image, the company name, and the online status is defined, followed by a “ul” element that describes the navigation menu. The logo image “img” element binds a logo variable from the controller in “ng-src” for late binding.

```
<img ng-src="{{vm.logo}}" class="img-circle" alt="Logo Image">
```

The navigation list is defined as a set of “li” elements.

```
<li class="header">MANAGEMENT</li>
<li>
    <a href="#/details">
        <i class="fa fa-gears"></i> <span>Info and Details</span>
    </a>
</li>
<li>
    <a href="#/users">
       <i class="fa fa-users"></i> <span>Users</span>
    </a>
</li>
<li class="header">ADMINISTRATION</li>
<li>
    <a href="#/customers">
        <i class="fa fa-suitcase"></i> <span>Customers</span>
    </a>
</li>
<li>
    <a href="#/starting-tools">
        <i class="fa fa-location-arrow"></i> <span>Starting Tools and Plans</span>
    </a>
</li>
...
```

The “#” is mandatory while dealing with anchor’s “href” in AngularJS as it triggers the “ngRoute” module.

The controller is defined in a file named “component/aside/controller.js” as the following.

```
(function () {
    "use strict";

    var app = angular.module('quote');

    app.controller('AsideController', ['$http', 'API', '$rootScope', function ($http, API,
$rootScope) {
        ...
    }]);
}());
```

As of all the JavaScript code pits, everything is wrapped in an IIFE and in strict mode. To define a controller, a reference to the “quote” module is returned and assigned to a variable named “app”. Then the controller is defined using the “controller” method which takes two parameters, the name of the controller, and an array or a function that contains the controller functionalities. The “$http” is an AngularJS built-in service that provides an API to ease up the process of making HTTP requests. The “API” an Angu-

larJS constant that has been defined in the “app.js” previously. The “$rootScope” is the father scope object and all other scopes are descendants of it.

The “AsideController” will call the API requesting the image logo using the “$http” AngularJS service.

```
app.controller('AsideController', ['$http', 'API', '$rootScope', function ($http, API,
$rootScope) {
    var vm = this;

    $http({
            method: 'GET',
            url: API + 'info/logo'
        }).then(function (response) {
            vm.logo = response.data.img;
        }, function (error) {
            console.log(error);
        });
}]);
```

First, creating a variable named “vm” that hold the “this” keyword, which references the controller function. Then, executing the “$http” call which accepts an object of options like the method name (“GET”), and the URL, returning a promise that is handled using the “then” method. The “then” method accepts two parameters that are callbacks functions, the first handles the success of the promise, while the second handles the fail. In that example, the success will assign the Base64 encoded logo image to “vm.logo” which is rendered as seen in the view previously. The API call should be moved to an AngularJS service or factory, but due to the simplicity of this controller, it was kept.

The content of the application is then rendered according to the URL, which will be resolved with the “ngRoute” as mentioned before. The landing page of the application is the details page, which is the page that contains quick statistics, general information about the company, and uploading logo image.

The “details” component consists of a view, a controller, and a service. The company’s details are molded in an HTML form, with un update button. On the other side, the upload logo form, which hence the name will upload a logo image.

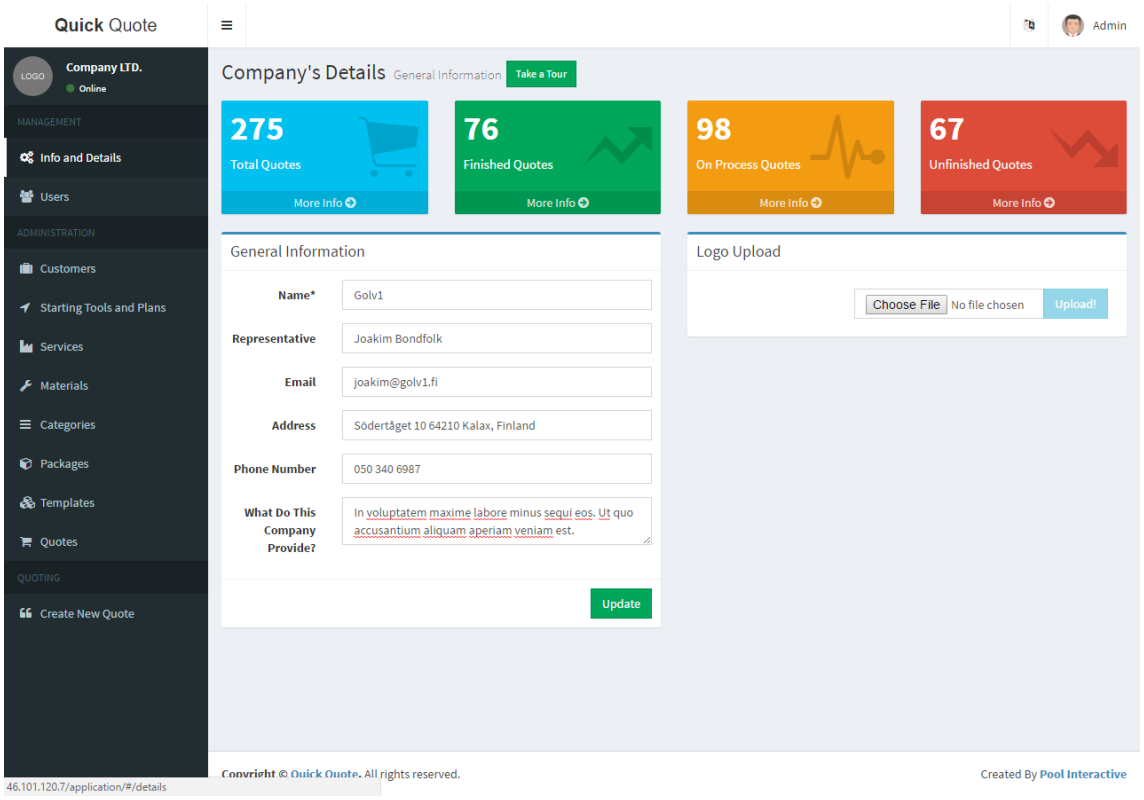


Figure 37. Details landing page

The update information form should have a name attribute, which will be used to access its input fields for validation. It should also have a "ng-submit" attribute, which is an AngularJS directive that will call a function on form submitting.

```
<form class="form-horizontal" name="updateDetails" role="form"
      ng-submit="details.submitUpdateDetails(updateDetails.$valid)" novalidate>
...
</form>
```

The "details" in "ng-submit" is the controller-as, which is how the view can have access to its controller. The "submitUpdateDetails" is a method that is defined in the controller and accepts a Boolean value. The "updateDetails.$valid" is an expression that will be translated to a Boolean depending on the form validity. The "novalidate" is an HTML5 attribute to prevent the browser from validating the form using the default validation and leave the process to AngularJS.

The process of validating the "input", "select", and "textarea" HTML elements is somehow similar with minor changes. For instance, the company's name field in the previous details update form is defined as the following.

```
<div class="form-group"
     ng-class="{ 'has-error': updateDetails.detailsName.$touched &&
updateDetails.detailsName.$invalid }">
       <label class="col-sm-3 control-label"><span>Name</span>*</label>

       <div class="col-sm-9">
              <input autocomplete="off" name="detailsName" type="text" class="form-control"
                     placeholder="Company's name" required
                     ng-model="details.company.name">

              <div class="text-danger" ng-messages="updateDetails.detailsName.$error"
                   ng-show="updateDetails.detailsName.$touched">
                     <div ng-messages-include="shared/messages.html"></div>
              </div>
       </div>
</div>
```

The previous structure is how Twitter Bootstrap assembles its input fields to be the appear on Figure 37.

The “ng-class” is an AngularJS directive used to dynamically add/change classes to an element using expressions that translate to Booleans. For instance, the “has-error” is a class defined by Bootstrap, which will mark the input in red color. The expression “updateDetails.detailsName.$touch” is the form name, followed by the input name, followed by the special AngularJS property “$touch” which translates to a Boolean value having true when the input was focused in and out. The other expressions “updateDetails.detailsName.$invalid” has the same first two parts while the last represents the invalidity of the input field, and return a Boolean value as well.

The “ng-model” attribute of the “input” is an AngularJS directive which binds to the property “name” of the object “company” in the controller. The “div” that follows the input acts as a container for the error messages using the “ngMessages” module. The “ng-messages” AngularJS directive holds the directed input field error as “updateDetails.detailsName.$error” where “$error” is a property that will run the registered validators and once the validity of the field change to invalid, the model will be sat to “undefine” presenting the right message defined in the “shared/messages.html”.

```
<p ng-message="required">This field can't be empty!</p>
<p ng-message="minlength">This field is too short</p>
<p ng-message="maxlength">This field is too long</p>
<p ng-message="email">Must be a valid email!</p>
```

The “ng-show” directive will display the “div”, which is hidden, once the expression validates to true. That expression, “updateDetails.detailsName.$touched” is the filed form name, followed by the field name, and the “$touched” special property that vali-

dates to true once the input is focused in and out. The inner “div” is a special include “div” that will include the “shared/messages.html” file using the “ng-messages-include”, and that file contains the previous set of messages in “p” elements.

Like every HTML form, a submit button should be added at the end.

```
<button type="submit" class="btn btn-flat btn-success pull-right"
        ng-disabled="updateDetails.$invalid">Update!
</button>
```

The “ng-disabled” is an AngularJS directive that works like the “disabled” HTML attribute but keeps the binding. This directive will evaluate the expression, and either disable or enable the button.

To test the desired results, the name of the company was left empty, and the entered email was invalid. The error messages appeared instantly, and the “update” button was disabled as shown in the following figure.

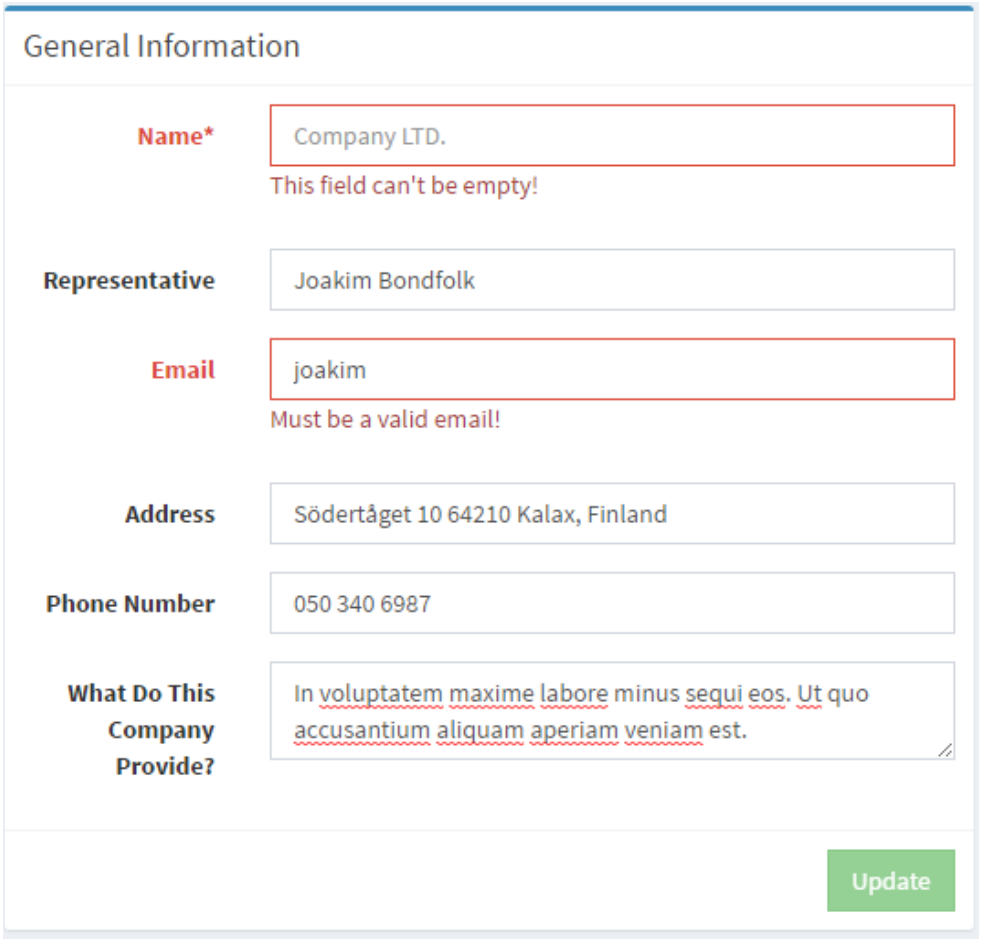


Figure 38. Invalid input error messages

The validation functionality was provided without writing any JavaScript logic, which is another AngularJS advantage.

The data of the company's information is loaded first in the form; then the form is submitted with the new data once the update button is clicked. That previous process is handled in the controller.

```
(function () {
    "use strict";

    var app = angular.module('quote');

    app.controller('DetailsController', ['Details', function (Details) {
       var details = this;
       // Business logic of the details controller
    }]);
}());
```

First, the controller is instantiated, and a dependency on the "Details" service is injected. The "Details" service is defined in "services.js" as a factory, and it depends on the "$http" service and "API" constant to make calls to the API.

```
(function() {
    var app = angular.module('quote');
    app.factory('Details', function ($http, API) {
        return {
            getDetails: function () {
                return $http({
                    method: 'GET',
                    url: API + 'info'
                }).then(function(response) {
                    return response.data;
                }, function (error) {
                    console.log(error);
                });
            }

        };
    });
})();
```

The "getDetails" is responsible for retrieving the company's information by sending a GET HTTP request to the API and handle the promise by returning the response data to the controller. The controller will assign the data to the desired model; the "company".

```
Details.getDetails().then(function (data) {
    details.company = data;
}, function (error) {
    console.log(error);
});
```

Due to the nature of the two-way data binding, the form will be populated with the data.

Updating the information is determined in the same way, where the “updateDetails” is defined in the factory’s returned object.

```
updateDetails: function (details) {
    return $http({
        method: 'PUT',
        url: API + 'info',
        data: details
    }).then(function(response) {
        return response.data.response;
    }, function (error) {
        console.log(error);
    });
},
```

The argument that is passed to this function is a JavaScript object which contains the company’s updated information which has been collected, and assigns that to the “data” in the options object of the “$http” service. That “updateDetails” is used in the “DetailsController” when the form is submitted.

```
details.submitUpdateDetails = function (isValid) {
    if (isValid) {
        Details.updateDetails(details.company).then(function (update) {
            if (update === true) {
                responseMessage('update_success');
            } else {
                responseMessage('update_fail');
            }
        }, function (error) {
            console.log(error);
        });
    }
};
```

The “submitUpdateDetails” is the function that will be executed when the form is submitted, and it is the one that has been bound to the “ng-submit” in the view. That function accepts a Boolean value which is passed from the view and denote the validity of the form. The object of the updated data is passed then to the “updateDetails” method, and a promise is returned and handled. In case the data has been updated successfully, the response from the service is “true”, and the “responseMessage” function will be called with “update_success”. Otherwise, the “responseMessage” will be called with “update_fail”. The “responseMessage” is a defined function that will accept a predefined action and will pass that action to a function on the “Details” service that will determine the response message to that action.

```
function responseMessage(action) {
    details.message = Details.messages(action);
    jQuery('#messageModal').modal('show');
}
```

The “jQuery” is a retrieved reference of the jQuery library which is used to open a Bootstrap modal to display the message. The “messages” is defined in the “Details” service like the following.

```
messages: function (action) {
        switch (action) {
            case 'update_success':
                return {
                    head: 'Update Action',
                    body: 'Company\'s general information have been updated
successfully!'
                };
            case 'update_fail':
                return {
                    head: 'Update Action Failed',
                    body: 'Sorry, we could not update your information, please try
again later!'
                };
            case 'upload_accepted':
                return {
                    head: 'Upload Logo',
                    body: 'Logo has been updated successfully!'
                };
            case 'upload_not':
                return {
                    head: 'Upload Logo Not Accepted',
                    body: 'Logo image not accepted: not supported type or file too
large!'
                };
            case 'upload_error':
                return {
                    head: 'Upload Logo Failed',
                    body: 'Sorry, we could not upload the logo, please try again
later!'
                };
        }
```

The “head” and the “body” are related to the modal, which is identified in the view as the following.

```
<div class="modal fade modal-primary" id="messageModal">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span></button>
                <h4 class="modal-title">{{details.message.head}}</h4>
            </div>
            <div class="modal-body">
                <span>{{details.message.body}}</span>
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-outline btn-flat" data-
dismiss="modal">Close</button>
            </div>
        </div>
    </div>
</div>
```

The model will dynamically display the response message. For instance, when the company's details are updated, the following modal will confirm.

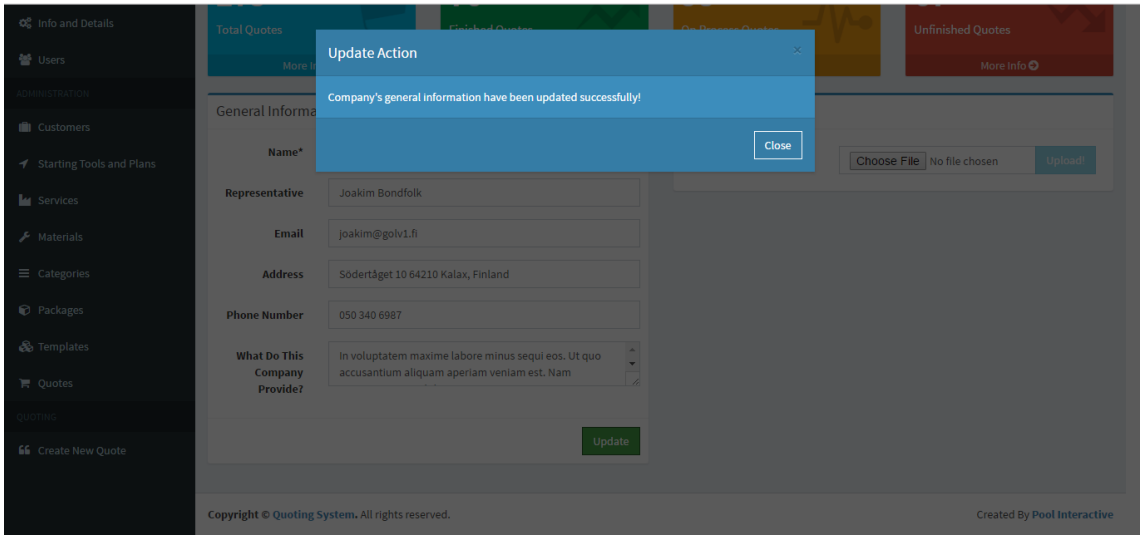


Figure 39. Details have been updated response modal

Pop-up modals are also used for other processes such as deletion confirmation where the modal indicates this by the message and the adaptation to the color scheme which increase the UX. The following is the modal that is used to delete a customer which will show after the "delete" button in the "customers" view table is clicked.

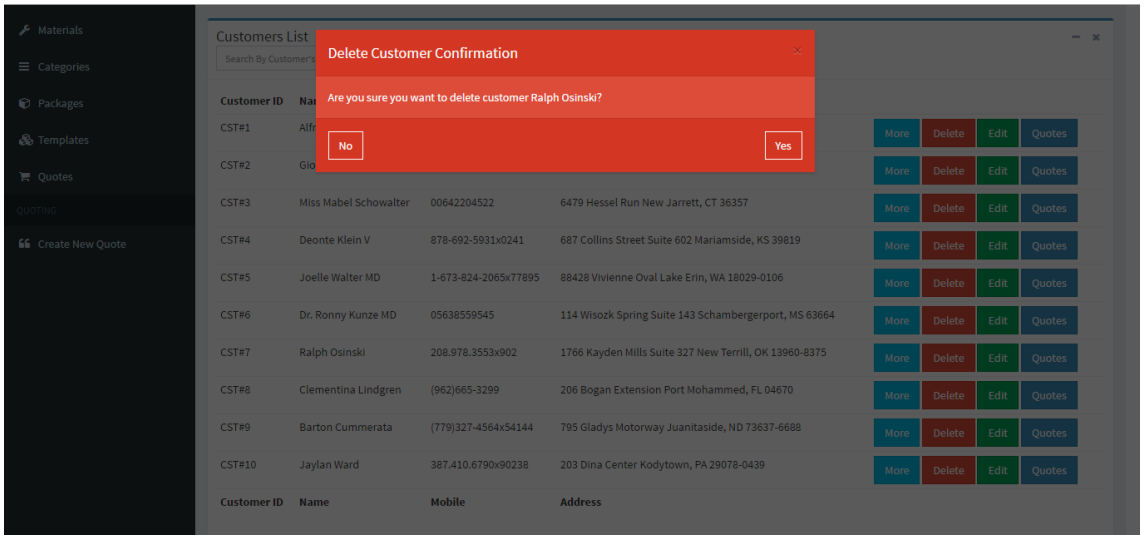


Figure 40. Customer deletion confirmation modal

7.4 Session Storage for Components

Creating a quote is a process that depends on the already-existing components such as services, materials, packages, and templates, besides customers, which means that those components should exist for a quote to be created. However, this might not be ideal for the user to be forced to do it in that particular order for multiple reasons, most importantly forgetting to follow that scenario. Taking that into consideration, the session storage in the browser was used to solve that problem and achieve a better UX.

The following scenario could help to simulate the problem and the solution.

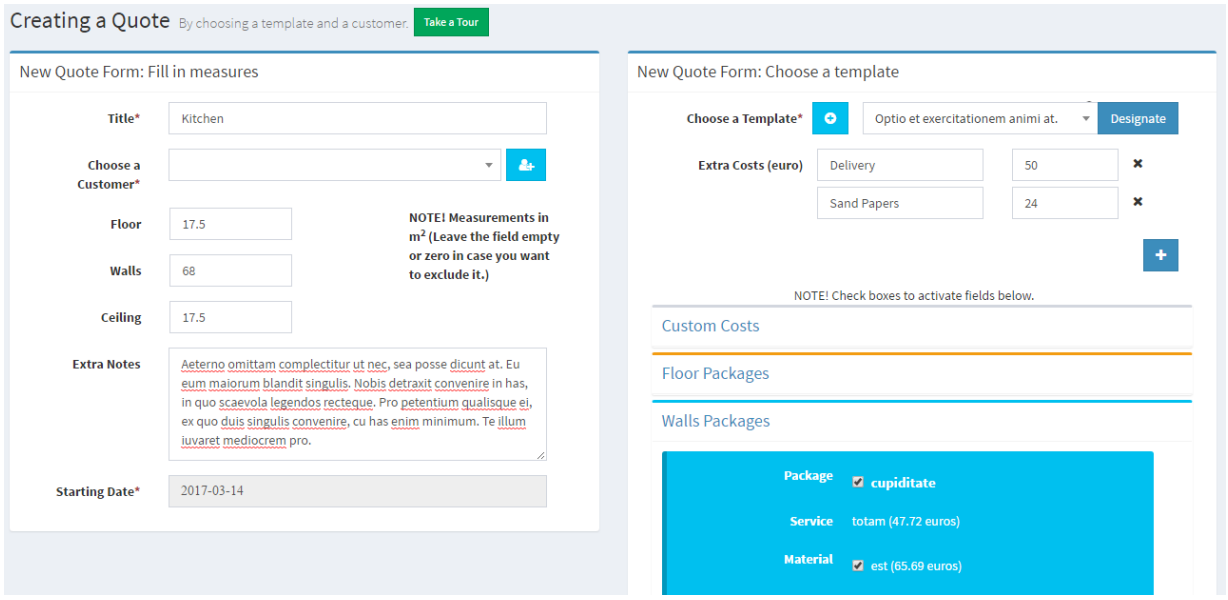


Figure 41. "Create New Quote" view

The “Create New Quote” view filled with data, as shown above, just to notice that the customer was not created yet. Navigating to the “Customers” view will reset the form, and the whole data will be gone. To prevent such scenario, an “Add” button was added to the view next to the “Choose a Customer” drop-down menu.

Clicking that button will keep the “Create a New Quote” form populated and navigate to the “Customers” view where the customer can be created.

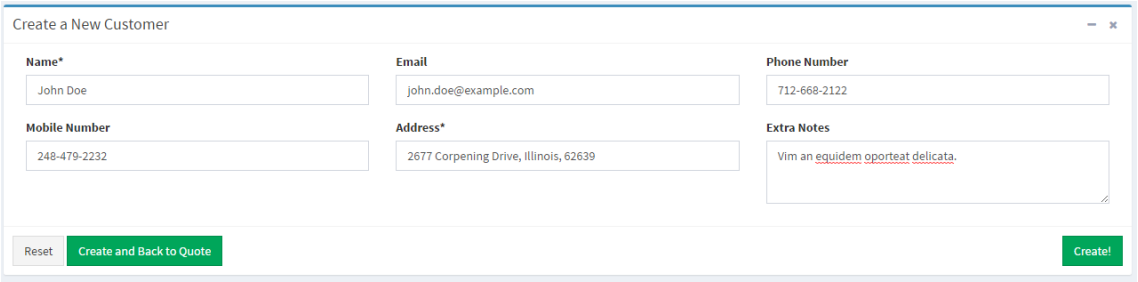


Figure 42. "Create a New Customer" form

The “Create and Back to Quote” button will only be displayed if the navigation was performed like the previous scenario. After clicking the button, the customer will be created and navigate back to the “Create New Quote” view with that customer being chosen.

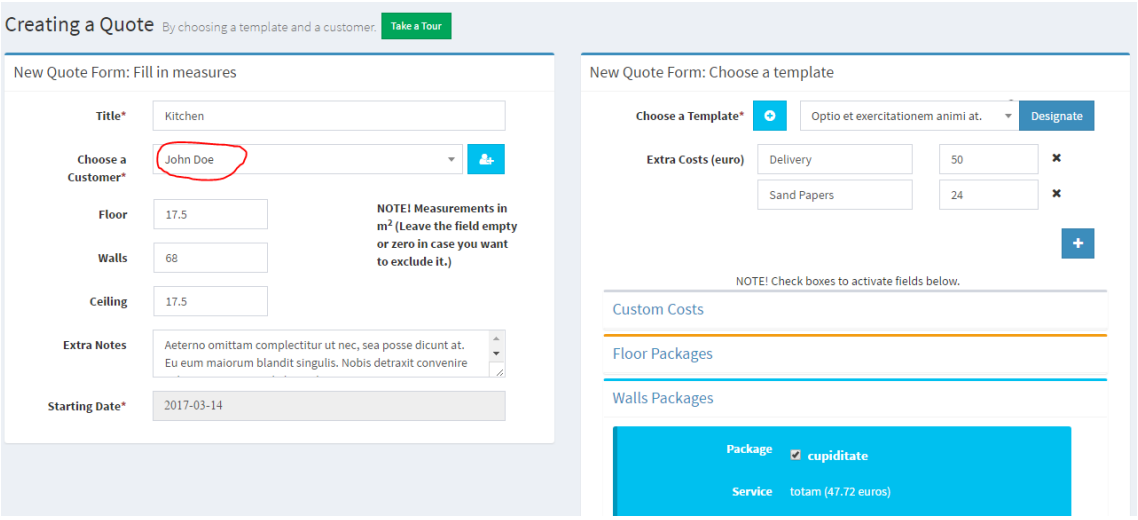


Figure 43. "Choose a Customer" populated after creation

This is not limited to the customers only but extended to the templates in this view as shown above. Furthermore, the packages, and starting tools and plans do have this feature added to the “Templates” view. The “Packages” view also include this feature to the services and materials.

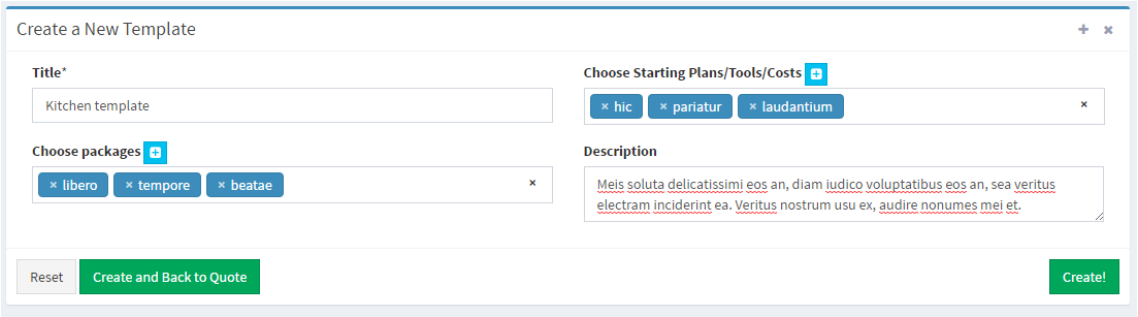


Figure 44. "Create a New Template" form

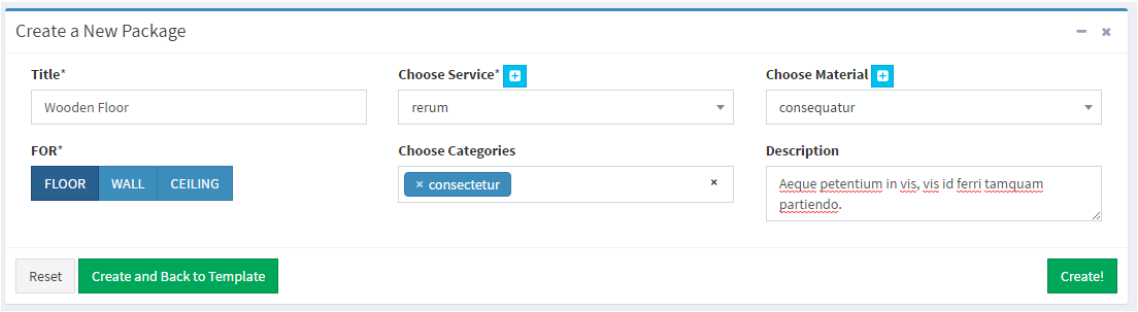


Figure 45. "Create a New Package" form

Creating these components back-to-back, the system will cover all the missing components without losing the state data.

To illustrate how this feature was developed, the customer creation was chosen for instance. Using the “ngStorage” module that has been installed previously, a session storage object instance was created at the top of the “quotes/new.controller.js” controller.

```
quote.$storage = $sessionStorage;
```

Next, a function is created that will set a property in the session storage to keep track of the status and navigate to the “Customers” view. The function will be bound to the “Add Customer” button on the view.

```
quote.addCustomer = function () {
    // Set to true to come back after creating the customer
    quote.$storage.quote_customer_loc = true;
    // Navigate to customers page to create customer
    $location.path("/customers");
};
```

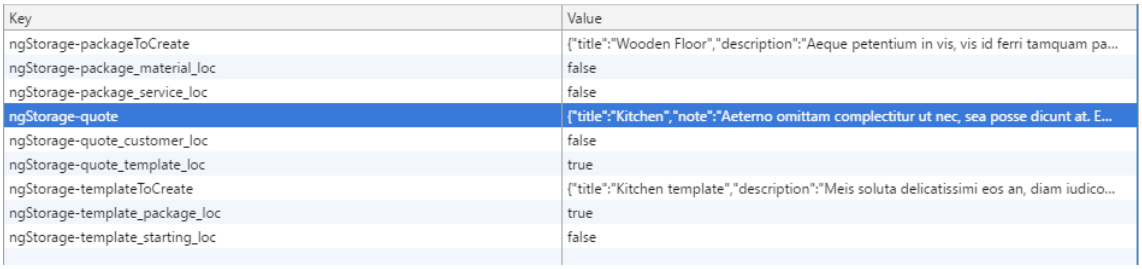
The quote object is created and saved as an instance in the session storage. Placing the initiation of the quote in a function is useful for resetting object functionality.

```
quote.$storage.quote = {
    title: '',
    note: '',
    customer: 0,
    template: 0,
    floor: 0,
    wall: 0,
    ceiling: 0,
    startings: [],
    status: 'created',
    start_date: '',
    extras: [],
    packages: {
        floor: [],
        wall: [],
        ceiling: []
    },
    total: {
        price: 0,
        tax: 24,
        final: 0
    }
};
```

Similarly, the customer object should also be saved in the session storage, which will be handled in the “customers/main.controller.js” controller. After that, when the “Create and Back to Quote” button is clicked, the following is executed.

```
// Set to false after finishing from creating the customer
customers.$storage.quote_customer_loc = false;
// Navigate to back to "creating a quote" page
$location.path("/create-quote");
```

The same process is used for all the components that require this temporary save, and because it is a session storage, the data will not be lost until the browser is closed.



Key	Value
ngStorage-packageToCreate	{"title":"Wooden Floor","description":"Aeque petentium in vis, vis id ferri tamquam pa...
ngStorage-package_material_loc	false
ngStorage-package_service_loc	false
ngStorage-quote	{"title":"Kitchen","note":"Aeterno omittam complectitur ut nec, sea posse dicunt at. E...
ngStorage-quote_customer_loc	false
ngStorage-quote_template_loc	true
ngStorage-templateToCreate	{"title":"Kitchen template","description":"Meis soluta delicatissimi eos an, diam iudico...
ngStorage-template_package_loc	true
ngStorage-template_starting_loc	false

Figure 46. Session storage of the components

7.5 Uploading Images

As discussed in Chapter 6 Section 6.11, the images should be sent as Base64 encoded files to the API where they will be decoded and saved on the disk with a reference in the database. For that, the front-end web application will take care of encoding the images. To illustrate this process, the “Upload Logo” feature is chosen as an example.

Using the module “angular-base64-upload” that has been installed previously, the process of converting files from file input to Base64 encoded becomes easy.

First, the file input should have an attribute “base-sixty-four-input”, which will tell the previous module to take over this file input and Base64 encode the file.

```
<input type="file" class="form-control" ng-model="details.logoImg"
       name="logoImg" accept="image/jpeg,image/png"
       maxsize="2000" required
       base-sixty-four-input>
```

The “ng-model” will take care of holding the output of the encoded file, besides other useful information such as the size, the file type, and the file original name. The “maxsize” is a validation attribute that the module provides, and it defines the maximum size of the file in Kilobytes. Besides those AngularJS attribute directives, the “accept” which will specify the accepted file types is a native HTML attribute.

To enhance the UX, the image is being displayed next to the file input. The image is using the encode file which will change every time a new image file was chosen, giving an instant view.

```
<img ng-if="details.logoImg" class="center-block img-responsive img-thumbnail"
     ng-src="data:{{details.logoImg.filetype}};base64, {{details.logoImg.base64}}"
     alt="Logo" />
```

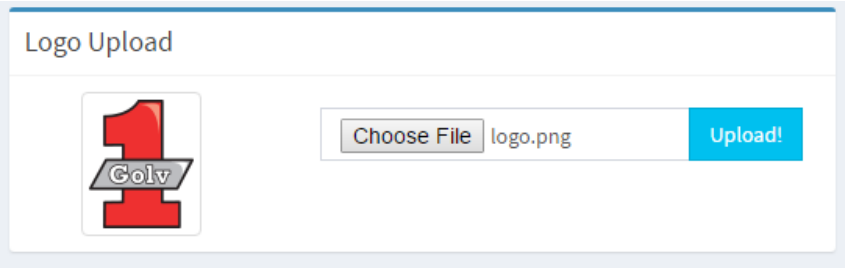


Figure 47. Logo Upload form

7.6 **Internationalization**

Internationalization, which is also referred to as ‘i18n’ is the process of making a product, or an application easily enables localization for viewers that vary in culture or language. For instance, in the Finnish language, the date format is “day.month.year”, while in American English it is “month/day/year”. The same for numbers, and currencies.

AngularJS helps with this matter using the internationalization module that was developed by the AngularJS team. By installing and including the module as a dependency for the app, it requires nothing else for it to wok. The localization that needs to be supported should be included.

AngularJS uses its filter mechanism to apply the loaded localization. For instance, the following shows how to apply the “date” filter on a Unix timestamp in seconds.

```
{{“1489320732” | date}}
```

The language is also a part of the internationalization. As seen before, the header menu has a language menu where a language can be chosen. Using the translation module that has been installed before, the language could be switched instantly without a need to request the page again. The module uses the two-way data binding feature that is provided by AngularJS to load the translations of a language.

The configurations of the translation module were handled before when the application was configured in the “app.js”. However, to make use of that, the translations are saved as JSON files in the “languages” as the standard abbreviation for the language such as ‘en’ for English, and ‘sv’ for Swedish.

The JSON object structured as each component has its own block, and the keys were all capitals. For instance, the “Services” component has the header, create and update form, the table which lists the services, the service details modal, and the delete modal, and each of those has a block in the “SERVICES” translation block. This consistency in the structure made it easier to search and detect the translation. The same structure should be used for the other languages in order to maintain the translation.

```
"SERVICES": {
  "HEADER": {
    "TITLE": "Services",
    "SUBTITLE": "View, create, and delete services. Services are what the company provide.",
    "TOUR" :"Take a Tour",
    "UPDATE_HEADER": "Update Service"
  },
  "CREATE_UPDATE": {
    "CREATE_HEADER": "Create a New Service",
    "UPDATE_HEADER": "Update Service Form",
    "TITLE": {
      "TITLE": "Title",
      "PLACEHOLDER": "Service Title"
    },
    "AMOUNT": {
      "TITLE": "Charging Amount",
      "PLACEHOLDER": "i.e. 5.55"
    },
    "DESCRIPTION": {
      "TITLE": "Description",
      "PLACEHOLDER": "What is this service?"
    },
    "BACK": "Back",
    "UPDATE": "Update!",
    "RESET": "Reset",
    "CREATE": "Create!"
  },
  "SERVICES_LIST": {
    "HEADER": "Services List",
    "TABLE":{
      "ID": "Service ID",
      "TITLE": "Title",
      "AMOUNT": "Charging (euro per square meter)",
      "ACTIONS": {
        "MORE": "More",
        "DELETE": "Delete",
        "EDIT": "Edit"
      }
    }
  },
  "DETAILS": {
    "HEADER": "Service Details",
    "TITLE": "Title",
    "AMOUNT": "Charging Amount",
    "DESC": "Description",
    "CLOSE": "Close"
  },
  "DELETE": {
    "CONFIRM": "Delete Service Confirmation",
    "MESSAGE": "Are you sure you want to delete service",
    "NO": "No",
    "YES": "Yes"
  }
}
```

To reflect these translations on the view, the “translate” directive is used. To demonstrate this, the service title “label” HTML element looks like the following.

```
<label class="control-label"
       translate="SERVICES.CREATE_UPDATE.TITLE.TITLE">Title</label>
```

Even though the translate covers most of the cases, some cannot be achieved using it. One of these cases is the HTML attributes content translation such as the “placeholder” attribute. In this case, the “translate” filter is used. For instance, the former label is followed by an input that has a “placeholder” attribute.

```
<input class="form-control" type="text"
       placeholder="{{'SERVICES.CREATE_UPDATE.TITLE.PLACEHOLDER' | translate}}"
       autocomplete="off" name="serviceTitle" required minlength="4"
       ng-model="services.serviceToCreate.title">
```

The module offers a service “$translate” that makes loading the language easy by using the “use” method. This method takes the language that needed to be loaded which matches the file name without the “.json” extension. For example, “$translate.use(‘en’)” will load the English JSON object. However, to make this change dynamic, a function which will return the language depending on the user choice was placed in the “components/TranslateController.js”.

```
var vm = this;

vm.$storage = $localStorage;

vm.changeLang = function (Lang) {
       // Update the language in the localStorage
       vm.$storage.lang = Lang;

       // Change the language in the view
       $translate.use(Lang);
};
```

The language abbreviation is stored in the browser local storage, and the function will update the user preference in the local storage for the future, and use that as a choice.

However, to enhance the UX, the system will detect the browser language, and load the language if it has any translation file, if it does not support it, then it will default back to English.

```
var checkLang = function () {
    // Get the language of the browser
    var browserLang = window.navigator.userLanguage || window.navigator.language,
        availableLangs = ['en', 'sv'];

    if (!vm.$storage.lang) {
        switch (browserLang) {
            case 'en-US':
                vm.$storage.lang = 'en';
                break;
            case 'en':
                vm.$storage.lang = 'en';
                break;
            case 'sv-SV':
                vm.$storage.lang = 'sv';
                break;
            case 'sv':
                vm.$storage.lang = 'sv';
                break;
            default :
                vm.$storage.lang = 'en';
        }
    }

    // Check the language
    if (availableLangs.indexOf(vm.$storage.lang) == -1) {
        // Default back to English
        vm.$storage.lang = 'en';
    }

    // Return that language
    return vm.$storage.lang;
};
```

By changing the language to Swedish, the services components page will look like in the following figure.

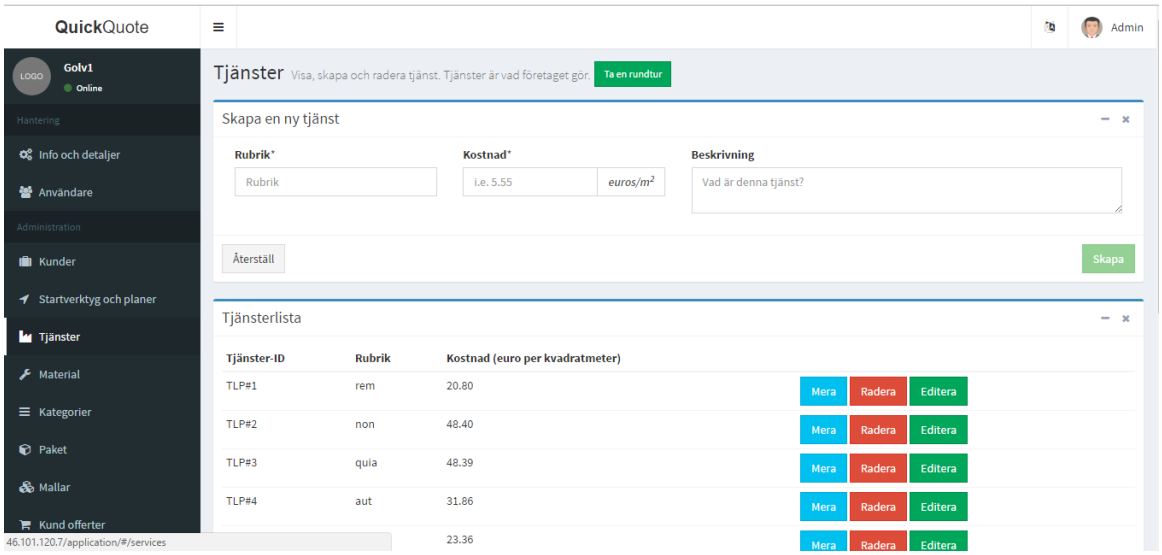


Figure 48. "Services" component view in Swedish

7.7 Interactive Tutorial for System Usage

A system that is not easy and clear to use is more likely to fail, regardless of the features that it might provide. Thus, most of the applications have documentation tutorials, video tutorials or hands-on training.

The hands-on training is the most efficient solution and is chosen by many enterprises. However, it is an expensive process, and hard to maintain when the system is rapidly developing and changing.

A similar result could be achieved by creating an interactive tutorial that will show the user the steps of using with the application. For each component view, and "Take a Tour" button appears at the top next to the header.

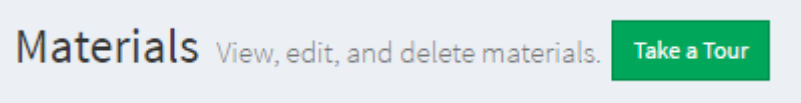


Figure 49. "Take a Tour" for the "Materials" component view

When the button is clicked, an interactive tour will start, highlighting the steps of creating that component while explaining in details each step.

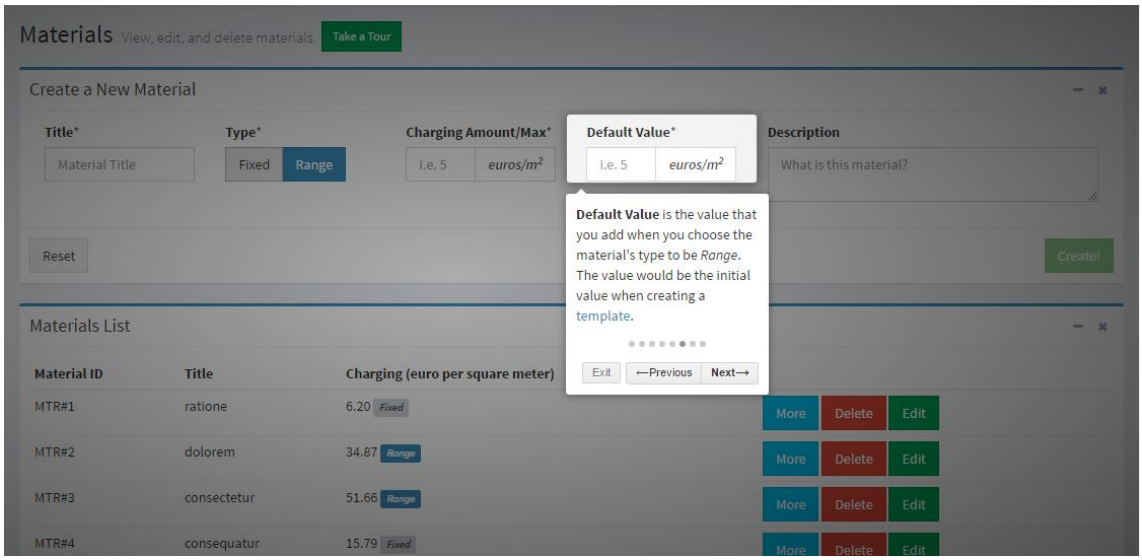


Figure 50. Interactive tour of setting up a material

The library that was used to achieve this result is "Intro.js", which is standalone and has no dependencies. To make it work with AngularJS, a wrapper module with directives was created. /52/

In each view, a set of id attributes for each HTML element that requires an explanation should be added. As a convention, the id holds a value of “step” followed the number of the step. For instance, the “div” element that contains the “Amount/Max” looks like the following.

```
<div class="col-lg-2 col-md-3 col-sm-12 col-xs-12 form-group" id="step3">
...
</div>
```

Then, in the controller that is responsible for that view, the tour steps should be defined as an array of object. Each object represents a step in the tour and contains two mandatory properties: an element which is the HTML element id and intro which is the description that associated with that element.

```
[
    {
     element: '#step1',
     intro: 'A material is the perceptible part of a <a href="#/packages">package</a>. The
material combined with a <a href="#/services">service</a> creates a package. Think of a
material as a substance or tool that helps to perform a service/work. A good example of a
material could be "Paint/Dye for Walls" and combined with a "Painting a Wall" service, creates
a package.'
    },
    {
     element: '#step2',
     intro: '<b>Type</b> determines the material charged amount while creating a quote.
<i>Fixed</i> is for when the material have a fixed price and can\'t be changed (for example
when you have only one class of a material in the market). <i>Range</i> is when you have a
variaty of material\'s prices.'
     },
     {
      element: '#step3',
      intro: 'Make sure to add the amount that you are going to charge for the material.'
     },
     {
      element: '#step4',
      intro: 'Materials are being charged as <b>euro per square meter</b>, and VAT is applied
upon them.'
     },
     {
      element: '#step5',
      intro: '<b>Default Value</b> is the value that you add when you choose the material\'s
type to be <i>Range</i>. The value would be the initial value when creating a <a
href="#/templates">template</a>.'
     },
     {
      element: '#step6',
      intro: 'A list of all the materials.'
     },
     {
      element: '#step7',
      intro: 'Come back again every time you want a tour.'
     }
];
```

To maintain the code quality, the array is returned from a service where the translation could be applied. Then, those are loaded into the controller where the write function is called depending on the chosen language.

```
var tourSteps = Materials.enTour(),
    nextLabel = '<strong>Next&xrarr;</strong>',
    prevLabel = '<span>&xlarr;Previous</span>',
    skipLabel = 'Exit',
    doneLabel = 'Done';

if (materials.$lStorage.lang == 'sv') {
    tourSteps = Materials.svTour();
    nextLabel = '<strong>Nästa&xrarr;</strong>';
    prevLabel = '<span>&xlarr;Föregående</span>';
    skipLabel = 'Avsluta';
    doneLabel = 'Färdig';
}
```

Finally, the options are loaded dynamically using the data-binding.

```
materials.IntroOptions = {
    steps: tourSteps,
    showStepNumbers: false,
    exitOnOverlayClick: true,
    exitOnEsc: true,
    nextLabel: nextLabel,
    prevLabel: prevLabel,
    skipLabel: skipLabel,
    doneLabel: doneLabel,
    tooltipPosition: 'auto'
};
```

To trigger the translation change, the translation module emits a call that other modules can subscribe to when the language is changed successfully.

```
$rootScope.$on('$translateChangeSuccess', function() {
    // updating steps config
    updateLangConfig();
});
```

7.8 **Performance**

Optimizing a web application performance should happen when needed, and as early as possible. It contains three steps: evaluating the performance, analyzing and detecting the issues/bottlenecks, and fixing those issues.

Evaluating and measuring the performance happens by mentoring it (on production). Some of the tools that are used to diagnose the performance are Google Chrome Dev-Tools and YSlow. /53/ /54/

Google Chrome DevTools is a very useful set of tools that comes with Google Chrome browser. It consists of multiple tabs where each one group functionality. For instance, the “Network” panel shows the requested resources and how they are getting download-ed in real time which makes it easy to detect and identify the ones that are taking time more than expected. This is considered a fundamental process to optimize the page per-formance. The following figure shows the “Network” panel recording for the Chrome DevTools website over a slow network.

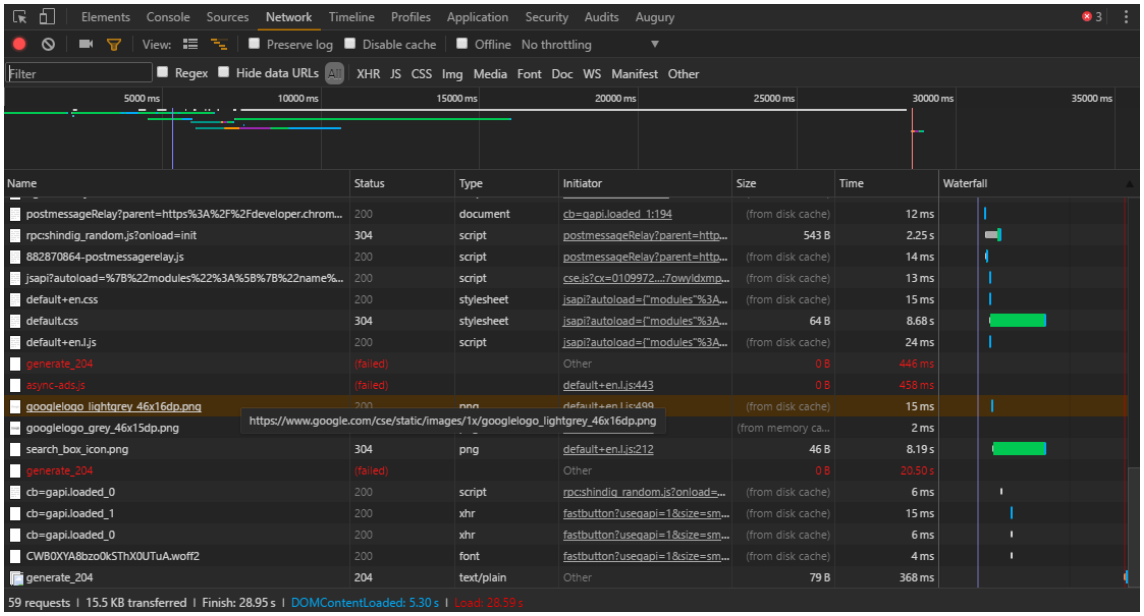


Figure 51. Network panel recordings of Chrome DevTools webpage

The “Profiles” panel helps to record and to profile the CPU and memory usage of Ja-vaScript and CSS. Some profiling types are “JavaScript CPU Profile” which shows the consumed time of the webpage’s JavaScript functions, “Heap Snapshot” which profiles the consumed memory of the webpage’s JavaScript objects and DOMs, and “Allocation Profile” which displays JavaScript functions’ memory allocation.

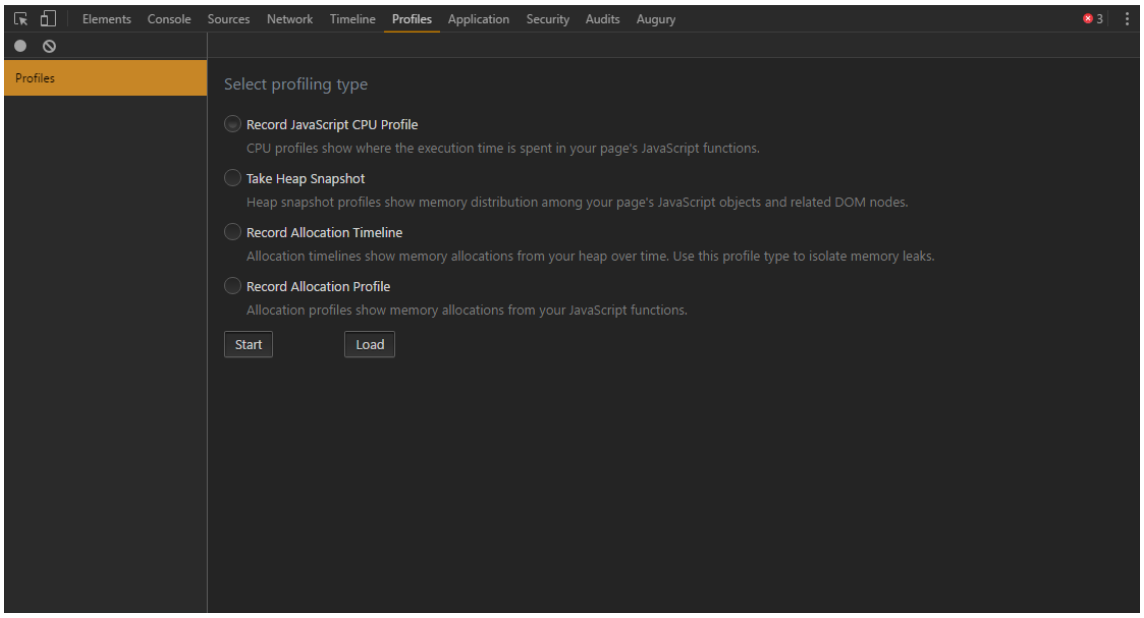


Figure 52. Chrome DevTools Profiles Panel

The "Audits" panel analyze the web page load and suggest methods to optimize it and reduce the load time.

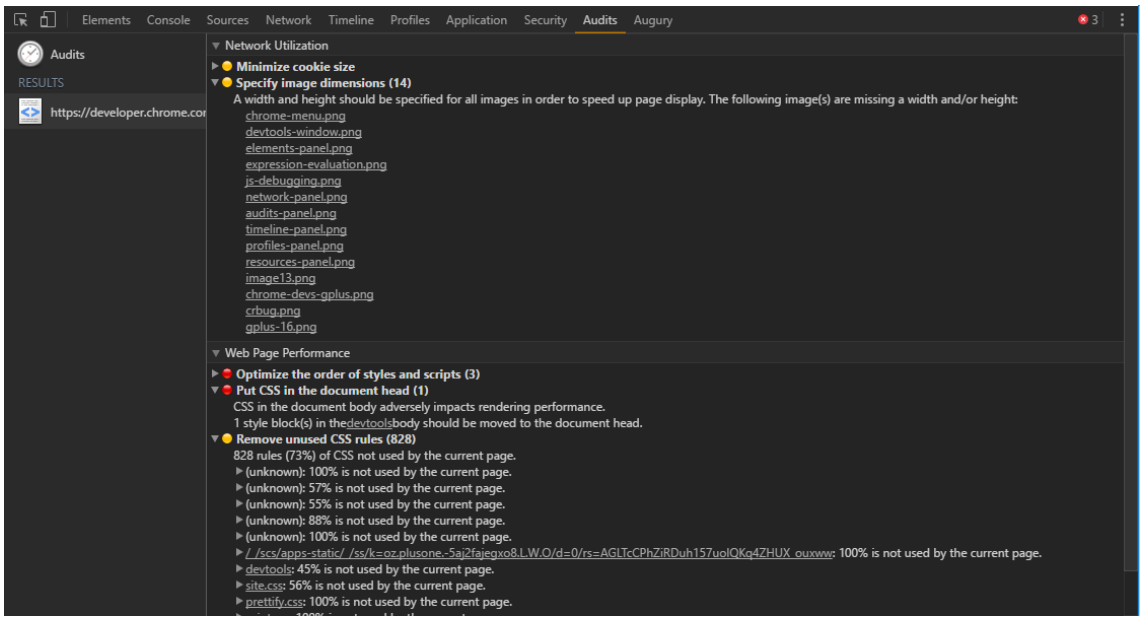


Figure 53. Audits panel suggestions for Chrome DevTools webpage

Some of the optimization processes that have been done on the network level were reducing the JavaScript, CSS, and image files size by using Gzip compression and "minifiers". Gulp was used to automate this process.

Also, content delivery network providers were used to fetch some external libraries and frameworks for production instead of fetching them from the server which saved the bandwidth and increased the performance by distributing the load.

8 SYSTEM IMPLEMENTATION: SECURITY

In this chapter, the authentication and authorization of the user and the service level are discussed.

8.1 User Communication Level

This level is about the communication between the client and the system. In more details, the web application and the API. However, the API is built as a REST API, and REST is stateless, as the name indicates. Thus, the session-cookie mechanism would not work. Token-based authentication is the solution, and JWT is chosen for multiple reasons mentioned in Chapter 5, Section 5.3. For both the API and the web application, the helper libraries were installed and configured previously.

From the API perspective, a route that will handle the authentication and returning the JWT is created like the following.

```
// Authentication
Route::post('authenticate', 'AuthenticateController@authenticate');
```

After that, creating the "AuthenticateController" with a method called "authenticate" which will receive the user's credentials and return the JWT corresponding to the user if the credentials are right.

```
public function authenticate()
{
	$credentials = Input::only('email', 'password');

	$user = User::where('email', '=', Input::get('email'))->first();

	$customClaims = ['id' => $user->id, 'email' => $user->email];

	try {
		if (! $token = JWTAuth::attempt($credentials, $customClaims)) {
			return Response::json(['error' => 'invalid_credentials'], 401);
		}
	} catch (JWTException $e) {
		return Response::json(['error' => 'could_not_create_token'], 500);
	}

	return Response::json(compact('token'));
}
```

The response is the JWT in JSON with a property called "token" in the case of proper credentials. Otherwise, 401 Unauthorized will be returned if the credentials are not valid, or 500 Internal Server Error in the event of an exception.

Now the routes that need to be protected should be wrapped in a group that applies the "jwt-auth" filter.

```
Route::group(array('before' => 'jwt-auth'), function() {
    // protected routes
});
```

Another method "getAuthenticatedUser" in the "AuthenticateController" is defined to retrieve the authenticated user from the token.

```
public function getAuthenticatedUser()
{
        try {
                if (! $user = JWTAuth::parseToken()->authenticate()) {
                        return response()->json(['user_not_found'], 404);
                }
        } catch (Tymon\JWTAuth\Exceptions\TokenExpiredException $e) {
                return response()->json(['token_expired'], $e->getStatusCode());
        } catch (Tymon\JWTAuth\Exceptions\TokenInvalidException $e) {
                return response()->json(['token_invalid'], $e->getStatusCode());
        } catch (Tymon\JWTAuth\Exceptions\JWTException $e) {
                return response()->json(['token_absent'], $e->getStatusCode());
        }

        return response()->json(compact('user'));
}
```

Now the web application which is consuming the API should use the "Authorization" HTTP header with "Bearer" followed by the token to authenticate. However, the token could be sent via a query string as "token" in the URL. Fortunately, Satellizer is configured to do so.

In the "auth" component on the front-end side, the "$auth" service, which is a special service comes with Satellizer, is used to send the credentials.

```
var credentials = {
    email: vm.email,
    password: vm.password
};

$auth.login(credentials).then(function(data) {
    if ($rootScope.myPrevLocation) {
        $location.path($rootScope.myPrevLocation);
    } else {
        $location.path("/details");
    }
}, function () {
    vm.showWrong = true;
});
```

A promise is returned, and if the response includes the token, a redirection takes place. Otherwise, an error will be shown in the view.

```
<div class="row" ng-show="auth.showSent">
  <div class="col-xs-12 text-success" translate="AUTH.RESET_SENT">
     An email has been sent to you, please check your mailbox!
  </div>
</div>
```

The token is stored automatically in the local storage of the browser. However, this behavior could be changed to use other methods such as a cookie, or session storage. To protect the web application AngularJS routes, the “resolve” property on the route “when” options is used.

```
resolve: {
    auth: ["$q", '$window', function($q, $window) {
        var userInfo = $window.localStorage.getItem('satellizer_token');
        if (userInfo) {
            return $q.when(userInfo);
        } else {
            return $q.reject({ authenticated: false });
        }
    }]
}
```

It will check for the token existence in the local storage, and send a request to the API authentication route to check if the user is authorized. If it is not, then an object with “authenticated” property sat to false is returned forcing the app to go to the authentication route again.

The JWT time is set to be short for security reasons. However, if the user was inactive for longer than the JWT lifetime, then the application should reject any request and navigate to the login page again.

Achieving this requires the “$httpProvider” and “$provider” injected in the “app.config” function.

```
app.config(function ($routeProvider, $authProvider, $translateProvider, $httpProvider,
$provide) { ... }
```

Then defining the method that will take care of the redirect when the token is expired, absent, invalid, or missing.

```
        function redirectWhenLoggedOut($q, $injector) {
            return {
                responseError: function(rejection) {
                    var $location = $injector.get('$location');
                    var rejectionReasons = ['token_not_provided', 'token_expired',
'token_absent', 'token_invalid'];

                    angular.forEach(rejectionReasons, function(value, key) {

                        if(rejection.data.error === value) {
                            $location.path('auth');
                        }
                    });

                    return $q.reject(rejection);
                }
            }
        }

        $provide.factory('redirectWhenLoggedOut', redirectWhenLoggedOut);

        $httpProvider.interceptors.push('redirectWhenLoggedOut');
```

8.2 **Microservices Communication Level**

The microservices communicate with each other's using REST and to secure the communication between the services, JWT was used. It is enough to demonstrate how the communication between two microservices, such as the quoting service and the invoicing service, is secured as the others follow the same strategy.

When a quote is delivered, a task to create an invoice for that quote will be created and pushed to a task queue system, which will be used later by a message broker to pull that task from the queue to the worker. The worker will delegate that task to the invoicing service, which will require access the quoting service to get the quote data. The access cannot be granted to any of the services unless it is authenticated.

In such a scenario, the JWT is used to secure the calls, and it is placed in an HTTP header. Some of the properties of the JWT's payload are used by the authentication process such as "iss", "aud", and "sub".

```
{
  "iss": "http://invoice.localhost/",
  "aud": "http://quote.localhost/",
  "sub": "http://quote.localhost/quote/5784/"
}
```

Each service has its private and public key, and the JWT that is generated is signed by the invoicing service using its private key. The keys will be used to validate the JWT. When the quoting service receives the request, it will extract the JWT from the header and decode it. After decoding it, the quote service will check the issuer, the “iss”, and know that it is the invoicing service. Then, the quote service will locate the public key of the invoicing service and validate the signature. Finally, it will respond depending on the validation, and the invoicing service will receive the quote data if the validation was passed.

For the other requests, such as “POST” and “PUT”, the data is passed as an MD5 hash to a custom claim “inf” in the JWT payload to ensure that the data is not tampered with along the way. However, the JWT is signed and not encrypted, so the SSL is used to encrypt the request.

```
{
  "iss": "http://invoice.localhost/",
  "aud": "http://quote.localhost/",
  "sub": "http://quote.localhost/quote/5784/",
  "inf": "<md5 hash of the payload>"
}
```

8.3 **Custom Properties and Authorization**

Each user has a role associated with his/her credentials. To pass the role between the web application and the API, an extra property in the JWT payload’s custom claims was added.

That claimed was used by the API through a filter applied to the targeted routes, and by the web application in the “resolve” of each AngularJS route. A simple check was performed to obtain the desired results.

```
{
  "role": "admin"
}
```

9 DOCKER DEPLOYMENT AND ORCHESTRATION

This chapter will give an insight into the Docker architecture and its ecosystem, how to install it with other components, and how to deploy Docker containers using Kubernetes clustering and orchestration engine.

9.1 Docker Architecture and Ecosystem

Until recently, most of the cloud platforms were based on virtual machines which as discussed in Chapter 3, Section 3.2, encapsulate guest operating systems alongside the libraries and the software applications. However, Linux containers are considered operating system level virtualization which provides isolated user-space instances, and processes on a single host machine (sharing the same Linux kernel). Containerization has made it clear that it is the future regarding software containerization in the cloud.

Docker as a tool was built as an abstraction on top of the low-level Linux containers allowing easy programmatic creation and distribution of container images, in addition to launching and deploying containers. It provides a command line tool and HTTP API which makes managing containers easy and automatable. Docker containers ship with the containerized software applications' dependencies which keeps the deployment process the same regardless of the technology or the Linux distribution running the container. This standardized unit of the application and its dependencies is the Docker image, which is a layered immutable approach to creating images. Docker encompasses an ecosystem of other tools and services such as Docker Hub which is a central repository that hosts Docker images. /55/

Docker is a client-server application with a number of components.

- Docker Engine: The core of the Docker platform.
- Docker Client: Is the client which will accept the calls from the command line and send them to the Docker Daemon via HTTP or TCP socket regardless of the hosting machine (same or different). However, it will not interact with the images and containers directly.
- Docker Host: The server-side of the Docker Engine which contains the Docker Daemon, the images, and the containers. It exposes an HTTP API for managing containers and images.

- Docker Daemon: The process that interacts with the images and the containers that are hosted on the same Docker Host.
- Docker Registry: A remote service that host and publish Docker images. Docker Hub is the official Docker Registry, and most of the common applications, programming language runtimes, and databases have official Docker Hub images.
- Docker Machine: Command line tool that is used to create and manage Docker Hosts which can be controlled by the local Docker Client.
- Docker Swarm: Clustering tool to administer a group of Docker Hosts acting as a single Docker Host. The communication is through an API that is quite similar to the Docker API. The Docker Swarm API is used by the Docker Client.
- Docker Compose: Specifies and groups containers to run and network together.

These are the main Docker components in its ecosystem, but there are much more.

9.2 Installing Docker Components

First, Docker Engine component must be installed in order to be able to install the rest. However, because the server is running Ubuntu Trusty 14.04, some packages are recommended to be installed such as "linux-image-extra-*" packages that allow Docker to use "aufs" storage drivers.

```
sudo apt-get install \
linux-image-extra-$(uname -r) \
linux-image-extra-virtual
```

As most of the packages on Debian distributions, Docker could be installed using repositories or by downloading the DEB package and install it manually. However, the former approach is much better as it eases up the installation and upgrading.

Docker has both Enterprise and Community Editions. However, the Community Edition will be installed here starting by allowing "apt" to use a repository over HTTPS.

```
sudo apt-get install \
apt-transport-https \
ca-certificates \
curl \
software-properties-common
```

Then adding the Docker's official GPG key.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

After that, the following command will setup the stable repository.

```
sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

Finally, updating “apt” package index and installing Docker Engine. The “VERSION” should be the Docker version. Specifying the version of Docker recommended on production systems.

```
sudo apt-get update
sudo apt-get install docker-ce=VERSION
```

Docker Daemon binds to a Unix socket that is used by “root” user, so other users need to use “sudo” in order to use the Docker commands. To avoid using “sudo” each time, a “docker” group could be created, and the user could be added to it.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

It is also recommended to run Docker on system boot, and this could be achieved using “systemd” or “upstart”.

```
sudo systemctl enable docker
```

Secondly, the Docker Machine will be installed. It is worth mentioning that Docker Machine usually installed automatically if Docker is installed on Mac or Windows, or by installing Docker Toolbox (old solution for Windows and Mac).

To install Docker machine, the binary should be downloaded and extracted to the “PATH”.

```
curl -L https://github.com/docker/machine/releases/download/v0.10.0/docker-machine-
`uname -s`-`uname -m` >/tmp/docker-machine &&
  chmod +x /tmp/docker-machine &&
  sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

By running the following command, the installation could be tested if it was successful.

```
docker-machine version
```

It will basically return the version of the Docker machine.

Finally, Docker Compose also installed by default for Windows and Mac users, but not for Linux. The following command shows how to download the binary.

```
curl -L "https://github.com/docker/compose/releases/download/1.11.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Then, the binary should be granted executable permissions.

```
chmod +x /usr/local/bin/docker-compose
```

To check if the installation was successful, the version of Docker Compose could be verified.

```
docker-compose --version
```

9.3 Clustering and Orchestration with Kubernetes

Docker Swarm is the native orchestration and clustering engine for Docker. However, it is not the only one. Kubernetes is another most common used engine to deploy containers inside clusters.

Google created Kubernetes which was the outcome of years of experience working with Linux containers. Kubernetes solved many problems that Docker had in its early stage around version 1.0, which made it a very useful tool alongside Docker. /56/

Kubernetes uses “etcd” which is a distributed key-value store for critical data of a distributed system. It has a load balancer integrated, and it provides an easy way to move containers without losing data by mounting persistence volumes. It also uses “flannel” for networking between containers. /57/ /58/

Kubernetes does not use Docker CLI or Docker Compose to define containers. It uses a different API and different definitions as it was not made for Docker exclusively. For that, Kubernetes has a learning curve handling its CLI and configurations.

The advantages of using Docker Swarm are the following:

- Easy and fast to install and use.
- Share almost the same API with Docker.
- Compliment Docker tools such as Docker Compose.
- Lightweight.

However, its disadvantages are:

- Limited in functionality to the Docker API.
- Limited fault tolerance.

Kubernetes, on the other hand, has the following advantages:

- Modular.
- Runs consistently on any OS.
- Easy service organization with pods.

However, its disadvantages are:

- Difficult to install and configure.
- Incompatible with Docker CLI and tools such as Docker Compose.

9.4 **Deployment Docker Containers**

The needs are to deploy applications quickly and automate most of the parts, easily scale Docker hosts, continuous deployment when new features come, and deploy on multiple providers.

By deploying Docker containers, the build happens during the development phase leading to much efficient consistency between development and production stages, not mentioning the apps are loosely coupled and modular by nature, and each container has its own IP.

To deploy a new application, the "Dockerfile" should be created, Docker machine started, and the image is built.

```
docker-machine start
docker build -t quotation-app:v1 .
```

To install Kubernetes and launch a cluster, the following could be run.

```
export KUBERNETES_PROVIDER=vagrant; wget -q -O - https://get.k8s.io | bash
```

It will run Kubernetes in Vagrant on a local virtual machine. However, it could be running on any desired provider that is supported by Kubernetes such as Google App Engine, AWS, and Microsoft Azure.

To run the quotation application container on the previously created cluster, the following could be run.

```
kubectl run quotation-app --image=PROJECT_ID/quotation-app:v1 --port=8080
kubectl get deployment quotation-app
```

The number of replicas could be scaled easily like the following, and they will be added automatically.

```
kubectl scale deployment quotation-app --replicas=4
kubectl get deployment quotation-app
```

To create a load balancer and expose the deployed container to be accessed through the web, the following commands should be executed.

```
kubectl expose deployment quotation-app --type="LoadBalancer"
kubectl get services quotation-app
```

An external IP will be returned, which could be used by the web server to direct the requests to the container.

To separate the tenants from each other, each tenant was given a namespace in Kubernetes which isolates the users of a certain cluster. Also, the scaling should be automated. The following command shows how to scale a deployment of a “quotation-app” on a number of pods between two and five and target the CPU utilization at 50%.

```
kubectl autoscale deployment quotation-app --min=2 --max=5 --cpu-percent=50
```

10 SUMMARY

This thesis presented the development of a quotation system that utilizes Microservices architecture on top of a multi-instance SaaS approach deployed using Docker and Kubernetes.

First, it started by discussing what is a quotation system and what is the purpose of building one that is directed to in-house construction and maintenance companies. Then, the difference between multi-tenant and multi-instance when building SaaS in the cloud and the practice of building a REST API that could be easily integrated with external web service and third-party software and APIs. After that, the web security principles and software containerization concepts were introduced. Next, the process of collecting the system requirements, analyzing them, and design the solution supported by the UML diagrams and the system architecture description. Then, the solution structure and the technology stack were introduced followed by the system implementation of the back-end as API and the front-end web application as a single-page application that consumes that API. Furthermore, the implementation of authorization access on the user and the API level was discussed. Finally, the deployment using Docker and Kubernetes was explained briefly.

It can be concluded that the implemented has a well-designed architecture, and met the expectation of the current customers and the sponsor. Moreover, the system could be improved and optimized, and more features could be added.

REFERENCES

/1/ Business Dictionary. Quotation Definition. Accessed 14.1.2017.
http://www.businessdictionary.com/definition/quotation.html

/2/ Wikipedia. Software as a Service. Accessed 14.1.2017.
https://en.wikipedia.org/wiki/Software_as_a_service

/4/ Wikipedia. Multitenancy. Accessed 14.1.2017.
https://en.wikipedia.org/wiki/Multitenancy

/5/ Service Now. Why Cloud Architecture Matters: The Multi-Instance Advantage over Multi-Tenant. Accessed 14.1.2017.
https://servicematters.servicenow.com/why-cloud-architecture-matters-the-multi-instance-advantage-over-multi-tenant/

/6/ Cloud Tweaks. Cloud Architecture – The Multi-Tenant Versus Multi-Instance Debate. Accessed 14.1.2017.
http://cloudtweaks.com/2016/08/cloud-architecture-multi-tenant-versus-multi-instance-debate/

/7/ Slideshare. Cloud Computing. Accessed 14.1.2017
http://pt.slideshare.net/Agarwaljay/cloud-computing-simple-ppt-41561620

/8/ Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. 2000. University of California, Irvine. Chapter 5: Representational State Transfer (REST). Accessed 21.1.2017
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

/9/ Wikipedia. Single Page Application. Accessed 21.1.2017
https://en.wikipedia.org/wiki/Single-page_application

/10/ AngularJS by Google. Accessed 21.1.2017
https://angularjs.org/

/11/ Ember.js. Accessed 21.1.2017
http://emberjs.com/

/12/ Meteor.js. Accessed 21.1.2017
https://www.meteor.com/

/13/ Mixu's single page app book. Modern web applications: an overview. Accessed 21.1.2017
http://singlepageappbook.com/goal.html

/14/ Wikipedia. Information Security. Accessed 21.1.2017
https://en.wikipedia.org/wiki/Information_security

/15/ SANS. Information Security Resources. Accessed 21.1.2017
https://www.sans.org/information-security/

/16/ Wikipedia. Defense in Depth. Accessed 21.1.2017
https://en.wikipedia.org/wiki/File:Defense_In_Depth_-_Onion_Model.svg

/17/ OWASP. Accessed 21.1.2017
https://www.owasp.org/index.php/Main_Page

/18/ OWASP. OWASP Top Ten. Accessed 21.1.2017
https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

/19/ Business Dictionary. Performance. Accessed 21.1.2017
http://www.businessdictionary.com/definition/performance.html

/20/ Webopedia. Containarization. Accessed 21.1.2017
http://www.webopedia.com/TERM/C/containerization.html

/21/ Docker. Accessed 21.1.2017
https://www.docker.com/

/22/ Docker. What is Docker? Accessed 21.1.2017
https://www.docker.com/what-docker

/22/ Golv1. Accessed 22.1.2017
http://www.golv1.fi/

/23/ James Lewis; Martin Fowler. Microservices. Accessed 25.1.2017
https://martinfowler.com/articles/microservices.html

/24/ UML. What is UML? Accessed 4.2.2017
http://www.uml.org/what-is-uml.htm

/25/ MongoDB. Accessed 5.2.2017
https://www.mongodb.com/

/26/ Redis. Accessed 5.2.2017
https://redis.io/

/27/ Neo4j. Accessed 5.2.2017
https://neo4j.com/

/28/ Wikipedia. MySQL. Accessed 5.2.2017
https://en.wikipedia.org/wiki/MySQL

/29/ Wikipedia. Model-View-Controller. Accessed 5.2.2017
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

/30/ Laravel. Accessed 5.2.2017
https://laravel.com/

/31/ Packagist. Accessed 8.2.2017
https://packagist.org/

/32/ JWT. Introduction to JSON Web Tokens. Accessed 12.2.2017
https://jwt.io/introduction/

/33/ Wikipedia. Security Assertion Markup Language. Accessed 12.2.2017
https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language

/34/ MSDN. Simple Web Token (SWT). Accessed 12.2.2017
https://msdn.microsoft.com/en-us/library/azure/hh781551.aspx

/35/ GIT. Getting Started - Installing Git. Accessed 18.2.2017
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

/36/ Composer. Download Composer. Accessed 18.2.2017
https://getcomposer.org/download/

/37/ Jeffry Way. Laravel-4 Generators. Accessed 18.2.2017
https://github.com/JeffreyWay/Laravel-4-Generators

/38/ Francois Zaninotto. Faker. Accessed 18.2.2017
https://github.com/fzaninotto/Faker

/39/ Sean Tymon. JWT-Auth. Accessed 18.2.2017
https://github.com/tymondesigns/jwt-auth

/40/ Barry vd. Heuvel. Laravel-DOMPDF. Accessed 19.2.2017
https://github.com/barryvdh/laravel-dompdf/tree/0.4

/41/ Intervention. Intervention Image. Accessed 19.2.2017
https://github.com/Intervention/image

/42/ Postman. Accessed 23.2.2017
https://www.getpostman.com/

/43/ npm. Accessed 23.2.2017
https://www.npmjs.com/

/44/ Gulp. Accessed 23.2.2017
http://gulpjs.com/

/45/ AngularJS Team. ngRoute. AngularJS Documentations. Accessed 23.2.2017
https://docs.angularjs.org/api/ngRoute

/46/ Angular UI Team. UI Bootstrap. Accessed 23.2.2017
https://angular-ui.github.io/bootstrap/

/47/ AngularJS Team. ngMessages. Accessed 23.2.2017
https://docs.angularjs.org/api/ngMessages/directive/ngMessages

/48/ Sahat Yalkabov. Satellizer. Accessed 23.2.2017
https://github.com/sahat/satellizer

/49/ G. Kay Lee. ngStorage. Accessed 23.2.2017
https://github.com/gsklee/ngStorage

/50/ Angular Translate Team. Angular Translate. Accessed 23.2.2017
https://github.com/angular-translate/angular-translate

/51/ Angular Base64 Upload. Accessed 23.2.2017
https://github.com/adonespitogo/angular-base64-upload

/52/ Intro.js. Accessed 13.3.2017
http://introjs.com/

/53/ Chrome DevTools Overview. Accessed 15.3.2017
https://developer.chrome.com/devtools

/54/ YSlow. Accessed 15.3.2017
http://yslow.org/

/55/ Docker Hub. Accessed 26.3.2017
https://hub.docker.com

/56/ Kubernetes. Accessed 3.4.2017
https://kubernetes.io/

/57/ CoreOS's etcd. Accessed 3.4.2017
https://github.com/coreos/etcd

/58/ CoreOS's flannel. Accessed 3.4.2017
https://github.com/coreos/flannel