Bachelor’s thesis

Degree programme in Information and Communications Technology

2020

Khoi Huynh

THE DEVELOPMENT OF A WEB APPLICATION

– The new trend - Serverless application

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information and Communications Technology

2020 | 53 pages

Khoi Huynh

THE DEVELOPMENT OF A WEB APPLICATION

- The new trend – Serverless application

Two decades, the time when the web application is just revealed to the world thanks to the invention of the internet, web pages were simply hard-coded with static texts and, images. Gradually, people's curiosity about web applications rapidly increased, then the studying of web development became more popular. In essence, the web architecture is a combination of two programs running concurrently on browsers(frontend side) for sending requests and servers(backend side) for responding. It seems that more and more developers tend to focus on the development of the browser-side. Serverless application is a new technology for those who do not need to understand about server-side aspects since they are managed by the vendors instead.

This thesis aims at providing an overview of the web application evolution, evaluation of different web app architectures, and a detailed introduction of serverless application, as well as discussion of its impact.

There are two main parts in this thesis: the theoretical and the practical. In the theoretical part, some modern popular architectures of web development are introduced. Serverless applications are explained in detail as well as its influence as well as its ecosystem as a new trend. The practical part demonstrates how to create a serverless application from scratch.

The thesis gives a clear understanding of how different the original architecture works compared to serverless through detailed analysis. Moreover, the practical part is utilized as an entry-level method for new-comers to approach serverless architecture.

KEYWORDS:

Single-page application, cloud computing, serverless, Authentication, Angular, MongoDB.

CONTENTS

LIST OF ABBREVIATIONS **6**

1 INTRODUCTION **6**

2 SINGLE-PAGE APPLICATION **8**

2.1 What is a Single-Page Application? 8

2.2 How does a Single-Page Application work? 10

2.2.1 Location Primer 11

2.2.2 Route matching 12

2.3 Pros and Cons 12

2.4 Basics of Angular 13

2.4.1 Architecture 13

2.4.2 Modules 14

2.4.3 Components 16

2.4.4 Services 18

2.4.5 Router 19

3 CLOUD SERVICES AND SERVERLESS COMPUTING **22**

3.1 Cloud computing 22

3.1.1 Service types 22

3.1.2 The upsides and downsides of investing Cloud computing 23

3.2 Serverless 24

3.3 Authentication 24

3.3.1 Sessions 25

3.3.2 JSON Web Token (JWT) 25

3.3.3 Building Custom Function 26

4 EXAMPLE OF BUILDING A LIVE SERVERLESS APPLICATION **27**

4.1 Project intention and requirements 27

4.2 Cloud platform preparation 27

4.3 Client-side framework and Code Editor 31

4.4 Setup project repository 32

4.5 Integrate with MongoDB Stitch 33

4.6 Authentication 36

4.7 Deployment 44

5 CONCLUSION **48**

REFERENCES **49**

FIGURES

Figure 1. Number of Internet Users from 2005 to 2019 [2]. 8
Figure 2. The difference between Traditional web application and SPA [5]. 9
Figure 3. Example of application loading with Stateful SPA [7]. 10
Figure 4. Example of application loading with Stateless SPA [7]. 10
Figure 5. The structure of Location object [7]. 11
Figure 6. Example of common case observed in Chrome DevTool. 11
Figure 7. An example of route configuration in SPA app (Angular). 12
Figure 8. Diagram showing the basic concept of Angular [9]. 14
Figure 9. A NgModule declaration example [10]. 14
Figure 10. Export classes in module to use in other modules [10]. 15
Figure 11. Component declaration example. 16
Figure 12. Component Lifecycle hooks [11]. 16
Figure 13. Forms of data binding in a component [9]. 17
Figure 14. Example of a global scope service (root). 18
Figure 15. How Router render component based on the URL [12]. 19
Figure 16. NgRoute configuration with nested routes example. 20
Figure 17. The router cycle for a route state change [12]. 21
Figure 18. Comparison between management types [13]. 24
Figure 19. Authentication process with JWT. 25
Figure 20. Steps of creating new Stitch App in MongoDB Atlas. 30
Figure 21. Initializing a Collection. 31
Figure 22. Installing Angular CLI. 32
Figure 23. Creating a new directory. 32
Figure 24. Serving the project in localhost. 33
Figure 25. Installing Stitch dependency. 33
Figure 26. Importing MongoDB Stitch. 34
Figure 27. Sample code of Stitch SDK initialization in app.component.ts. 34
Figure 28. Query with Collection in database. 35
Figure 29. The response with data. 36
Figure 30. Rules configuration UI. 37
Figure 31. Editing view of "owner" rule. 37
Figure 32. Editing view of "shared" rule. 38
Figure 33. Sample code of singing up user with credentials. 39
Figure 34. Defining confirmation URL view. 39
Figure 35. Confirming a registered user. 40
Figure 36. Authenticating a user. 40
Figure 37. Authorized domain configuration view in Google Console. 41
Figure 38. Authorized redirect URIs configuration. 42
Figure 39. Setting view in MongoDB Stitch. 43
Figure 40. Login to Google account function. 43

Figure 41. Handling the redirect user data. 43
Figure 42. Deployment history records view. 44
Figure 43. Command to build project into production bundle. 45
Figure 44. The settings of the projects. 45
Figure 45. Uploading the built folder to the platform. 46
Figure 46. Defining the redirect root file. 47
Figure 47. Adding the new address in Redirect URIs settings. 47

LIST OF ABBREVIATIONS

AJAX	Asynchronous Javascript and XML
CLI	Command Line Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DOM	Document Object Model
FaaS	Function as a Service
HTML	Hypertext Markup Language
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
JWT	JSON Web Token
OS	Operating System
PaaS	Platform as a Service
SaaS	Software as a Service
SDK	Software Development Kit
SPA	Single-page Application
URI	Uniform Resource Idendifier
URL	Uniform Resource Locator
VS Code	Visual Studio Code
XML	Extensible Markup Language

1 INTRODUCTION

The idea of sharing information has been raised for a long time. Therefore, the Internet, which is as known as the information superhighway, was born after Tim Berners-Lee's proposal was published in 1989 [1]. Browsers have become a sharp weapon in the industry due to their availability of rendering content being served on a faraway computer. However, the slow speed and shared bandwidth with phone service are its limitations.

Years later, to satisfy users' demand, companies found that deploying and managing a huge number of servers is necessary. This provided opportunities for service providers to take advantage of operating multiple environments on a single host with the purpose of simplifying the deployment and management of the servers. Then Virtual Machines (VM) were receiving support from Datacenter providers. Yet, still, the process of code deployment was not really simple as much of configuration and setup were prerequisites. Fast forward a few more years, Docker, a brighter way for infrastructure providers, was developed. It steadily became well-known because of its investigation for "containers", which was proved that can reduce the developers' worries about deploying applications and provided a portal for delivering code directly into production. Yet, development did not stop there, developers could not stop themselves to curious about a chance to focus on even less. The term "Serverless" immediately attracted developers by its mechanism which allows them to reduce the time and difficulties of handling servers by delegating to the vendors. [1].

Since Serverless is becoming a new trend and thus, it would be an attractive topic to be discussed. Along with this thesis and for demonstration, a serverless application is built with detailed examples to give certain practical points of view about this new technology.

There are five chapters in this thesis:

- Chapter one is the introduction. In this chapter, the history, and the origin of the term serverless will be briefly introduced.
- Chapter two discusses in detail about the Single-page Application, how it works as well as its advantages and disadvantages.
- Chapter three focuses on the main topic of the thesis, serverless applications. This chapter gives a detailed introduction of Cloud computing.

- Chapter four is an example of building a serverless application from scratch. This part will provide step-by-step instructions with the necessary tools.
- The last chapter summarizes all the topics discussed throughout the whole thesis.

2 SINGLE-PAGE APPLICATION

, As technology continuously evolves, so do the internet users' needs. As stated in this graph, the number of internet users highly rocketed during 2010 and 2019 (Figure 1).

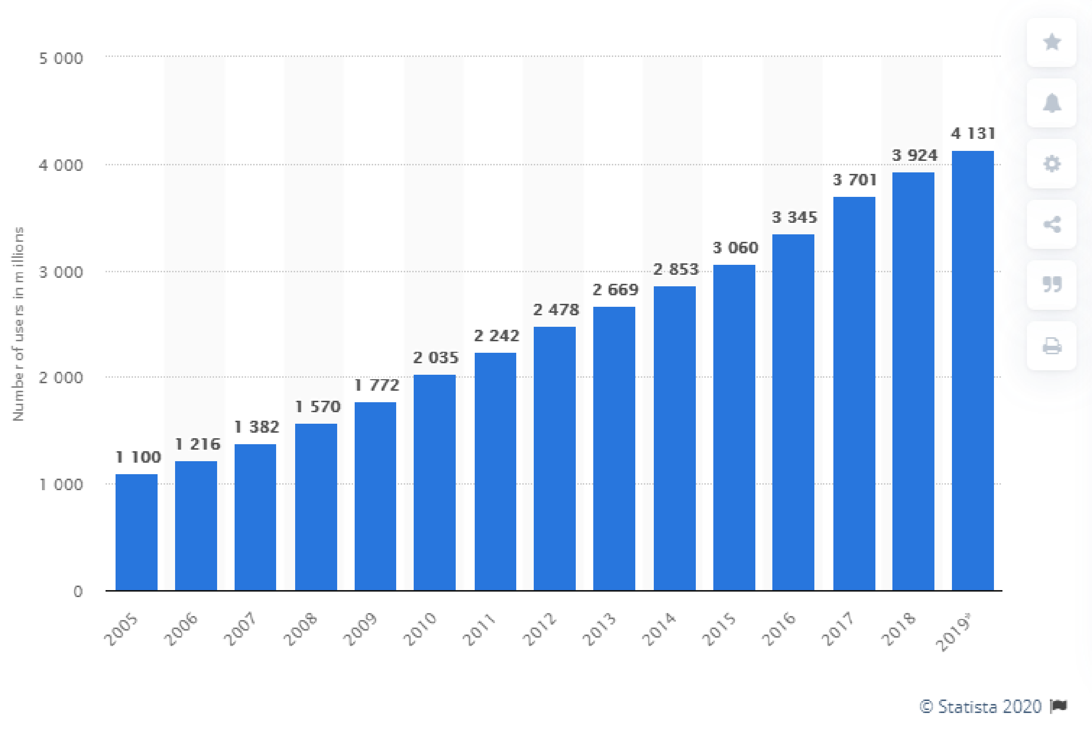

Figure 1. Number of Internet Users from 2005 to 2019 [2].

In this technology evolution, there are also modern approaches to app development to serve the needs of s internet users. One such is the Single-Page Application (SPA).

In this chapter, the definition of SPA and relevant technologies will be discussed deeply. The chapter will discuss how these technologies affect daily life in general and web development specifically .

2.1 What is a Single-Page Application?

A single-page application (SPA) is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web-

server, instead of loading completely new pages from the server each time for a user action. [3]

Particularly, in a more traditional web page architecture, an index.html page might link to other HTML pages on the server that the browser will download and display from scratch, [4]. This means that for every request of fetching data, the server will render the whole web page with new data on the server-side, and then return it as a response. On the contrary, SPAs allows users to keep interacting with the page and updates data concurrently without interruption (Figure 2).

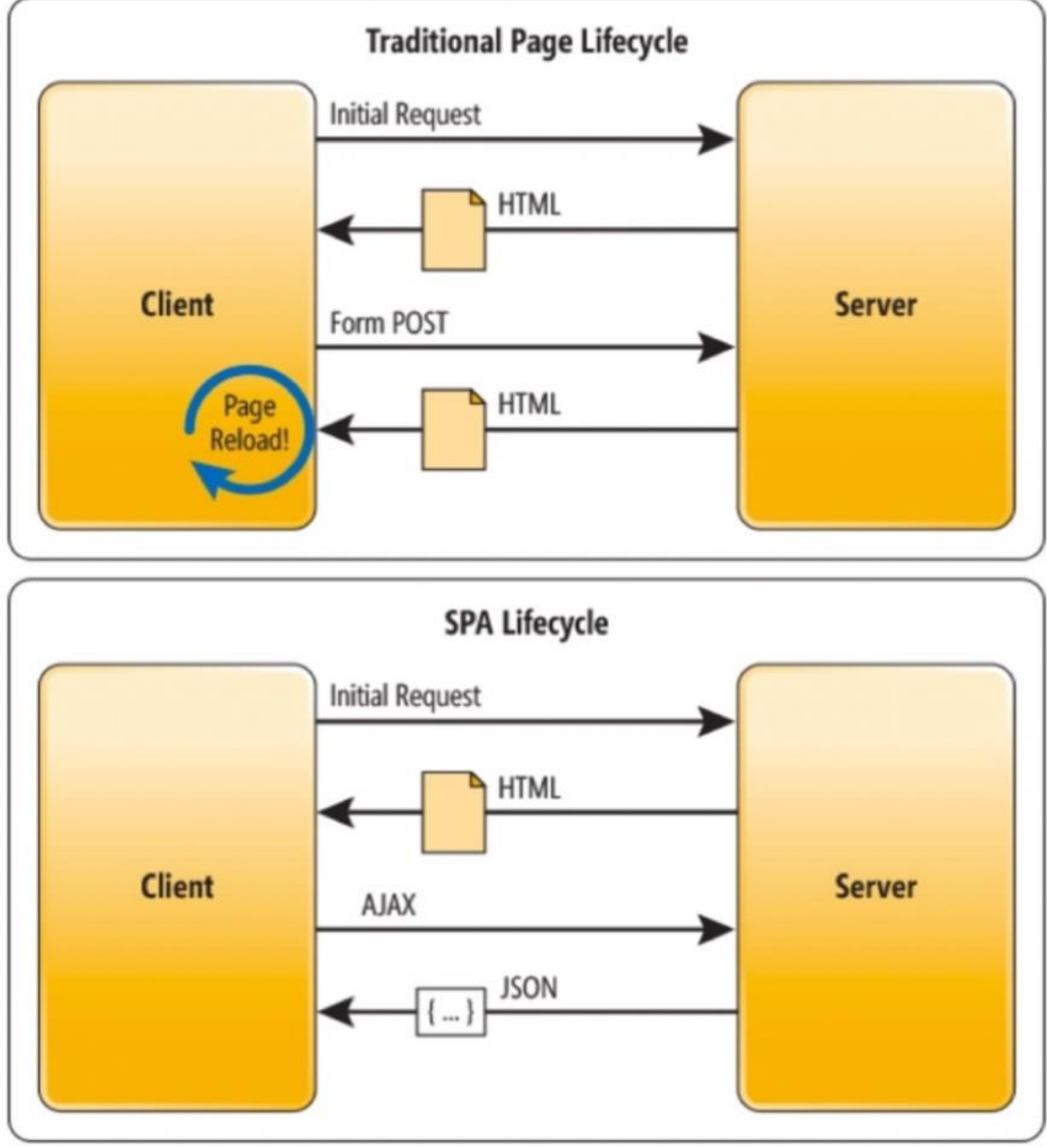

Figure 2. The difference between Traditional web application and SPA [5].

Note: Figure 2 illustrates how the communication between Client and Server using AJAX. Besides, Websocket is an alternative that supports bi-directional communication and provides a low-latency connection [6].

2.2 How does a Single-Page Application work?

As stated in [7], SPA is divided into two types, Stateful (Internal-state), and Stateless (URL-based). The differences come from each type's method of implementation. The stateful type has only one entry, which means that no matter which URL is used to access the application, users will start at the root page (Figure 3). On the other hand, the Stateless type provides the application data based on the URL (Figure 4).

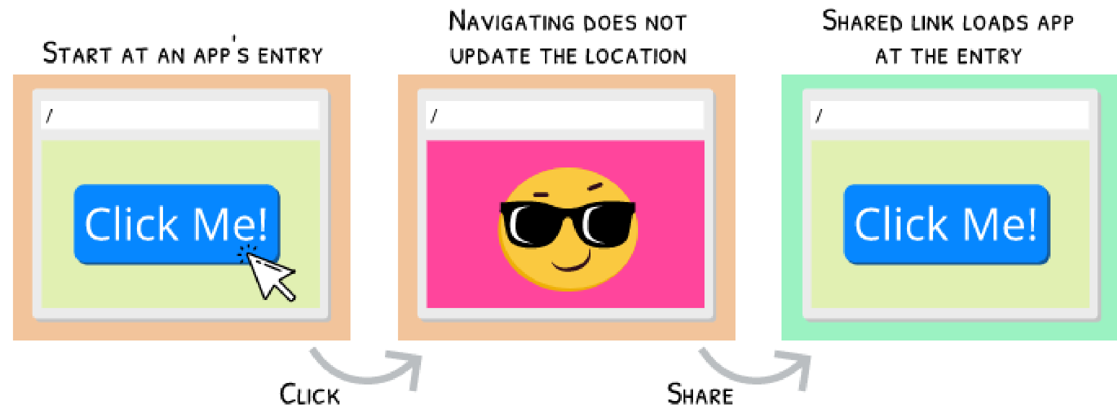

Figure 3. Example of application loading with Stateful SPA [7].

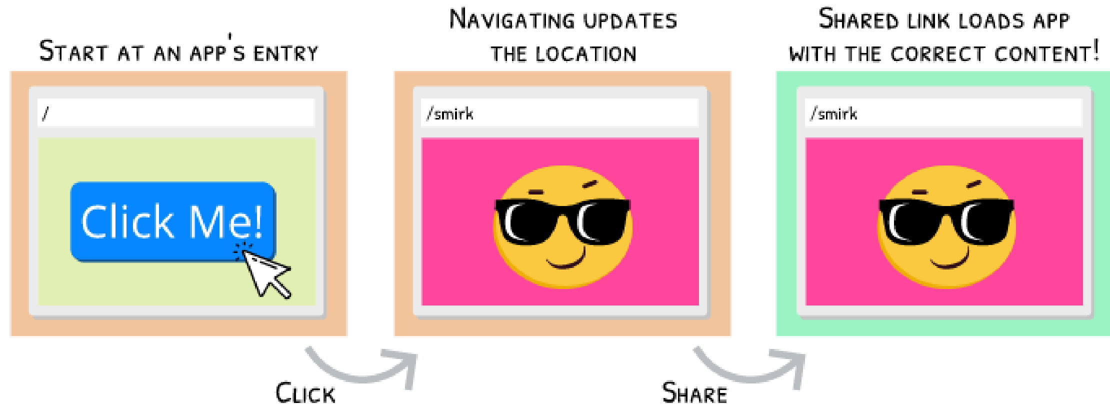

Figure 4. Example of application loading with Stateless SPA [7].

Most of the modern SPAs are Stateless including Angular, which will be used in this thesis. Thus, this part will focus on the Stateless SPA type.

2.2.1 Location Primer

SPAs use "window.location" object to access different parts of the URL for interacting (Figure 5). The parts after the hostname in the URL, particularly pathname, search and hash , are crucial for specifying which content to render, according to [7].

```
https://www.example.com/one?key=value#trending

location = {
  protocol: "https:",
  hostname: "www.example.com",
  pathname: "/one",
  search: "?key=value",
  hash: "#trending"
}
```

Figure 5. The structure of Location object [7].

In some common cases, the search property could be included in the pathname as a parameter following the format **pathname/:search**, and the **hash** can be deducted (Figure 6).

```
pathname: "/notes/5d4d6df734b9b19bf57f797f"
search: ""
hash: ""
href: "https://notes-app-kpbad.mongodbstitch.com/notes/5d4d6df734b9b19bf57f797f"
```

Figure 6. Example of common case observed in Chrome DevTool.

2.2.2 Route matching

SPAs mostly rely on a router, which is made of routes describing the location they should match. These routes can be static (/auth) or dynamic (/notes/:id, in which the id is a parameter). For example:

```
const routes: Routes = [
  { path: 'notes', component: NotesListComponent,},
  { path: 'notes/:id', component: NoteEditComponent, resolve: [NotesResol
verService] },
  { path: 'notes/new', component: NoteEditComponent },
];
```

Figure 7. An example of route configuration in SPA app (Angular).

In the example above (Figure 7), each object is a route. Route matching is comparing the current location to each route and find the one that matches. The router then will trigger a re-render of the application with corresponding data.

2.3 Pros and Cons

Besides the basic difference between original architecture and SPA, there are some more convincing reasons to invest this kind of web application, as demonstrated in [8].

The first reason is SPA means faster applications. The fewer requests to the server, the less waiting time for the response. Since only data is transferred, hence the bandwidth usage is also reduced. Additionally, in most cases, SPA stores data locally for use even in offline mode.

A faster application leads to better user experience. The page is dynamically rendered based on the data change, so there is no more waiting time to reload the whole page. The users will not face any kind of interruption in the meantime. Consequently, SPA creates a higher level of user experience.

SPAs are super easy to deploy in production. There is just one HTML file, one CSS bundle, and one JavaScript bundle which could be hosted in any static content server.

However, there are also some drawbacks to consider, mentioned by ElSayed [8]. The critical thing about SPAs is that it has poor Search Engine Optimization. Since there is

only the initial index.html file, the Search Engine cannot index the content. Moreover, the complexity is also one of the downsides of SPA. Developers find that the SPA building process is complex and requires different dynamic approaches.

2.4 Basics of Angular

There are a couple of libraries and frameworks for building SPAs such as React, Vue, Svelte, and Angular. Each has both good and not good sides. This section will shortly introduce the basis of Angular, which is supported not only on web but also on various platforms (desktop, mobile).

2.4.1 Architecture

Angular is a TypeScript based framework that provides libraries that can be imported to the SPA. An Angular app is a set of basic building blocks call **Modules** which are compilations of **Components**. A component defines views, which are sets of screen elements that Angular will handle correspondingly with program logic and data. And **Services**, which deal with data or logic that is not directly related to views and shareable across components, can be injected into components as dependencies to make the code more modular and reusable (Figure 8). [9]

Each factor is a class and the specific decorator that Angular marks each of those will determine what they are (component, service, directives, .etc.).

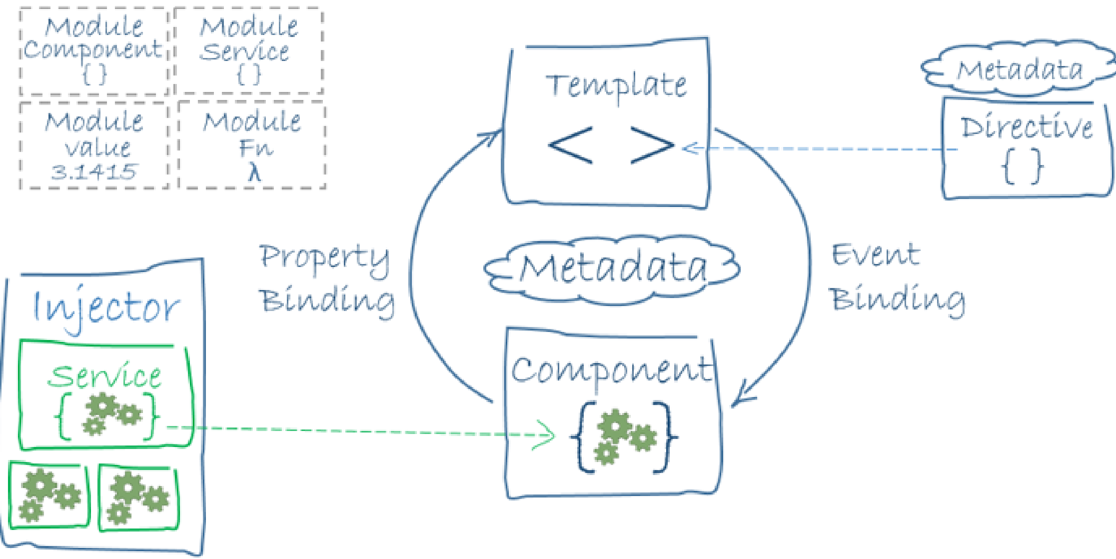

Figure 8. Diagram showing the basic concept of Angular [9].

2.4.2 Modules

Modules are those composed an Angular application. This part will dive deeply about NgModules or Angular Modules.

For app with great scale and high level of complexity, it is a good idea to explicitly declare things and group them together, as stated in [10]. It is also the main purpose of NgModules.

```
import { NgModule } from '@angular/core';

import { SomeComponent } from './some.component';
import { SomeDirective } from './some.directive';
import { SomePipe } from './some.pipe';
import { SomeService } from './shared/some.service';

@NgModule({
  declarations: [SomeComponent, SomeDirective, SomePipe],
  providers: [SomeService]
})
export class SomeModule {}
```

Figure 9. A NgModule declaration example [10].

Angular groups things with two main structures (Figure 9):

- **Declarations** are for those to be used in templates, which are components, directives, and pipes.
- **Providers** are for the classes that hanle data called services.

Scopes

The scope of declared classes are local, which means that they are usable with the module. They must be exported for other modules to use (Figure 10).

```
7   @NgModule({
8     declarations: [SomeComponent, SomeDirective, SomePipe],
9     exports: [SomeComponent, SomeDirective, SomePipe]
10  })
11  export class SomeModule {}
```

Figure 10. Export classes in module to use in other modules [10].

While on the other hand, services, which are in the global scope, are injectable for the whole app.

Import strategy

The difference between scopes leads to confusion when and how the module should be imported into other modules. [10]. Modules are classified into types:

Modules to import only once (normally are modules for the use of services):

- HttpClientModule, which is an Angular built-in module for handling client-side requests.
- Featured modules that provide services only.

Modules to import multiple times when necessary (modules for the use components, template-related things such as directives, pipes):

- UI modules
- CommonModule, which gives all the basics of Angular.
- Featured modules exporting components, directives, .etc.

2.4.3 Components

A component is a basic building block of UI in Angular with metadata determining how the component is used at runtime (Figure 11). [9]

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-loading',
  templateUrl: './loading.component.html',
  styleUrls: ['./loading.component.scss']
})
export class LoadingComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
```

Figure 11. Component declaration example.

Lifecycle hooks

A component control and handle its behavior with various stages called lifecycle hooks (Figure 12).

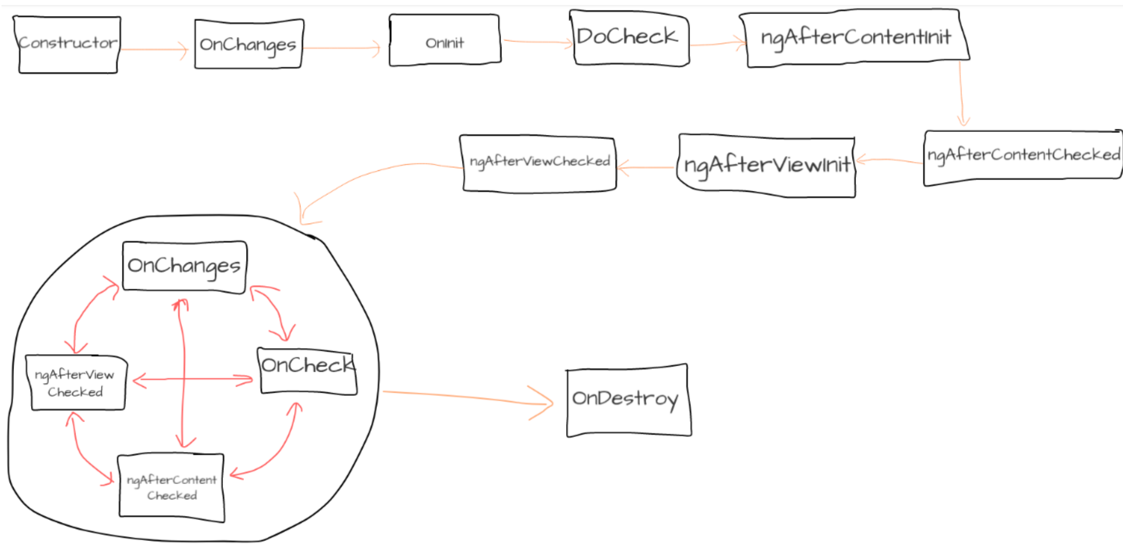

Figure 12. Component Lifecycle hooks [11].

- Constructor: runs when the component is activated which will initialize and inject all services.
- ngOnChanges: render the Document Object Model (DOM) multiple time whenever an input property’s value is changed before it binds to view.
- ngOnInit: runs only once after the initialization of all properties.
- ngDoCheck: activates everytime there is a change of component’s properties.
- ngAfterContentInit: called after the component’s view values changed.
- ngAfterContentChecked: this hook runs after Angular checked whether the content has been projected into the component.
- ngAfterViewInit: runs after the view has been initialized.
- ngAfterViewChecked: called after ngAfterViewInit to check if view has any changes.
- ngOnDestroy: only called when the component is removed.

Data bindings

Angular supports a mechanism called Data binding to take over the pushing data into HTML controls and turns user responses into actions and value updates. Data binding plays an important role in communication not only between a template and its component but also between different components. [9]

There are several forms (Figure 13):

Figure 13. Forms of data binding in a component [9].

- Interpolation is the most basic way to bind data that fills value from the component to the template.
- Property binding passes data one-way from the component to the property of another component in the template.
- Event binding links any event triggered in the DOM to the component.
- Two-way binding allows property and event to be bound from component to the DOM and vice versa. Data flows from the component to the HTML controls as with property binding, and synchronous any changes made to those controls back to the component.

2.4.4 Services

An application always works with data that can be handled inside any component. However, as the application scale expands, data usage also significantly increases. Hence, managing data within separated components leads to high complexity and inconsistency. Services, which provides data or logic across the app, are considered the solution. A NgService is typically a decorator marked Javascript class (Figure 14) with a well-defined specific purpose that enhances the modularity and reusability, not to mention that it helps the components lean and more efficient. [9]

```
@Injectable({
  providedIn: 'root'
})
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

Figure 14. Example of a global scope service (root).

2.4.5 Router

The Router Module gives the ability to display components dynamically based on the URL. For example, the HomeComponent and NotesComponent should be displayed correspondingly as below (Figure 15):

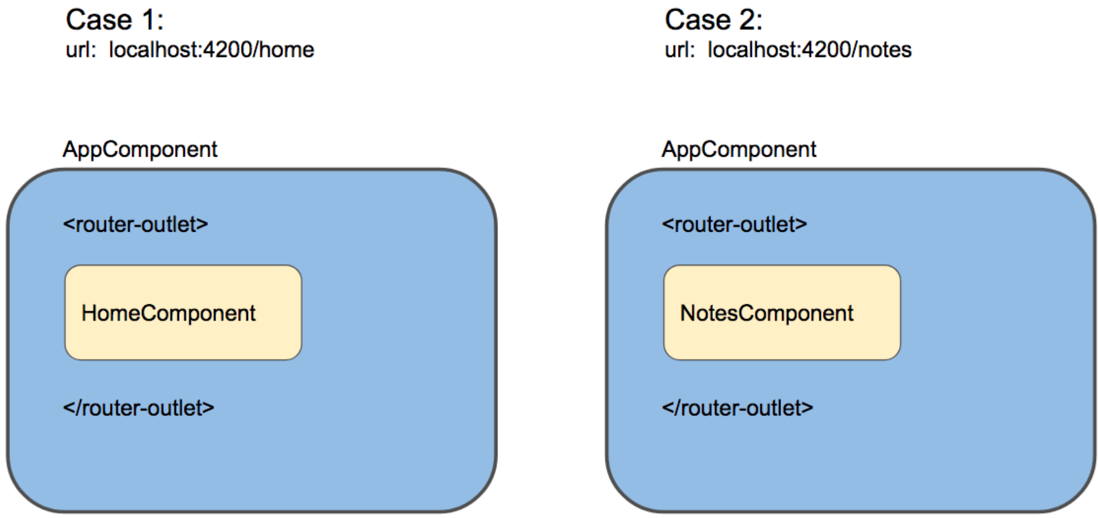

Figure 15. How Router render component based on the URL [12].

Each of the above routable components is considered a router state in a tree of router states, which can be configured with Router Module.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'notes',
    children: [
      { path: '', component: NotesComponent },
      { path: ':id', component: NoteComponent }
    ]
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figure 16. NgRoute configuration with nested routes example.

As defined, each state will match a component. Angular Router also provides nested-route. In particular, for /notes, there are two sub-routes defined as /notes/ and /notes/:id (:id is a query string). For each **id**, the NotesComponent should contain a different instance of NoteComponent as a child route with different data. (Figure 16)

Router Lifecycle

The Router also has a chain of steps to cycle through every single time that the state changes (Figure 17).

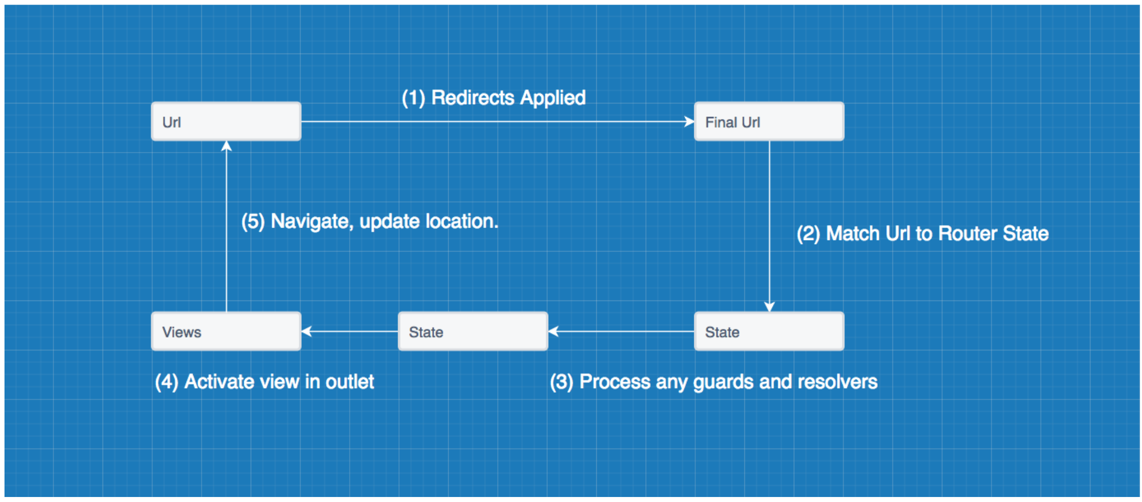

Figure 17. The router cycle for a route state change [12].

- (1) A redirect occurs, then the URL must be processed. The router does not redirect at this stage, so the app stays unchanged.
- (2) The router uses a first-match strategy to match the url to a route state that already defined in the configuration.
- (3) In this stage, the router triggers an extra-check whether there is any guard that prevents navigation for the matched route.
- (4) The corresponding component is rendered.
- (5) The cycle finishes and waits for another redirect.

Lazy loading

As a large-scale application grows and has more feature modules over time, the increasingly long loading time thus leads to bad user experience. To solve this problem, Angular supports a mechanism, called Lazy loading, which will only load the required features on demand when users navigate instead of all of them at the entry time.

3 CLOUD SERVICES AND SERVERLESS COMPUTING

Building only a good client-side SPA is not enough for production. Like any application, an SPA also needs to deal with data and servers. However, deploying a server is a tremendous commitment. To get everything up and running, operators have to manage all the Server, Operating System (OS) and Network, required dependencies and the maintenance for years. Virtual machines (VMs) are the alternative solution for a while but keeping them running does not get rid of the heavy workload of server management. Furthermore, VMs deployment charges per-second even the system is idle and the resources are not in use. [13]

Cloud computing rapidly becomes an alternative that plays a crucial role in serverless computing. The next section will shortly describe Cloud computing and Serverless terminology.

3.1 Cloud computing

Generally, cloud computing is that all hardware and software needed to serve a website is provided by an outsourcing company as a service, which can be accessed over the internet. Developers are not required to manage how the platform performs to keep applications operating. Thus, organizations become more agile and manage expenses better.

3.1.1 Service types

Infrastructure as a Service (IaaS)

IaaS provides virtualized underlying resources such as OS, networking, etc. over the internet for developing applications. For instance, users buy access to the raw computing hardware like servers or storage with a monthly charge to host a website. IaaS reduces the management expenses for building their own datacenter as well as prevent the high complexity.

Software as a Service (SaaS)

SaaS is a method of delivering not only the infrastructure but also the software application. The provider takes care of the security and availability of both operation and maintenance. One case is that Web-based email in which Google power from developing to serving the application over the interconnected network.

Platform as a Service (PaaS)

PaaS is the cloud service is the mixture of the other mentioned types that deliver an on-demand environment for the software developing cycle including building, testing, deployment and management.

3.1.2 The upsides and downsides of investing Cloud computing

As mentioned by Jain in [14], the first reason for using Cloud service is that it addresses the wasted resource issue. And the application billing plan can be modified quickly according to the application scaling. The app globalization is easier with Cloud provider support. Cloud computing refers deliver the proper amount of resources from the right geolocation, serves applications in multiple domains all over the world, and increases the user experience. Losing important data due to disaster or unexpected incident is a nightmare. Using cloud services means all data is backed up and protected by the provider, making it more invulnerable.

Everything has both sides, even Cloud service can issue some problems, taking care of what is coming is never redundant. Using cloud computing means buying services, or renting. Thus, upfront costs become ongoing operating costs and may be more expensive in a long time instead of owning the system. Furthermore, the dependence on the supplier restricts the users need as off-the-peg solutions are not available. Last but not least, the service's availability is not manageable. If the providers stop supporting, users have no guaranty to keep the application working.

3.2 Serverless

Hence, Serverless computing potentially makes a better solution as it provides "pay-as-you-go" payment method. That means there is no more wasted budget if the system is not currently in use. Moreover, the platforms conduct all the deployment procedures (Network, OS, Maintenance, etc.) automatically (Figure 18).

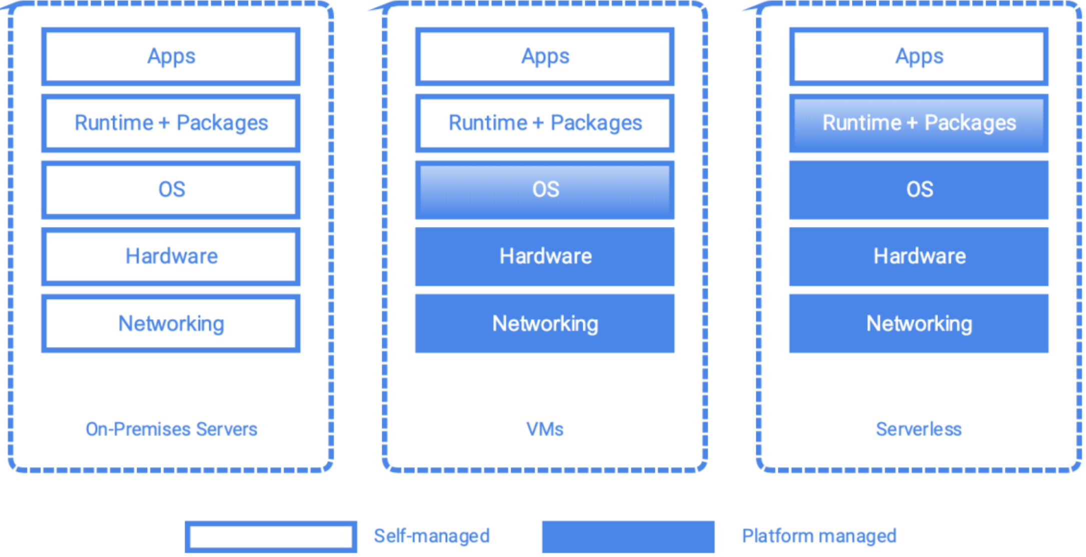

Figure 18. Comparison between management types [13].

Serverless computing is exactly a cloud computing model wherein the developers have the minimal visibility and control on the above server stuffs – kernel, file system, OS and so forth.

Function as a Service (FaaS) is the computation aspect of serverless. This allows developers to upload modular codes as functions into the cloud and executes them independently as a REST server.

3.3 Authentication

User authentication was always an important topic, more precise for serverless applications. Since there are no real servers managed by developers, this issue has to be getting more consideration.

3.3.1 Sessions

According to [15], Sessions, a standard approach for handling user identity, may lead to a high datastore cost because of retrieving sessions and prolong loading time. The reason is that for every request sent, ther server has to look up user and all related information in the database, which makes the response time longer.

3.3.2 JSON Web Token (JWT)

JWT is said to be more convenient for Serverless applications. As JWT allows storing additional information directly into the token other than just credentials itself, the server does not need to look up those in the database anymore. [15]

Figure 19 shows how JWT is used for authenticating users. When a user logs with their credentials, which is then forwarded to the Authorizer, the JWT will be returned in success. Every request made from the user to the Serverless platform should include the JWT. The platform will validate the JWT before granting access to protected resources. [16]

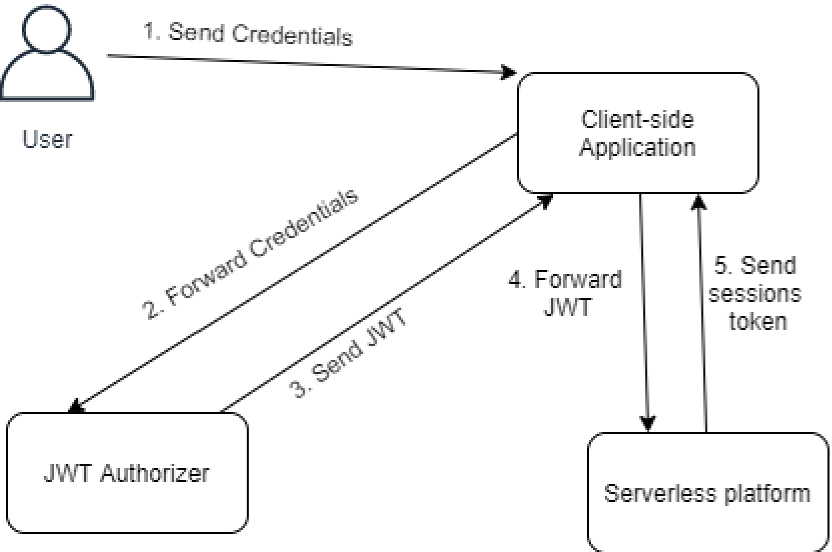

Figure 19. Authentication process with JWT.

3.3.3 Building Custom Function

As FaaS is a key characteristic of Serverless, developers are able to implement custom functions with their authentication logic, validation strategy, etc.

4 EXAMPLE OF BUILDING A LIVE SERVERLESS APPLICATION

4.1 Project intention and requirements

According to the mechanism of serverless, no infrastructure management is needed. However, there is also a difficulty as the understanding of a targeted platform provider is a crucial part of the whole development process. Fortunately, this new developer generation is strongly backed by "big whales" companies. Particularly, there is a variety of cloud-platform providers to be considered such as Azure, Amazon Web Service (AWS), Google Cloud Platform, MongoDB Stitch, .etc with lots of services like Authentication, Database Management, and Hosting.

This part gives a detailed view of what is necessary for building a serverless application by a working demo project. Since this is only for demonstration, the technologies and tools should possibly be a low-cost budget plan and with basic features.

The project intends to provide an online note-taking application where people can create and manage amazing notes. Taking-note used to be personal, but in the present life in which technology is blooming thanks to the internet, people tend to socialize daily activities, some even love sharing their very confidential to the world. Thus, making a greatly meaning note to be collaborative to everyone is not an exception. This project is to give people a mini social network for nothing more than sharing their useful notes.

Besides how to build a complete serverless application, additional technologies and useful tools will be introduced along with this chapter as well.

4.2 Cloud platform preparation

The very first step is to choose a cloud platform. According to the requirements, it should be one that can provide a real-time database management system with third-party authentication support also.

MongoDB Stitch is the chosen one as it can satisfy the requirements quite well. The next section will demonstrate how to get started with a MongoDB Stitch application.

Create a new MongoDB account

Go to the registration address: https://account.mongodb.com/account/register and sign up by going through these steps:

1. Provide the following:
   - **Email address**
   - **First Name and Last Name**
   - **Password**
   - **Phone Number**
   - **Company**
   - **Job Function**
   - **Country**
2. Review and accept the **Privacy Policy** and the **Terms of Service.**
3. Click **Sign up.**

Create an Atlas cluster

After successfully completed. MongoDB Atlas can be accessed by using MongoDB Account. After logging into MongoDB Atlas, create an Atlas cluster with these steps:

1. Click **Clusters** in the left navigation pane, then **Build New Cluster** button.
2. In **Create New Cluster** page, choose preferred provider and region, tier. The corresponding cost will be the displayed at the bottom. This project uses **Free Tier (M0)** for the demo purpose only.
3. Name the cluster (**NotesApp**) and click **Create Cluster** to finish.

Add a Stitch App (Figure 20)

1. Click **Stitch** in the left-hand navigation.
2. Click **Create New Application**.
3. In the poped-up window, fill in the required fields:
   - **Application Name (notes-app).**
   - Select the cluster (**NotesApp**) from the **Link to Cluster** dropdown. Stitch will automatically create a MongoDB service that is linked to this cluster.

- Enter the name of the service (**mongodb-atlas-notesapp**) that Stitch will create.
- Choose **Global Deployment** model and **Region**. Then click **Create**.

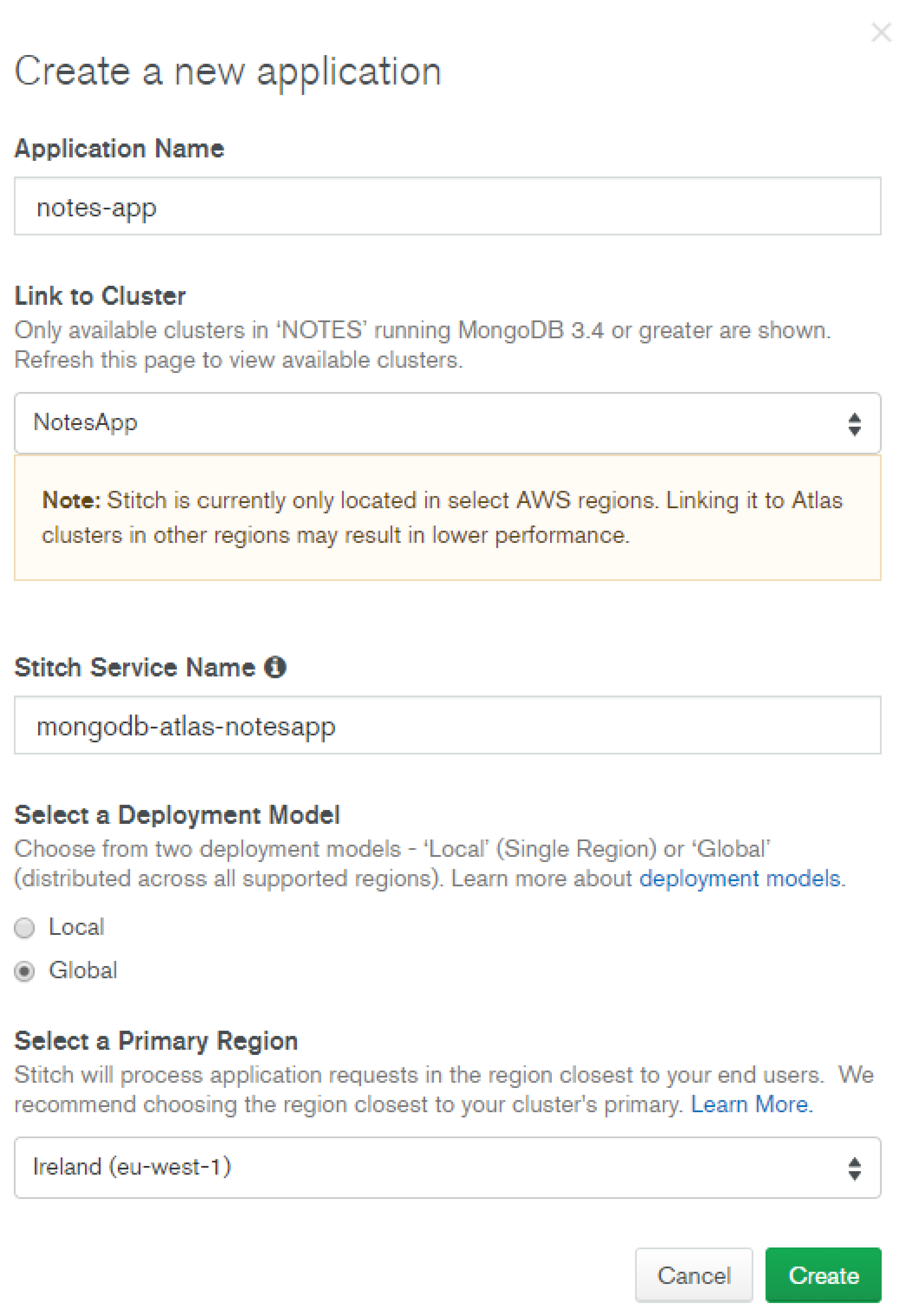

Figure 20. Steps of creating new Stitch App in MongoDB Atlas.

Create a database and add collections (Figure 21)

Navigate to Stitch dashboard by clicking Stitch in the left pane. Scroll down to the Getting started section and do these steps:

1. Skip **Turn on Authentication** as this step will be mentioned in detail later in this part.
2. **Initialize a MongoDB Collection**:
   - The **Linked Cluster Name** will be automatically selected.
   - A collection must belong to a database (A Stitch app can have multiple databases). Create a new **Database name.**
   - Then, enter the **Collection name**.
   - Click **Add Collection**.

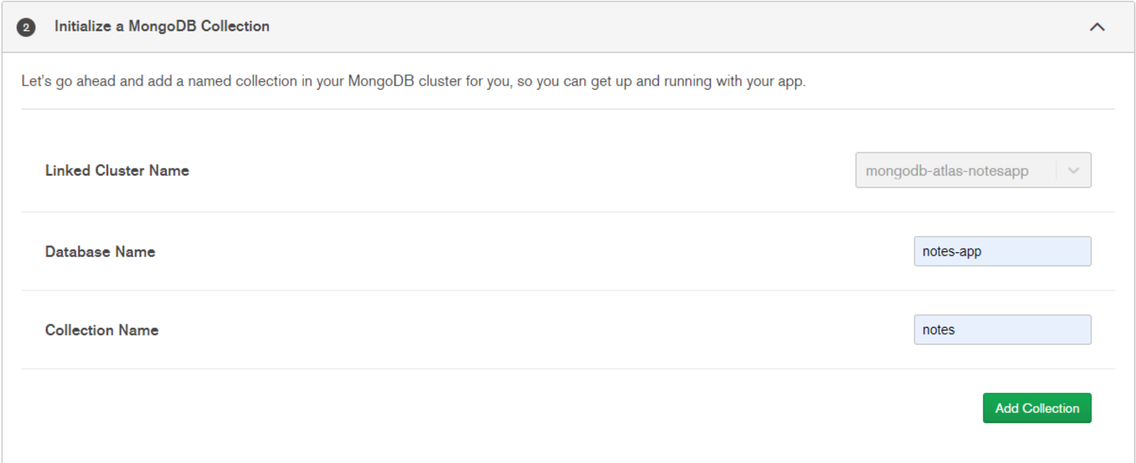

Figure 21. Initializing a Collection.

4.3 Client-side framework and Code Editor

This part will introduce the basis of Angular, one of the modern Javascript Framework, as there is no restriction on specific frameworks or languages.

A suitable code editor also plays an important role as it can boost the speed of the development process. Some are favored more by developers such as Sublime, Vim, Atom and Visual Studio Code (VS Code). All give a beautiful visual user interface.

Sublime will come with a charged fee, Atom and Vim are quite slow compared to others. Thus, VS Code will be used in this thesis because it is not only fast and light-weight but also provides a huge extensions market which can help developers save more time for programming.

4.4 Setup project repository

Install Node.js

Angular requires a current, active LTS, or maintenance LTS version of **Node.js**.

- To check Node.js version, run **node -v** in a terminal/console window.
- To get Node.js, go to nodejs.org.

Install Angular Command Line Interface (CLI)

Angular CLI can create projects and a variety of ongoing tasks such as Testing, Bundling, and Deployment.

Open a **terminal/console** window and enter command (Figure 22):

```
npm install -g @angular/cli
```

Figure 22. Install Angular CLI.

Create a application with CLI

After successfully installing CLI, type the command below to create a project named **my-app** (Figure 23):

```
ng new my-app
```

Figure 23. Create new directory.

Run the application

1. Open **my-app** folder with VS Code.
2. Open the integrated **terminal** in VS Code by pressing **Ctrl + `.**
3. Launch the project by typing this command (Figure 24):

```
cd my-app
ng serve --open
```

Figure 24. Serve the project in localhost.

Now the project is up and running locally.

4.5 Integrate with MongoDB Stitch

This step will illustrate how to connect the project to the Stitch project.

Install Stitch SDK

Navigate to **my-app** project folder. Open **terminal** and type (Figure 25):

```
npm install mongodb-stitch-browser-sdk --save
```

Figure 25. Install Stitch dependency.

The Node Packages Manager (NPM) will download and add StitchAppClient SDK to the repository.

Note: the **--save** flag will update the packages version to package.json file.

Import Stitch Dependencies

```
import {
  Stitch,
  RemoteMongoClient,
  BSON
} from 'mongodb-stitch-browser-sdk';
```

Figure 26. Import MongoDB Stitch.

Note: The import is required for every .ts files in the project where the APIs are called (Figure 26).

Initialize the global AppClient instance in the main ts file (Figure 27)

```
import { Component, OnInit } from '@angular/core';

import { Stitch } from "mongodb-stitch-browser-sdk";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  title = 'Notes';

  ngOnInit(): void {
    //Called after the constructor, initializing input properties, and th
e first call to ngOnChanges.
    //Add 'implements OnInit' to the class.
    Stitch.initializeDefaultAppClient('<APP-ID>');
  }
}
```

Figure 27. Sample code of Stitch SDK initialization in app.component.ts.

The application will instantly connect to MongoDB services once it runs. This next step will show an example of querying data from Stitch.

Get all records from notes collection (Figure 28)

```
@Effect() getNotes = this.action.pipe(
    ofType(NotesActions.actionTypes.GET_NOTES),
    switchMap((actionData: NotesActions.GetNotes) => {
      /** Select the database in Stitch via the AppClient */
      const db = Stitch.defaultAppClient.getServiceClient(RemoteMongoClie
nt.factory, 'mongodb-atlas-notesapp').db('notes-app');
      /** Return the notes collection to process */
      return from(db.collection('notes').find().toArray())
        .pipe(
          map((notes: any[]) => {
            /** Log the result to the screen */
            console.log('Notes Effect Sync: ', notes)
            const processedNotes = notes.map((note) => {
              return new Note(note.content, note._id.toString(), note.use
r, note.createdDate, note.isShared)
            })
            return new NotesActions.SyncNotes({ notes: processedNotes })
          })
        )
    })
  )
```

Figure 28. Query with Collection in database.

The returned data is logged into the screen (Figure 29).

```
Notes Effect Sync:                                  notes.effects.ts:23
▼(8) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▼0:
    ▶_id: ObjectId {id: Uint8Array(12)}
     user: "5d399d0f88657438a5cbe512"
     content: "hippo 1"
    ▶createdDate: Fri Jul 26 2019 14:34:34 GMT+0300 …
    ▶__proto__: Object
  ▶1: {_id: ObjectId, user: "5d399d0f88657438a5cbe5…
  ▶2: {_id: ObjectId, user: "5d399d0f88657438a5cbe5…
  ▶3: {_id: ObjectId, user: "5d345b2c6f1d968fe6e43b…
  ▶4: {_id: ObjectId, user: "5d345b2c6f1d968fe6e43b…
  ▶5: {_id: ObjectId, user: "5d345b2c6f1d968fe6e43b…
  ▶6: {_id: ObjectId, user: "5d345b2c6f1d968fe6e43b…
  ▶7: {_id: ObjectId, user: "5d399d0f88657438a5cbe5…
   length: 8
  ▶__proto__: Array(0)
```

Figure 29. The response with data.

4.6 Authentication

Besides the CRUD functions for managing data, Stitch also supports users with its authentication or integration with third-party providers.

As data is sensitive, even for sharing purposes, well data management with restrictions is a really good investment. Regarding this, Stitch provides rules on each collection and fields which are modifiable to make data invulnerable from unauthorized access.

This section gives a simple case of how to integrate the authentication service.

Define rules in Stitch (Figure 30)

1. Log into MongoDB Stitch and navigate to the notes-app application. Click Rules in the left pane.
2. Add a new MongoDB Collection Namespace by selecting Add Database/Collection in the context menu of the linked cluster.
3. Choose a rule template, this could be optional. The result should look like this:

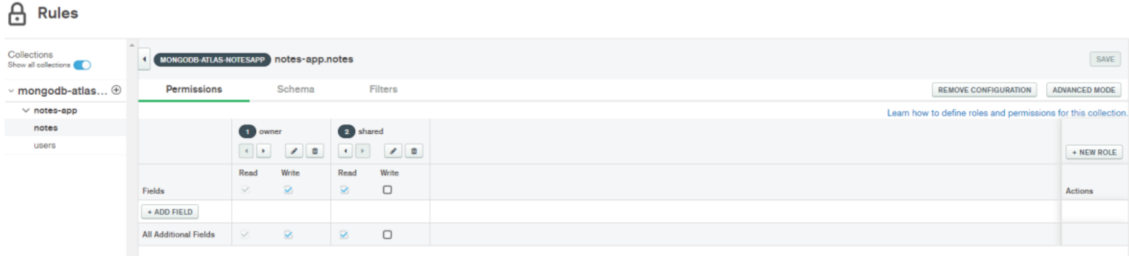

Figure 30. Rules configuration UI.

4. Define “Apply When” Condition

For notes collection, there are 2 roles (Figure 31 and 32):

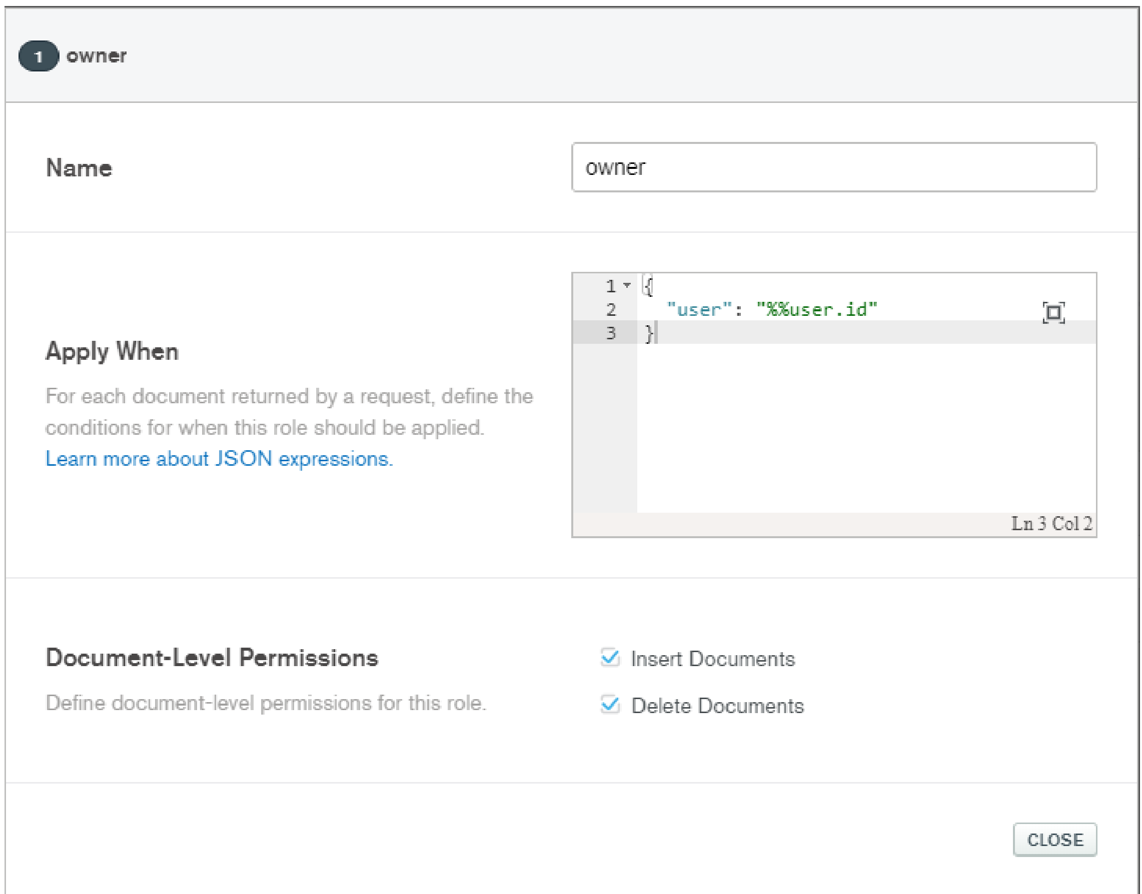

Figure 31. Editing view of "owner" rule.

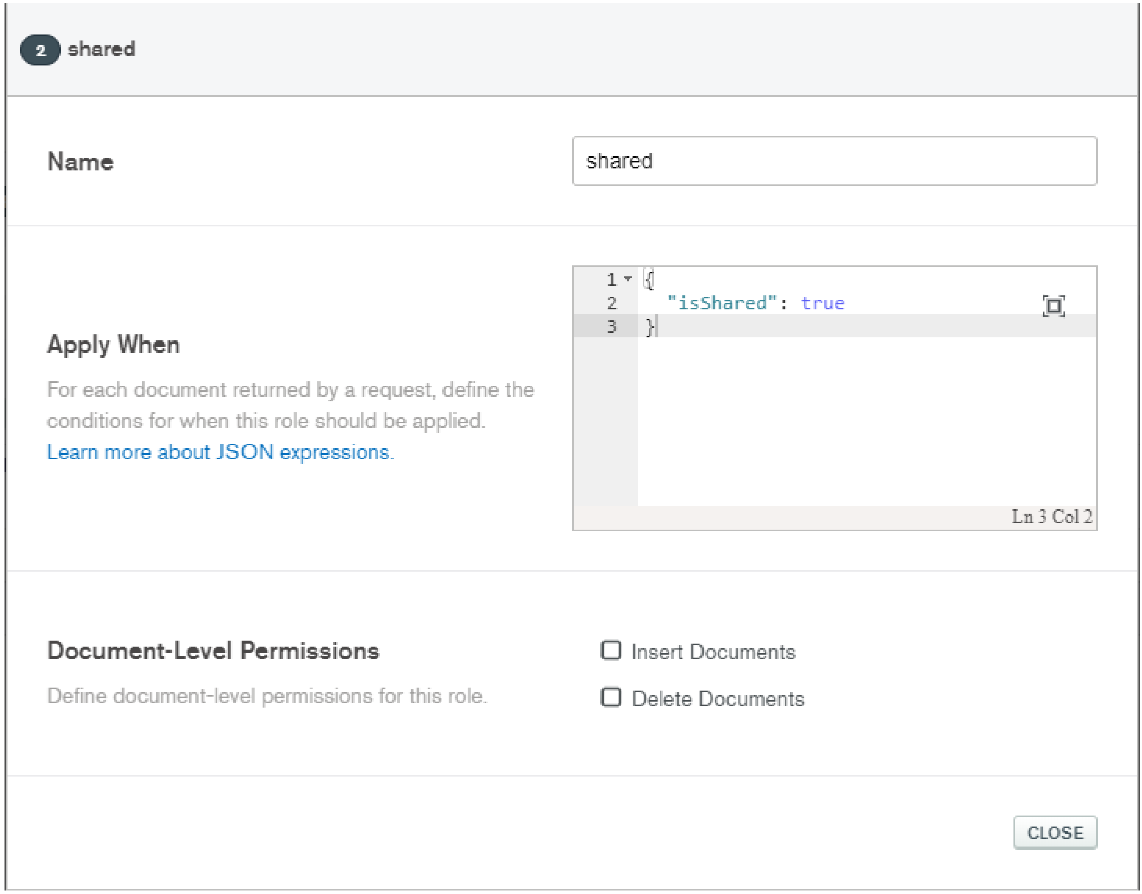

Figure 32. . Editing view of "shared" rule.

A role's "Apply When" condition written in JSON Expression determines whether the role applies to a particular document for the user that issued a query.

These two mean a logged-in user has full permissions with their own records in notes collection, and can only read a note if it is shared.

Stitch supports integrated authentication services from several providers for client-application including Facebook, Google or with only Email/Password. This section will go through how to implement each of the mentioned above.

Email/Password with Stitch

1. Register new user with provided Email/Password (Figure 33)

```
/** Register new user with email/password */

Stitch.defaultAppClient.auth.getProviderClient(UserPasswordAuthProviderCl
ient.factory)

.registerWithEmail(requestData.payload.email, requestData.payload.passwor
d)

.then(() => console.log("Successfully sent account confirmation email!"))

.catch(err => console.error("Error registering new user:", err))
```

Figure 33. Sample code of singing up user with credentials.

After the application sends a request, Stitch sends an email with a **pre-set link** including the token which opens the confirmation URL. The confirmation URL then processes the token in the final step to create the user object in the Stitch database.

2. Configure the comfirmation URL (**pre-set link**) (Figure 34)

Click **Users** tab in the left pane in Stitch page.

Choose **Providers** tab. Select **Email/Password.**

Enable and enter the **Confirmation URL** (specific route of the application to handle token).

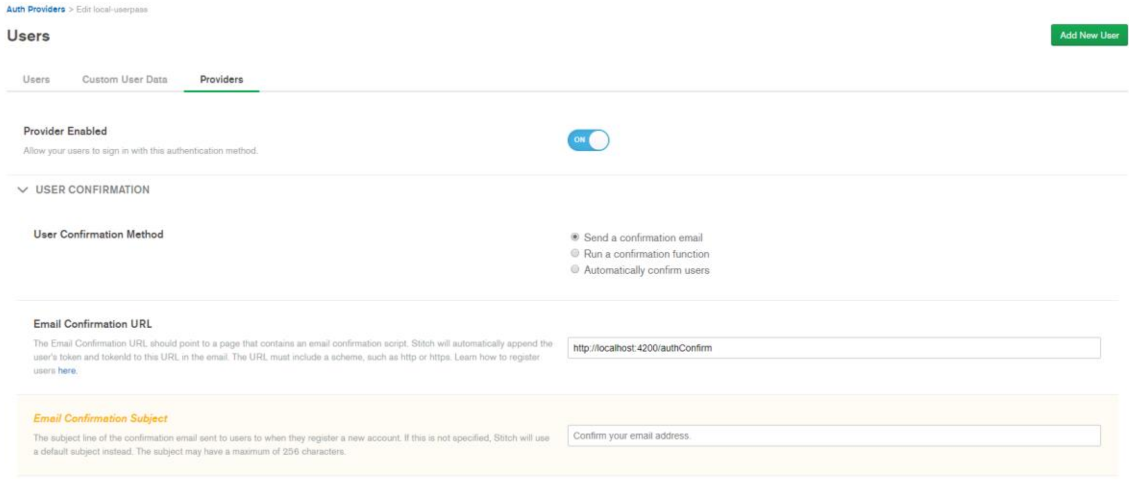

Figure 34. Define confirmation URL view.

Enter **Email Subject** and **Save**.

3. Confirm and create User Object (Figure 35)

```
/** Send request to Stitch to create User object */

Stitch.defaultAppClient.auth.getProviderClient(UserPasswordAuthProviderClient.factory)

.confirmUser(confirmationCredentials.token, confirmationCredentials.tokenId)
```

Figure 35. Confirm registered user.

4. Authenticate registered user (Figure 36)

```
/** Authenticate user with provided Credentials */

const credentials = new UserPasswordCredential(authData.payload.email, authData.payload.password)

Stitch.defaultAppClient.auth.loginWithCredential(credentials)

.then(authedUser => console.log(`successfully logged in with id: ${authedUser.id}`))

.catch(err => console.error(`login failed with error: ${err}`))
```

Figure 36. Authenticate user.

Third-party provider (Google)

To identify a user from Google as a third-party service, Stitch needs OAuth 2.0 access token provided by Google when a user logs in successfully. Stitch uses the token to obtain approved data from Google APIs.

1. Setup required configurations

Create a Google Account and sign in to the Console Cloud Platform at: https://console.cloud.google.com/.

Open **Navigation Menu**, choose **APIs and Services** and select **Credentials** in the extended menu.

Click **Create Project**. Enter Project name then **Create**.

Oauth requires configurating consent screen as a prerequisite. Click **Configure consent screen** to proceed.

Tick the **External** User Type to make it available for any user with a Google account. **Create**.

Name the application.

In the Authorized domains, add **mongodb.com** (Figure 37):

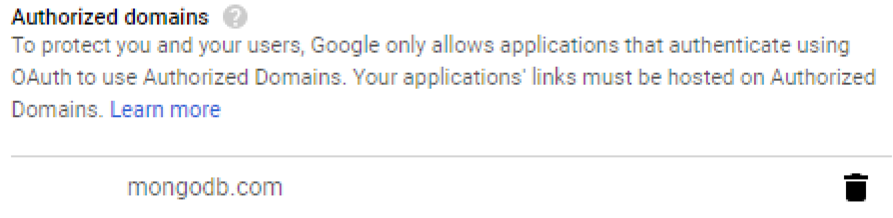

Figure 37. Authorized domain configuration view in Google Console.

Save. Go back to the Credentials page and Click **Create Credentials**. Choose **Oauth client ID**.

Choose Web application type. Name the OAuth client ID freely as it is not for displaying.

For Restrictions: URIs for Stitch application that is hosted in the eu-west-1 region (this could be modified later) (Figure 38).

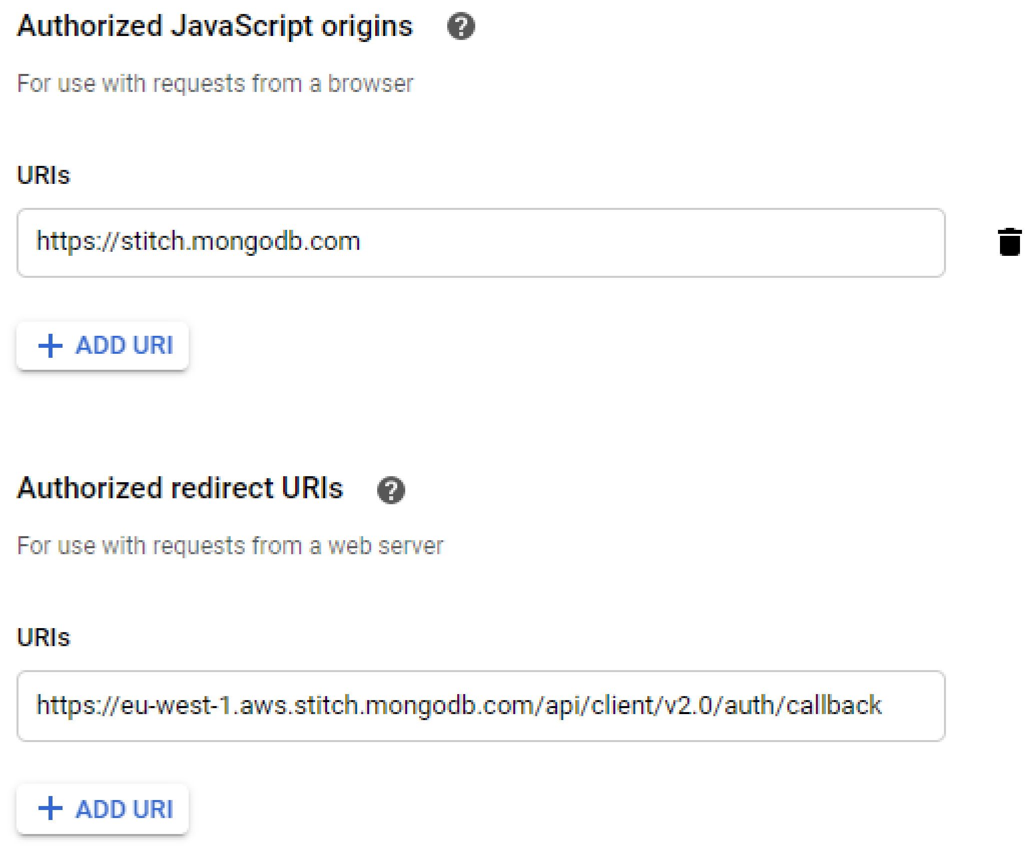

Figure 38. Authorized redirect URIs configuration.

The **Client ID** and **Client secret** are necessary for Stitch settings.

Log into Atlas and navigate to **Stitch** application.

Select **Values & Secrets** to privately store **Google Client secret.**

Name the secret and enter the Client secret as value.

Do the same step above to configure **Google Providers**.

Enable and enter **Client ID** from Google, choose a created secret in the dropdown.

Enter **Redirect URIs** as the client-side application will authenticate user (Figure 39):

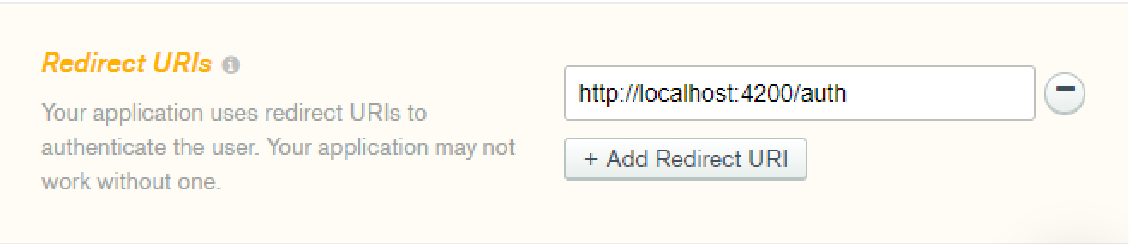

Figure 39. Setting view in MongoDB Stitch.

Select **Metadata fields** those will need approvement of user.

2. Implementation in code

To begin with Google authentication process. Call loginWithRedirect() function (Figure 40).

```
loginWithGG() {
    const credential = new GoogleRedirectCredential();
    Stitch.defaultAppClient.auth.loginWithRedirect(credential)
  }
```

Figure 40. Login to Google account function.

The application starts redirecting the user to the Google login page. Once users proceed with their credentials, Google will authenticate their identity and, if users are confirmed successfully, ask for permission to share data corresponding to Metadata fields. Google will then navigate to Stitch to save access token and the user is redirected back to the redirect URI as specified (**localhost:4200/auth**) to give users access to use application by calling (Figure 41):

```
/** Handle user if there is a redirect result from
    third-party authentication provider
*/
if (Stitch.defaultAppClient.auth.hasRedirectResult()) {
  Stitch.defaultAppClient.auth.handleRedirectResult().then(user => {
  // console.log(user);
  this.store.dispatch(new AuthActions.Signedin(user))
  });
}
```

Figure 41. Handle the redirect user data.

4.7 Deployment

Server-side app deployment (Stitch app)

Stitch app is automatically deployed within MongoDB Stitch server, as a server-side application. To watch the deployment history (Figure 42):

Go to Stitch page after successfully logged into Atlas.

Choose **Deploy** beneath the **Manage** section in the left-hand pane.

Choose the **History** tab.

Deploy

History Configuration Import/Export App

Only the last 100 deployments will be captured in deployment history. Automatic Deployment: OFF Refresh

Time of Deployment	Status	Change Origin	Actions
02/18/2020 12:47:34	Successful	Stitch UI	
02/18/2020 12:38:42	Successful	Stitch UI	
02/16/2020 12:57:23	Successful	Stitch UI	

Figure 42. Deployment history records view.

Host a Client side application into Stitch

1. Build application for production mode.

The current build of the project is in development mode. To make it ready for hosting, follow these steps:

Open terminal/command prompt and go to the project directory (Figure 43). Run:

```
ng build --prod
```

Figure 43. Command to build project into production bundle.

As defined in the angular.json file (Figure 44), the output directory of built version path is: **dist/Notes** based on the project directory with the root file is **index.html**.

```
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-angular:browser",
        "options": {
          "outputPath": "dist/Notes",
          "index": "src/index.html",
```

Figure 44. The settings of the projects.

2. Upload the built folder to Stitch.

Click **Hosting** under the **Manage** section in the left hand pane in Stitch page.

Choose **Upload Files** and select the folder **Notes** in the **/dist** path.

The process ends up with the uploaded directory is served at the URL (Figure 45):

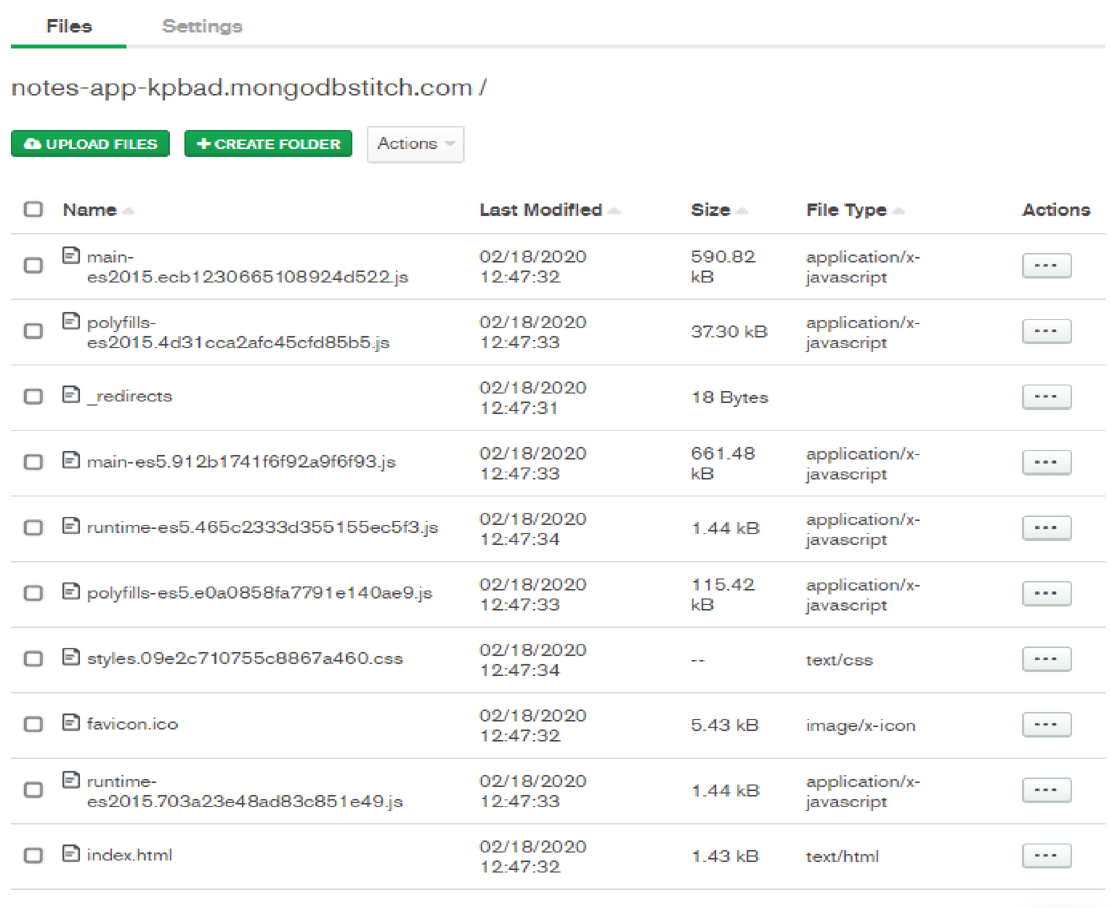

Figure 45. Upload built folder to the platform.

Most of the things are up and running, yet Angular or other modern frameworks use a client-side router as a behavior of a Single Page Application. This can return a 404 error page for any request other than the root file.

In this case, Stitch provides a configuration for handle all requests to the root page.

Go to the **Settings** tab in the **Hosting** page.

Select Choose file next to **Single-Page Application** and choose the root file **(/index.html**) (Figure 46).

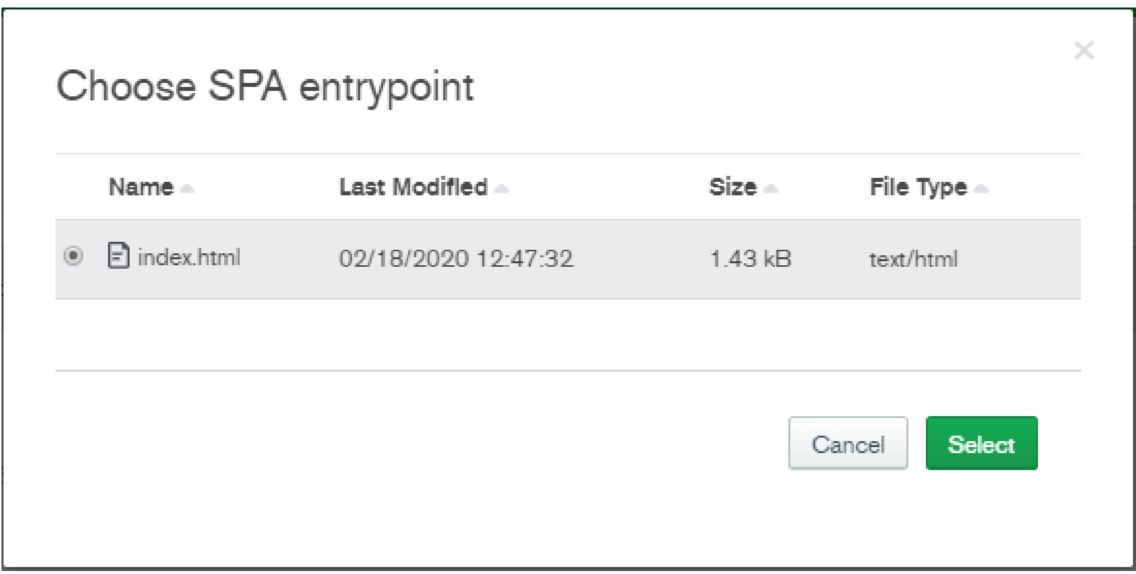

Figure 46. Define the redirect root file.

The project is served and works properly as in the development mode.

3. Re-configure the redirect URIs

Since the real URL of the project is quite different from in the development build (localhost). Hence, this could affect the request for third-party authentication services.

Navigate to **Users** on the **Stitch** page.

Select **Providers** and choose the one that the client-side application uses (**Google**).

Add the URL for authentication confirmation as a new **Redirect URI** (Figure 47). Then hit **Save**.

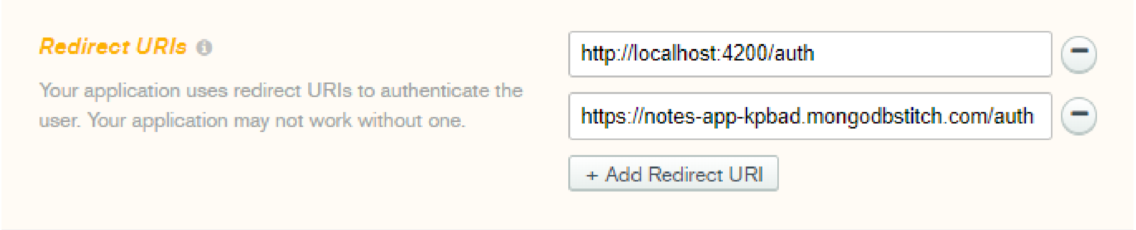

Figure 47. Add the new address in Redirect URIs settings.

It is done as the Google Authentication works perfectly now.

5 CONCLUSION

The purpose of this thesis was to provide an overview of web application development in general and specifically the potential of SPA. SPA has thoroughly been introduced throughout the thesis. The mechanism of how SPA works, as well as its advantages and disadvantages are also mentioned to give a better idea of why it gain more developer focuses.

Along with the uprising power of SPA, Cloud computing is becoming more and more popular due to not only the improvements it created but also its reliability in scale management of servers. Different types of cloud computing models are discussed in the thesis to give a comprehensive idea of each type.

This thesis also provides a good practical method of building a serverless application from scratch, which offers a good start for those who want to dive into Serverless applications.

REFERENCES

[1] A. Boten, "The evolution to serverless and where we stand today," 2018. [Online]. Available: https://medium.com/@codeboten/the-evolution-to-serverless-and-where-do-we-stand-today-bc9586990b18. [Accessed 2 2020].

[2] J. Clement, "Number of internet users worldwide from 2005 to 2019," 1 2020. [Online]. Available: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/. [Accessed 3 2020].

[3] Wikipedia, "Single-page Aplication," [Online]. Available: https://en.wikipedia.org/wiki/Single-page_application. [Accessed 2 2020].

[4] A. Groom, "What Is a Single-Page Application?," 18 7 2018. [Online]. Available: https://dzone.com/articles/what-is-a-single-page-application. [Accessed 2 2020].

[5] M. Wasson, "ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET," 08 06 2015. [Online]. Available: https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/november/asp-net-single-page-applications-build-modern-responsive-web-apps-with-asp-net. [Accessed 2 2020].

[6] Wikipedia, "WebSocket," 25 2 2020. [Online]. Available: https://en.wikipedia.org/wiki/WebSocket. [Accessed 2 2020].

[7] P. Sherman, "How Single-Page Applications Work," 11 4 2018. [Online]. Available: https://blog.pshrmn.com/how-single-page-applications-work/. [Accessed 2 2020].

[8] G. Elsayed, "WHY (SPA)? Let's dive into Single-page Application," 13 3 2019. [Online]. Available: https://medium.com/@ghadaalsayed2/why-spa-lets-dive-into-single-page-application-d0ea92be986c. [Accessed 2 2020].

[9] Angular, "Introduction to the Angular Docs," [Online]. Available: https://angular.io/docs. [Accessed 2 2020].

[10] C. Tuzi, "Understanding Angular modules (NgModule) and their scopes," 7 4 2017. [Online]. Available: https://medium.com/@cyrilletuzi/understanding-angular-modules-ngmodule-and-their-scopes-81e4ed6f7407. [Accessed 2 2020].

[11] R. Khalaf, "Angular life-cycle hooks," 20 6 2017. [Online]. Available: https://stackoverflow.com/questions/44648066/angular-life-cycle-hooks. [Accessed 2 2020].

[12] N. Lapinski, "The Three Pillars of Angular Routing," 4 9 2018. [Online]. Available: https://medium.com/angular-in-depth/the-three-pillars-of-angular-routing-angular-router-series-introduction-fb34e4e8758e. [Accessed 2 2020].

[13] R. Y, "Serverless on Google Cloud Platform: an Introduction with Serverless Store Demo," 24 1 2019. [Online]. Available: https://medium.com/google-cloud/serverless-on-google-cloud-platform-an-introduction-with-serverless-store-demo-41992dec085. [Accessed 2 2020].

[14] N. Jain, "Introduction to Cloud Computing – Growing Importance," 2 4 2018. [Online]. Available: https://www.whizlabs.com/blog/cloud-computing/. [Accessed 2 2020].

[15] J. Coffield, "Serverless blog," [Online]. Available: https://serverless.com/blog/strategies-implementing-user-authentication-serverless-applications/. [Accessed 3 2020].

[16] JWT, "Introduction to JSON Web Tokens," [Online]. Available: https://jwt.io/introduction/. [Accessed 3 2020].

[17] A. Abel, "Introduction to Angular component," 2 12 2017. [Online]. Available: https://medium.com/@agoiabeladeyemi/introduction-to-angular-component-138e9c24b54a. [Accessed 2 2020].

[18] B. Han, "An Introduction to Serverless and FaaS," 5 11 2017. [Online]. Available: https://medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2. [Accessed 2 2020].

[19] I. MongoDB, "Welcome to the MongoDB Docs," [Online]. Available: https://docs.mongodb.com. [Accessed 2 2020].

[20] A. Perera, "Introduction to Serverless," 12 10 2018. [Online]. Available: https://dzone.com/articles/introduction-to-serverless. [Accessed 2 2020].

[21] C. Woodford, "Cloud Computing," 11 5 2019. [Online]. Available: https://www.explainthatstuff.com/cloud-computing-introduction.html. [Accessed 2 2020].

[22] P. R, "Angular 2 Life cycle hooks," 25 4 2017. [Online]. Available: https://www.techjini.com/blog/angular2-lifecycle-hooks/. [Accessed 2 2020].