Givantk

Graduation Project Documentation

Ain Shams University

Computer Engineering and Software Systems Department
2019

Supervised by: **Prof. Ahmed Hassan**

Hint:

Givantk is an mobile application that is aiming to make life easier. It is based on crowdsourcing.

Givantk depends on the idea that people have different needs, and taking advantage of the fact that your mobile phone is your best friend in our modern world, we believe that our application represents a **breakthrough** in the Egyptian society.

We used the **IMRaD** format in our documentation, as a recommendation from Dr. Ahmed Hassan.
IMRaD is an acronym for Introduction – Method – Results – and – Discussion.

The **IMRaD** format is a way of structuring scientific articles. The language will be as plain and as unambiguous as possible. There is no place in this type of writing for personal views and fanciful language.

Participants:

- Muhammad Muhammad Salah Eldin
- Abdelrahman Mohamed Ahmed Mohamadeen
- Osama Mohamed Hafez
- Doaa Jaber Atteya

INDEX

Introduction

Brief Introduction

Givantk is not only a mobile application.
Here is the core concept:

For users, it will be a mobile application ("*Givantk* App"). For admins, it will be a web application (Control Panel).

There are two types of users, **user A** will post on the app a micro to medium-sized service that he wants to be done for him/her. (For example: Buying something from the convenience store, recharging his/her credit card, providing information about a specific topic, standing in the queue instead of him.. etc.)
The service can be as big as consulting for building a startup, and as small as buying something from the supermarket.
User B will pick a service, and do it for an amount of money paid by **user A** (paid through the application), or voluntarily.

The app's core depends on crowdsourcing, which is defined as:

"The practice of engaging a 'crowd' or group for a common goal;

often innovation, problem solving, or efficiency."

Detailed Description of the Provided Services

1- The user can reach other users who can help him/her in daily tasks, and these users are just normal people ..it will be freelancing for the ordinary tasks.

2- The user starts a new service request and writes its content, whether it's that he wants help in establishing his new startup, find someone to buy cheese from the supermarket, needs to recharge the mobile credit, he/she is lost on the road and needs some guidance or to know what mean of transportation to take, needs some information about governmental papers or issues from someone who has a previous experience, etc..

3- Services could be free or paid, and the user is the one who determines when the request will be initiated.

4- Free services makers will take points as an incentive for every free service they make ,and in the future maybe they can exchange it for discounts in certain stores or restaurants.

5- One service request may need more than one person to do it, the user chooses the number of people to satisfy his/her request and also chooses the amount of money he/she is willing to pay.

6- The user specifies the criteria needed for the service maker such as location, age, job, hobbies, or any keyword chosen by the user.

7- The user chooses the type of the service whether it’s transportation, purchasing, information, etc ..or he can choose none of them if his service type is not listed.

8- The request may vanish after a certain period time determined by the service initiator, if no one responds to it.

9- The app search for service makers that have the criteria requested based on what they wrote on their profile when they registered on the app.

10- The user can choose also whether the service will be published on the homepage for all other users or not.

11- The user can ask for a service anonymously if the service was about exchanging information only.

12- The app sends a notification to the service maker.

13- If another service maker accepts the notification/request first, its state will be changed to 'resolved'.

14- The service maker can accept or reject the request, if it’s paid he can also suggest certain amount of money, higher than the amount pre-determined by the service initiator.

15- If service maker wants more information or wants to negotiate, he can write a comment to the asker in the request, and the asker responds to him/her, without direct chat.

16- After service maker accepts the request, the one who asks for the requests has to confirm.

17- After accepting, the service maker and the user can contact each other through chat , phone or text messages, and if there is more than one service maker in the request, the user can make a group chat between all accepted members.

18- Every user must provide his/her identity card, verified phone number, also a profile photo identical to that of identity card, before asking for paid services, in order to ensure safety for all users.

19- To handle problems that may happen in paid services, we thought of some procedures to be taken:

a- To make a paid request, user have to charge his balance on the app first by visa, or another payment tool (to be determined), in order to fine him if he cancelled a request after someone accepted it.

b- The user who asks for a service, may ask for any guarantee -while he is initiating the request-in order for him to make sure that the service will be done correctly. These guarantees could be a receipt, a photo, a screenshot, or anything the user determines.

c- The user has to accept or reject the guarantee provided by the service maker, if he/she doesn’t for 2 days; the service maker can take his/her money.

d- If the user rejected the guarantee, they can discuss the problem through chat, and if there is still no resolution, any one of them can request for app resolution, and the app checks the guarantee provided.

20- The asker and the service maker will rate each other after the service is finished.

21- Every service maker below certain rating will be blocked in order to ensure services quality.

22- In the homepage, service makers can subscribe for certain type of services to see new request in their homepage.

23- The supports user references.

Problem Statement

In 2019 there will be around 28 million smartphone users in Egypt, and a total of 2.5 billion users around the world (36% of the population).

So 36% of the world population are having these amazing mobile devices that have a touchscreen interface, Internet access, and an operating system capable of running downloaded apps.

It's with no doubt the communication revolution at its climax, everyone of course is realizing that, and we can make sure of this by reflecting upon a simple fact that:

"An average of 6,140 mobile apps are released everyday through the google store, Apple App store reaches a number of 1,434 mobile apps released daily."

Although there is a large number of apps that is emerging every day,
A lot of them fail, and we need to ask ourselves a vital question before we proceed, which is: Why do they fail?

According to a study made in the International Journal of Software Engineering & Applications (IJSEA), Vol.5, No.5, September 2014
By Venkata N Inukollu, Divya D Keshamoni, Taeghyun Kang, and Manikanta Inukollu.

There are two main reasons for failing of lots of mobile apps:

I-Technical Problems:

According to the top negative reviews and statistics made by the authors of the above study, “44% verbally express they would expunge a mobile app immediately if app did not perform as expected”. The numbers clearly point out that there are good apps, and lamentable apps in the app market.

App users not only uninstall the app, but also provide negative reviews on the app when customers do not relish the app.

With social media and word of mouth being so popular negative reviews spread rapidly, which rigorously affects the reputation of developers and poses a threat to their future releases.

Users own a significant role in the success/failure of an app. A Mobile App review survey was conducted on a sample of over 500 American mobile app users, aged 18 years or older.

According to the results of the survey “96% of the American mobile app users say there are frustrations that would lead them to give an app a bad review”, including the following:

• Application/system freezes – 76%

• Application/system Crashes – 71%

• Slow responsiveness – 59%

• High battery consumption – 55%

• Considerable amount of ads and promotions – 53%

The survey also has recorded the statistics of the number of users for whom performance matters the most. Without any doubt, the number is 98%, i.e. almost every app user considers performance as his/her main priority.

When the users were questioned about the type of apps, for which the performance mattered the most:

(74%) said banking apps

Followed by maps (63%)

Mobile payments (55%)

Mobile shopping (49%)

Games (44%)

Social media (41%).

Interesting statistics have also been recorded regarding the consequences of poor performance of apps:

44% of the users would delete the app immediately.

38% would delete the app if it would freeze for longer than 30 seconds.

32% would use a negative word of mouth to inform about the bad performance.

21% would post their negative comments on social media (such as facebook, twitter, and blogs.

18% of the sample responded by saying that their patience time is only about 5 seconds, i.e. they would delete the app if it froze even just for 5 seconds and this number is expected to increase drastically in the future. Even though the number is small.

27% of people said they would keep a paid app a little longer in spite of its poor performance, but the damage is already done which is not easily repairable. The users would not prefer to buy any such apps in the future from that particular developer/brand.

II-Marketing Problems:

According to the study With so many developers building new apps each day, it is becoming extremely difficult to acquire/attract, retain and monetize customers and to develop brand equity and loyalty.
Developers and apps are similar to brands who need to be marketed. App marketing plays a vital role in the success/failure of a mobile app. Marketing and social efforts are required to keep consumers engaged after the app is downloaded to their device.

Insufficient marketing efforts and marketing strategies will lead to decline in return on investments and hence will result in disappointments and frustrations. 70% of developers are frustrated with the current state of app marketing

So are they only the technical and marketing flaws the problems that we have to solve to develop a successful app?

We think that the answer is no, we think that a successful app needs to be built on an unsatisfied need. This is our core question, which is to find this unsatisfied need.

Even though these astonishing numbers of emerging apps are offering new solutions to everyday problems, and lots of them are mainly about communication, but unfortunately there still are lots of communication needs that aren't satisfied.

If we think carefully we will find that the problem of communication at its heart isn't completely solved yet.

The problem of communication in its core is not just about reaching others, and it's not just about the number of those others, but rather it's about Who will you communicate with to satisfy your needs and how will you communicate? It's not about the quantity but the quality. We don't need random communication, we need the most suitable and beneficial communication.

So let's have some **examples of real life situations** in which we we want to communicate efficiently with others to satisfy a need or service:

1-Ahmed is a visually impaired man, who can’t see well without the aid of others, he wants to communicate with a sighted person to read something for him.

2-khaled is reluctant to choose between different departments in his college, he wants to easily contact students in each department to help him taking a decision.

3-Mohamed is going to get a driving license , but he is too afraid of taking the test. He wants to chat with someone who experienced it before to tell him about the test circumstances, and all the procedures to be taken.

4-Abdelrahman wants to launch his own startup, but he can’t launch it alone as he needs another partners, how could he find trustworthy partners with the right qualifications for the startup, and initiate discussion with them?

5-Doaa makes homemade food and sells it for others online, but she faces a problem of finding someone with a suitable price who can deliver her food to the customers.

6-Mariam is going to get married, she wants someone to move boxes of her belongings from her current home to her future home via a truck. She can’t find one easily.

7-Mona wants to buy a ticket to attend her favorite band concert but she is very busy and transportation is difficult, she just wants to find someone who can buy it for her and receive it from him/her later.

8-Hussein wants to know what transportation to be taken to a certain destination, he googled it but he found outdated information. He wants to quickly contact anyone in this destination to help him figuring the most suitable transportation.

9-Farid wants to rent the football field in his college, he doesn't want to attend college this day because he and his friends are in vacancy,so he needs to find another student in the college to rent the field for him.

10-Gamal is terribly shy, he can't make any friends. He wonders if there is any way that he can find friends who share his hobbies and enjoy their time together.

In all of the examples above in order to find solutions, we find that Timing is very critical, matching people with specific criteria is very critical.

So the problem is mainly about having a need or request that we want to satisfy but we can’t find the right people in the right place in the right time, in order to satisfy those needs.

We can sum the problem up in the following words:

Our problem is: “people matching according to service’s type and specific criteria chosen by service requester, plus providing efficient ways of communication after that ”.

And we mean by matching here not one to one matching, but how to recommend services to the right people in the right time.

And the question is: can technology help us?

.

Literature Review

Now let’s dig a little bit deeper and explore the available solutions of this problem. The approach that we will take is simply to discuss the available solutions in web and mobile app technologies to each of the ten examples which are discussed previously in the “problem statement” section .

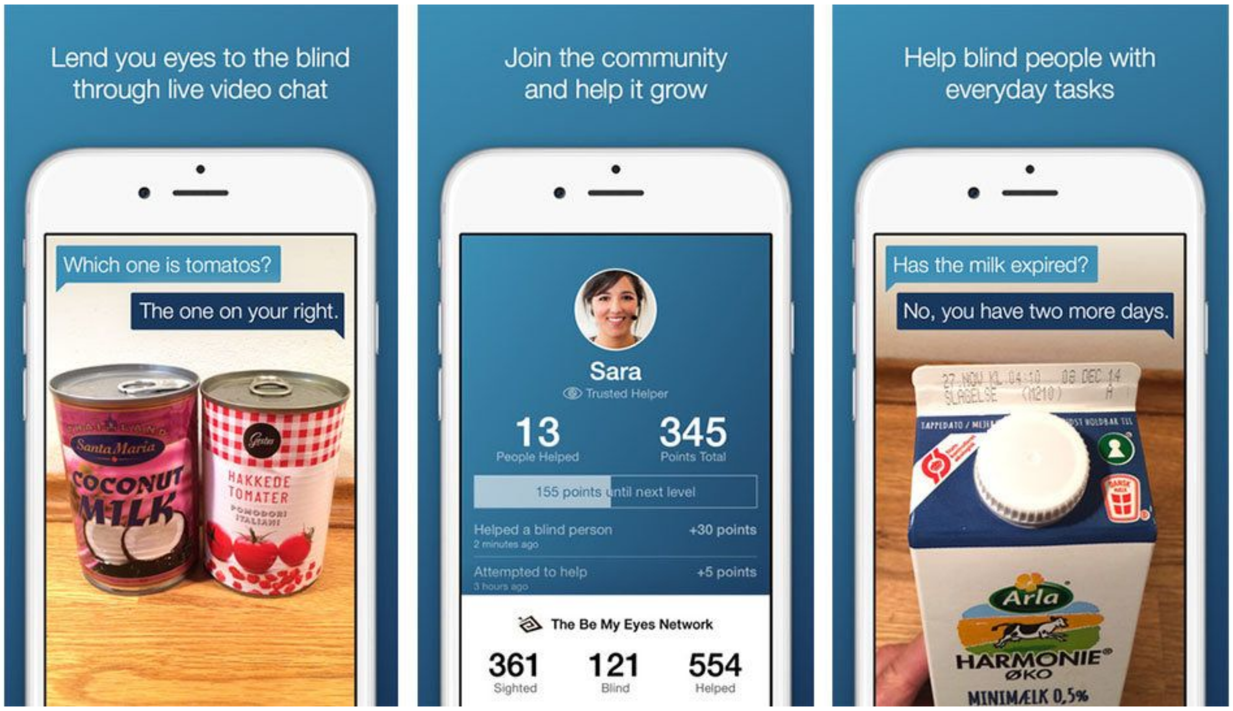
To show you what we mean let’s begin with the first example:

Example one:

“Ahmed is a visually impaired man, who can’t see well without the aid of others, he wants to communicate with a sighted person to read something for him.”

Current Solutions:

After doing some research about apps that helps visually impaired or blinded people, we found an app called “Be My Eyes”.



Lend you eyes to the blind through live video chat

Which one is tomatos?

The one on your right.

Join the community and help it grow

Sara

Trusted Helper

13	345
People Helped	Points Total

155 points until next level

Helped a blind person +30 points

2 minutes ago

Attempted to help +5 points

3 hours ago

The Be My Eyes Network

361	121	554
Sighted	Blind	Helped

Help blind people with everyday tasks

Has the milk expired?

No, you have two more days.

We noticed seven main things:

1-It’s a cross platform app, which means it runs on (android and ios).

2-The main idea of the app is to connect volunteers with visually impaired people, in order to help them.

3-It’s very accessible to the users, they can use microphone and camera to make voice or video calls to the volunteers. The action buttons themselves are large in width and font size.

4-The matching technique that connects users to volunteers is “the language”.The app asks for your first language before you can get your request.

5-It took 38 seconds to reach the first English volunteer while experimenting the app. When we changed the language into “Arabic” it took around the same time, which is a very good timing indeed.

6-The app was released on oct 4,2017 which is about a year from the date we are writing this chapter on, the number of volunteers reached 1,809,689 volunteer, and the number of Blind & low-vision is 105,760.The ratio of volunteers to visually impaired people is 17 times, which is an amazing number relative to how recent the app is.

7-There is no focus on attractive design, but mainly on the functionality and speed.

Examples: two and three:

"khaled is reluctant to choose between different departments in his college, he wants to easily contact students in each department to help him taking a decision.”

“Mohamed is going to get a driving license , but he is too afraid of taking the test. He wants to chat with someone who experienced it before to tell him about the test circumstances, and all the procedures to be taken.”

Current Solutions:

The most popular way to connect with experienced people in a specific domain is using facebook groups.

By analyzing how things are going on these groups, we noticed the following:

1-People are matched together based on their “interest in the same topic” .

2-Requests or opinions are written in posts, that are seen to the other members, which might be annoying to someone who likes privacy.

3-Questions in knowledge exchange groups may take up to one day or more without any response. Only you receive notifications from members who follow your question.

4-Connection through such groups is mainly through text, which is not very reliable, fast, nor easy.

5-Users can take advantages of old questions to find answers to their request.

6-It’s not easy to find some groups about specific topics if you aren’t already a member.

7-There is no reputation system or incentives for the users to reply.

There are many other alternative platforms that help you connecting with experienced users, but the realized thing is that they are too technically oriented, and communication is also mainly through text.

Example four:

“Abdelrahman wants to launch his own startup, but he can’t launch it alone as he needs another partners, how could he find trustworthy partners with the right qualifications for the startup, and initiate discussion with them?”

Current Solutions: “Ideas voice” is a website that helps you to find a cofounder.

cup of ideasvoice
Find your Partner, Empower your Project
Meet | Pitch | Discuss | Co-create

We noticed the following :

1-Limited number of users.

2-The site has an optional verification system by checking the identification card, so verification is not guaranteed all the time.

3-You have to pay 25 dollars to get your identity verified.

4-Matching is done mainly by location.

5-To contact a certain cofounder, you have to fill your needs and write about your project, and we see that as unnecessary work.

6-No reputation system is founded.

Examples five and six:

“Doaa makes homemade food and sells it for others online, but she faces a problem of finding someone with a suitable price who can deliver her food to the customers.”

“Mariam is going to get married, she wants someone to move boxes of her belongings from her current home to her future home via a truck. She can’t find one easily.”

Current Solutions:

“Taskrabbit” is a popular website and application that defines itself as “The convenient & affordable way to get things done around the home”.

TaskRabbit

We noticed the following:

1-It generally deals with handy jobs.

2-It divides tasks into categories such as:

- Mounting and installations.
- Moving and packing.

- Furniture assembly.
- Heavy lifting.
- General Handyman.

3-It divides users into normal users and taskers.

4-People are matched together based on “task categories” (Point number 2).

5-The app is accessible, and many request are done successfully.

5-There are complaints that taskers have to pay money (20 dollars) to register.

6-There are complaints about customer services, and not handling accountability for taskers mistakes.

7-There are complaints about taskers cancellation and not showing up.

Example seven:

"Mona wants to buy a ticket to attend her favorite band concert but she is very busy and transportation is difficult, she just wants to find someone who can buy it for her and receive it from him/her later.”

Current Solutions:

There are some websites like “stubhub”, “seetickets”, “Twickets”.They define themselves as a fan to fan market, but in all of them we noticed the following:

1-They have very limited users.

2-Their coverage is mainly in US and UK .

3-Searching for an artist, event, or team is the main way to match you with ticket sellers.

4-In twickets website a 12% fee is taken on each bought ticket.

5-Some sites provides delivery of tickets , while others depend on the fans to handle delivery options.

6-In some of these websites there is a guarantee to cover for the full amount of tickets if they aren’t as described.

Example eight:

“Hussein wants to know what transportation to be taken to a certain destination, he googled it but he found outdated information. He wants to quickly contact anyone in this destination to help him figuring the most suitable transportation.”

Current Solutions: No specific solution is founded that involves human interaction.

Facebook groups might be a solution.

Example nine:

“Farid wants to rent the football field in his college, he doesn’t want to attend college this day because he and his friends are in vacancy,so he needs to find another student in the college to rent the field for him.”

Current Solutions: No specific solution is founded.

Posting in user's profile on Facebook or using facebook groups might be a solution.

Example ten:

"Gamal is terribly shy, he can't make any friends. He wonders if there is any way that he can find friends who share his hobbies and enjoy their time together."

Current Solutions:

A solution for this problem is an app called "patook", which defines itself as "strictly platonic". It doesn't allow approaching other users for romantic reasons, it's all about making friends.

We noticed the following:

1-It has automated computer algorithms that detect unwanted contributions (e.g: flirt detector, spam detector,etc..).There is very minimal human moderation or support.

2-The app matches users according to information that they have to fill in their profile like:

- Martial status.
- Offspring.
- Orientation.
- Ethnicity.
- Religion.
- Politics.
- Smokes.
- Drinks.
- Education.
- Diet.
- Personality type.
- Shyness.
- Astrological Sign.
- Country of origin.
- Languages.

It also supports adding keywords

3-The reputation system depends on points based on many things such as :

- Logging recently.
- Having a photo.
- Filling their profiles.
- Quick response.

4-There is a questionnaire that users can take. This allows Patook to assign “points” for user traits. Example, if you are looking for people who list museums and music on their page, they get more points and appear higher in your matches. This feature might feel overwhelming and distracting from user experience.

5-Complaints about customer support being unresponsive.

6-Complaints that detection algorithms are not very accurate.

Now let’s discuss in the next section: what’s unsolvable yet?, and how can we modify the flaws in these solutions in order to reach a whole new solution to provide better and faster communication?

What’s unsolved yet?

For all the previous solutions, we realized that they are different specific solutions to a general problem, which is as stated before: “people matching according to service’s type and specific criteria chosen by service requester, and providing efficient ways of communication after that”.

The approach that is taken by these apps depends on: “crowd sourcing” which is simply: “The practice of engaging a ‘crowd’ or group for a common goal”.

Let’s discuss the flaws of having many solutions to the same problem:

For every different service, users have to search for an app (and may or may not find one), register in it, prove their identity, fill their profile, so lots of work has to be done repeatedly.

The emerging of a group of apps with limited number of users in each of them, and subsequently limited options for matching.

Some of the needs for services are occasional needs, so users are not online or active frequently.

All of the solutions lack one or more vital feature like: incentives system, reputation system, identity verification system, privacy, support, reliable communication, and secure payment options.

So what keeps us from implementing one solution to this problem? why do we have so many apps to handle different domains of the same problem?

We need to provide one solution to this problem, as there is nor real need for that many solutions, but this solution should have certain features in order to make it successful

And we will discuss these features in the following points:

1-Recommending users:

Our solution is to use a general matching technique that depends on service type and description. The technique consists of the following steps:

Registered users have to complete a questionnaire with attractive user interface and points earned on every answer first before they can start using the app.

The questionnaire will collect information about users in every field of their life, starting with main information like (their location, languages they know, their job, study, skills) until it reaches (their favorite music, places brands,   and subjects). All these information will be saved to the database.

A machine learning algorithm will collect keywords from service description and compare these keywords with information stored in our database, to see relevancy between data stored and these keywords.

User himself can assign values to different criteria that he can choose from in (posting service screen).

The app sends notifications and recommendations to matched users.

2-The using of incentives: Incentives, which are defined as “positive motivational influences”, are very important for encouraging users to respond for different requests.

To use incentives properly, we thought of two kinds of incentives:

Users can make free services in exchange of points, these points can be exchanged later into discounts or free offers.

Users can make paid services for real money.

3- Reputation system:

The presence of rating for every user will keep the quality of their services and engagement. This will make it easy to ban users of low quality, or warn other users before dealing with them.

4-Privacy system:

Users can request some services or ask for information without revealing their identity to others .

Security procedures will be maintained to prevent leaking of any of the users information to ensure privacy.

5- Spam detection:

A machine learning algorithm will be used to detect spam, and fake users.

6- Handling cancellation:

we recommend the following procedures to handle cancellation:

for service makers, if they cancel or don't make the requested service after a certain time:
we will have second in turn service makers, who are ready to take place for the canceled ones.
If the first option is not available a compensation will be offered to the requester.
For service requester who cancel the request after a certain time:
A fee will be taken from him.
Points will be taken from him.
He will be banned if he repeated it again.

7-verification system:

Verification of identity will be a must for services that require real world interaction.
Verification will be through providing id documents, along with a photo for the user holding it.

9-Reliable communication:
The system should have all the components that provide reliable communication such as :
chat (one to one or many).
Voice calls.
Link attachments.

10-Ease of access:
User interface should be intuitive for users, and requesting a service should be very easy, for an instance users can request the service via voice not just written services.

11-Setting of Guarantees:

Requester may set any guarantee (such as receipts) he wants to assure that the service will be satisfied.
The presence of these guarantees will help the user to trust the app and take courage to request a service.

All of these features provide healthy environment for crowd sourcing and good communication between people to satisfy their requests.

Our application will work on every point to ensure that it will be achieved, and you will see the techniques that we use to implement these features in details if you continue to read the documentation.

Method

Overview

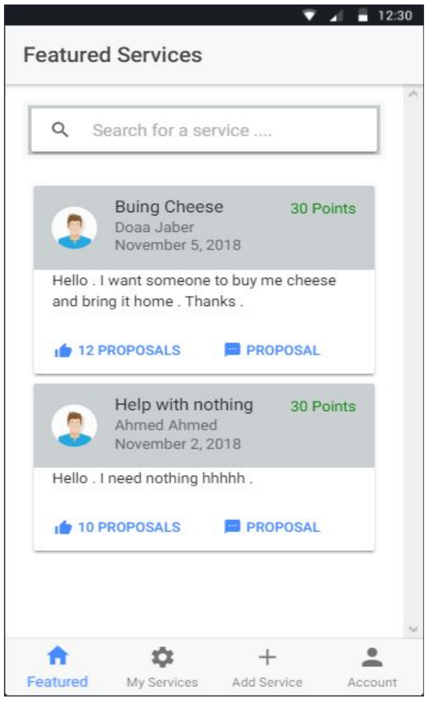
We used **React Native** to build the mobile app, **React JS** to build the admin website, and **Node JS** to build and manage the backend.

Main higher level tools used:

- **React Native**
- **React**
- **Node JS**
- **Express JS**
- **MongoDB**

Application Design

We have initially decided that our app will have 4 main tabs, ‘Featured’, ‘My Services’, ‘Add Service’, and ‘Account’.


Featured Services

Search for a service ....

Buing Cheese 30 Points
Doaa Jaber
November 5, 2018

Hello . I want someone to buy me cheese and bring it home . Thanks .

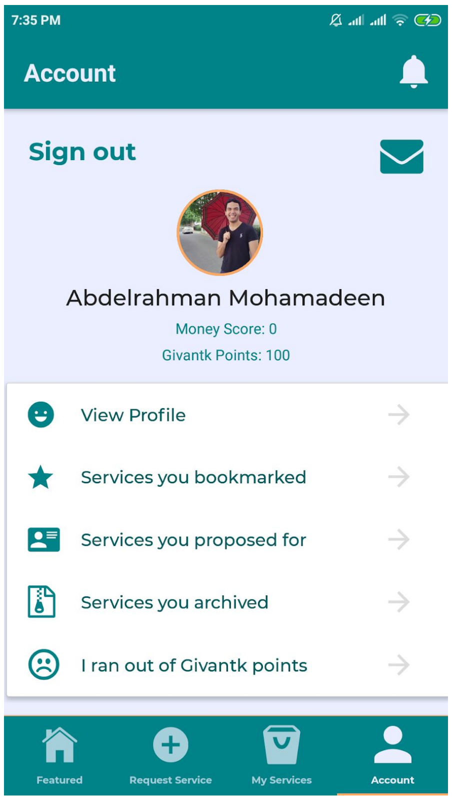
12 PROPOSALS PROPOSAL

Help with nothing 30 Points
Ahmed Ahmed
November 2, 2018

Hello . I need nothing hhhhh .

10 PROPOSALS PROPOSAL

And here is the design at the end:


Account

Sign out

Abdelrahman Mohamadeen

Money Score: 0

Givantk Points: 100

View Profile

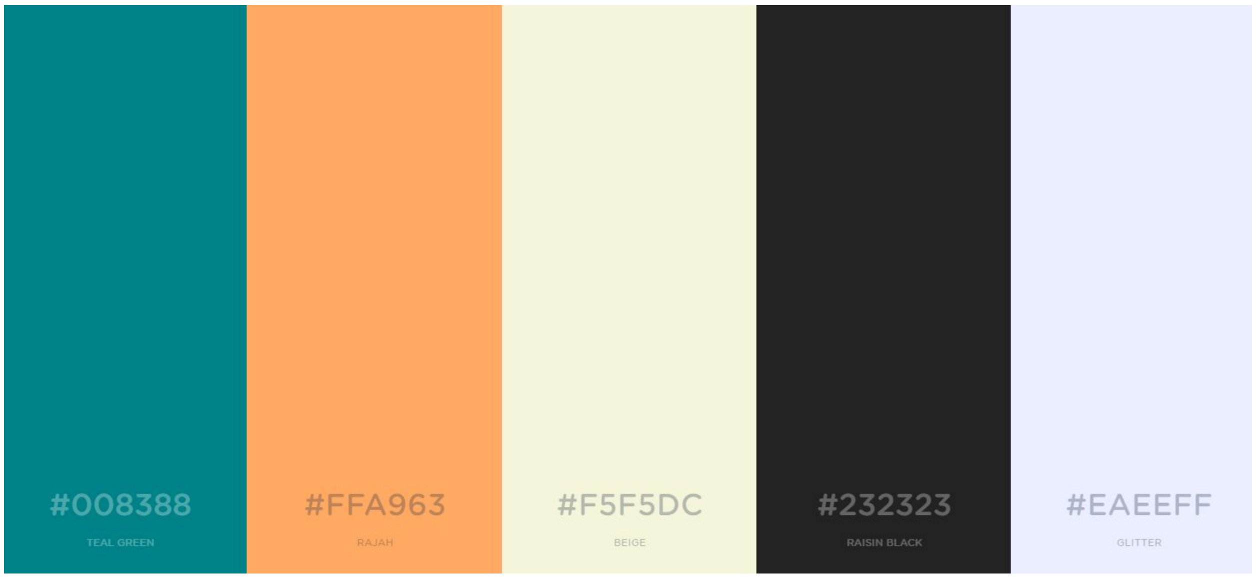
Services you bookmarked

Services you proposed for

Services you archived

I ran out of Givantk points

Here is the colors pallette for our app:



Application Front End

Files Architecture

1) Assets
1.1) Constants
Contains all logos needed for the app
1.2) Data
Contains all the data about the countries, currencies, Cairo districts, services natures and service types and testing notifications.
1.3) Styles
Contains the base styling of the app and all the types of fonts used.
1.4) Utils
Contains the http services and how to get User Image.

2) Components
2.1) Main Screen Components
2.1.1) Notification Card
Used to notify a user if someone replied to his proposal.
2.1.2) Notification Indicator
notification that contains the badges to show how many notification arrived.
2.2) Account Screen Components
2.2.1) Account List Items

List all the items in each account.

2.3) Account Inner Screen

2.3.1) Messages List Item

A component to list all the messages received per user.

2.4) Commons

2.4.1) Chat components

contains everything related to the chat app like the chat input text, the send button and the messages sent in the screen.

2.4.2) No Profile Disclaimer

A warning message sent to the user to inform him that the profile data is not complete.

2.4.3) Payment Related Components

Contains the amount, Allow Remember Me,Currency, Description, ImageUrl , Pre populated Email,
Style, on Payment Success method and on Close method.

2.4.4) Rating Related Components

Contains the rating card that contains the 5 stars and its styles and also the rating list which contains all the users that rated a certain user and the accumulation of the ratings.

2.4.5) Service Related Components

Same thing as Rating Related Components but for services only.

2.4.6) UI

Contains Announcement, Avoid keyboard Card list, content list, default button, default date picker, default text input, loading, main button, picker, quick modal, quick notification, snake navigator, text input Styling

2.5) Signup Screen Components
Contains every component related to the sign up process like sign up inputs and their header.

3) Routes
3.1) Bottom tab navigating
Contains all navigation for the main screens only which are Featured Screen, Add Service Screen, My Services Screen and Account Screen.
3.2) Main Navigator
Contains navigation for the rest of the app.

4) Store
4.1) Actions
all the redux (Description found in the packages sections) actions related to authentication, chat, givantk points, payment, profile and services.

4.2) Reducers
all the redux (Description found in the packages sections) reducers related to authentication, chat, givantk points, payment, error, profile and services.

Screens Architecture

1) Main Screens
1.1) Featured Screen
It is basically the home page of the app which and it contains all the services and basic information about the asker like the name and rating for the service as well as a search bar to search for specific services or users.
1.2) Add Service Screen
It is a screen that is used to add new service, the user is asked about the name, description, the service type, nature, select if it's paid or free, amount of points you want to give, amount of givantk points, amount of money you want to pay and select an optional picture.
1.3) My Services Screen
A screen that lists all the services done by a user as well as the services that is made by that user in the form of two separate lists to distinguish between them.
1.4) Notifications Screen
A screen that is responsible for delivering all the notifications for the users whether it is related to a user accepting a service another user required or a user got a notification that another user will do his/her service, we get notifications also if a user rated a service using our five-star rating service.
1.5) Account Screen

A screen that contains all the personal information about a certain user, it is the home of the notification badges and contains the chat system where the user has all the chats open with the users that he has business with weather it is a doer giver relationship or the opposite.

2) Featured Inner Screen
2.1) Searched Results Screen
It is a screen where you that is directed from the main featured screen after a text is entered into the search bar. the screen contain the services found according to the keywords entered into the search bar that is found in the main featured screen.

3) Notifications Inner Screen
3.1) Announcement Screen
It is screen that has all the announcement made by the administrators to inform the users for example of new updates or a new rule added to the app or any kind of warning or reward to a specific user.

4) Account Inner Screen
4.1) Archived Service Screen
When a user finishes a certain service or a user made the proposal of the service then this service is archived and is removed from the main featured screen to allow the users to

search for other service and not to submit other proposal to the same service again after someone already did.

4.2) Bookmarked Service Screen
It is a screen that contains all the bookmarked services which means that if a user wants to save a specific service so that he could present a proposal to it later or if he simply like this service or wants to save it to refer it to a user that he thinks is capable of doing this service.
4.3) Charge Money Score Screen
It is a screen that is related to the payment method that asks the questions of how the service will be paid, cash or with vodafone cash or with visa or even with goods exchange.
4.4) Givantk Points Screen
It is a screen that displays the givantk points which is a method to reward our users for doing a great job with the services that they chose to perform, points can also be given between users, and each user is given points for just signing up for the app. Points then changed with money to buy items or to use it as the payment method for a certain service.
4.5) Invite Friends Screen
It is a screen used to invite friends to the app, since we apply a "sign in using Facebook" method we can invite friends from Facebook to join our app and reward the user who do that.
4.6) Make Profile Screen

It is a screen where the user is allowed to fill up his/her profile with all the information needed to allow him/her to start working on the app, these information is like the name, age, email, profile picture though it is optional, personal preferences, bio and what type of services does he/she like.

4.7) Messages Chat Screen

It is a screen that contains everything related to the chat system where the users could make their deals privately concerning a specific service and this chat will be monitored by the administrators of the app.

4.8) Messages List Screen

It is a screen what holds all the chat messages history made by a certain user and it is ordered according to the last conversation made, it can be found in the account screen.

4.9) Personal Info Screen

It is a screen that contains the personal data of a user after he/she finished writing all of his/her information during the sign up process. Any user can view this information because they need to know who they are dealing with and to clear any misconceptions.

4.10) Profile Screen

It is a screen that holds all the personal information, messages list and the givantk points.

4.11) Proposed For Services Screen

It is a screen that tells the users what services they have proposed to since to can be a bit difficult to track every service that a user has applied to and without this information many deals may go wrong.

4.12) Verify Identity Screen

It is a screen that checks the identity of a user and to prevent any fraud actions made by users, like making more than one account or not showing the real name or real profile picture.

5) Registration Screen

5.1) Login Screen

It is the first screen the user sees and he/she can login using a previously made account during the sign up process or sign in using facebook.

5.2) Signup Screen

The signup screen requires minimal information just to start using the app like the username and the email, other information will be required if the user start to post any service or present a proposal.

6) Common

6.1) Chat Screen

It is a feature that is embedded inside the service card where the user who wants to present a proposal can enter a chat with the owner of the service and it is a fast way to take business instead of sending messages.

6.2) Loading Screen

It is a simple screen that show a loading spinner to inform the user that wait because we are loading data from the database or leading a certain screen from the server.

6.3) Service Screen
It is the screen that is presented when a user tabs on a certain service in the main featured screen, this screen shows every detail about the service like its type and the method of payment and the owner with every information about him/her.

React Navigation

First :The main components we used while using React Navigation library:

- A navigator is a component that implements a navigation pattern (eg: tabs)
- Each route must have a name and a screen component.
  - The name is usually unique across the app.
  - The screen component is a React component that is rendered when the route is active.
  - The screen component can also be another navigator.
- Each navigator must have one or more routes.
  - A navigator is a parent of a route.
  - A route is a child of a navigator.
- In our app we used two kinds of navigators:
  - Stack Navigator:Screens are stacked on top of each other.
  - Tab Navigator: User can switch between different tabs.

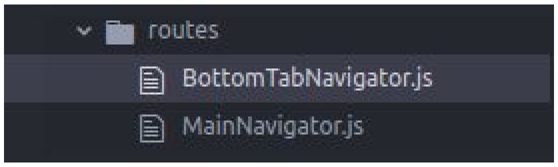
Second : installing the library :

```
npm install --save react-navigation
```

We wrote the following command in the terminal in the same directory of the project in order to install the library.

Fourth : using the library :

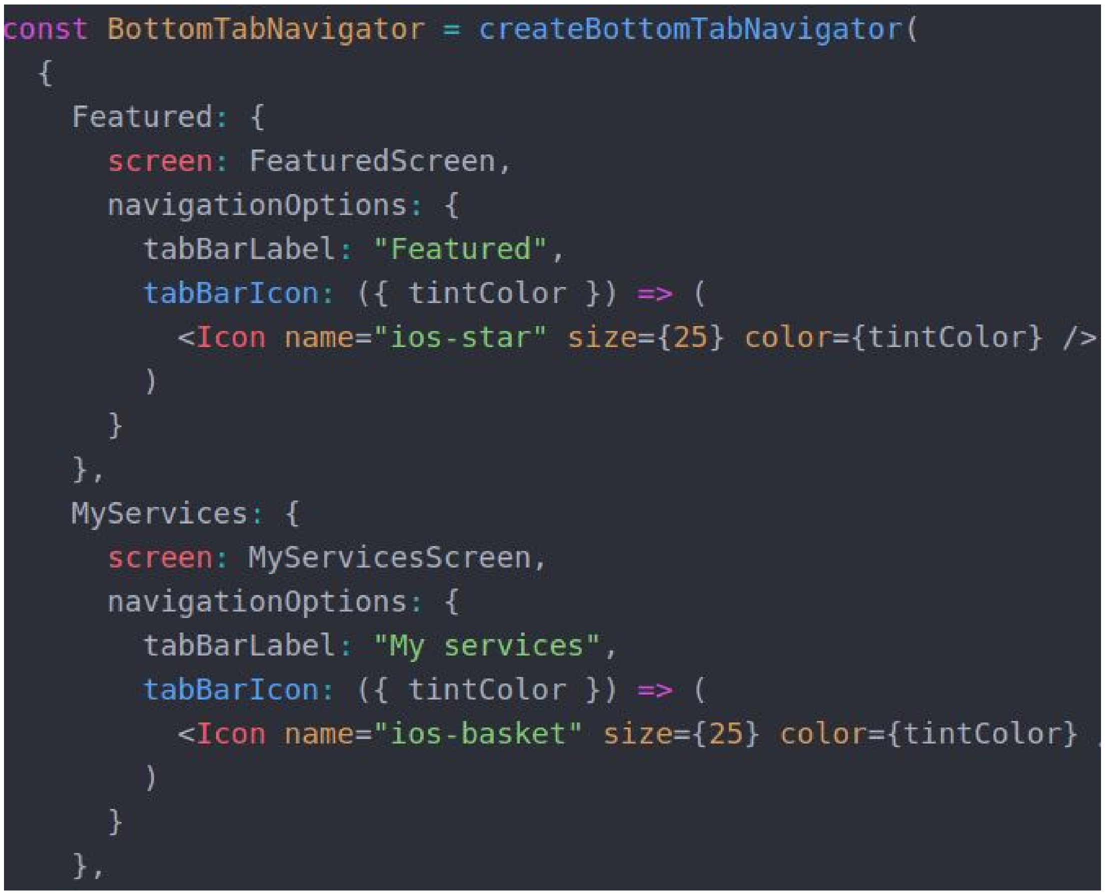
- We created a folder called ‘routes’ in the src folder.



- It consists of two files:
    - **BottomTabNavigator.js:**
        - We imported the createBottomTabNavigator function .
        - Inside this function we created 5 routes :
            - Featured.
            - MyServices.
            - NewService.
            - Notifications.
            - Account.
        - Inside the **naviagationOptions** object we assigned each of these routes to a certain screen component (*the screen components are imported from a folder we made called screens and we put*

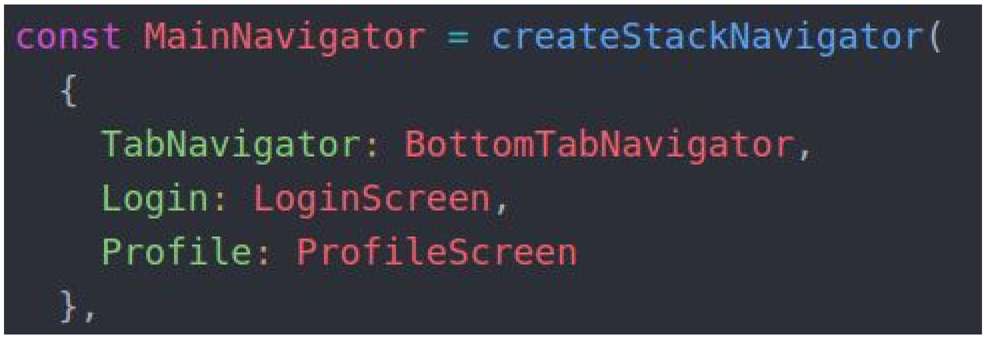
our screen components inside it) and we also determined icon & label for each of them to be shown in the Tab Navigator

- The following image shows a code snippet of what's described

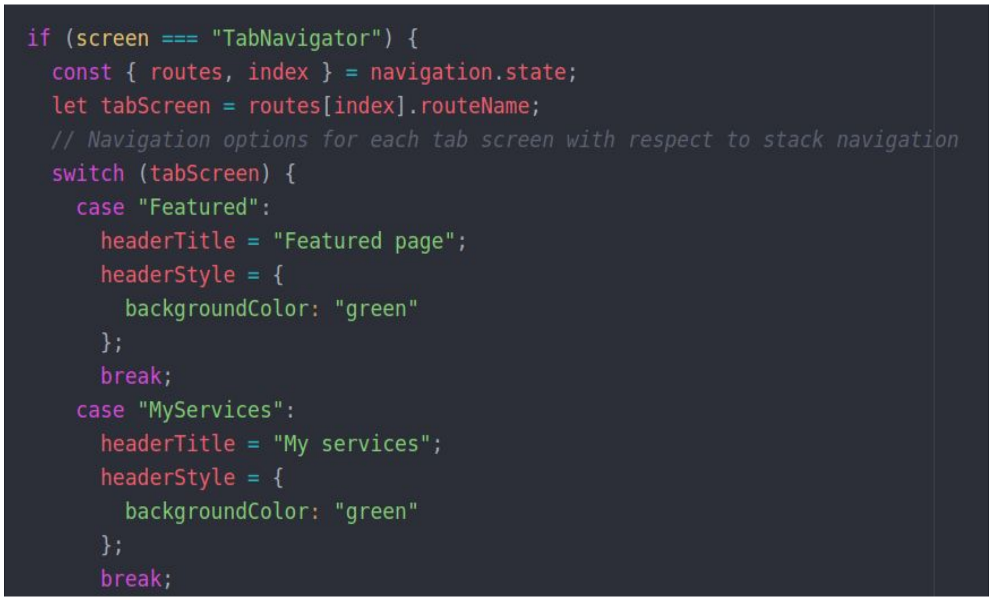


```
const BottomTabNavigator = createBottomTabNavigator(
  {
    Featured: {
      screen: FeaturedScreen,
      navigationOptions: {
        tabBarLabel: "Featured",
        tabBarIcon: ({ tintColor }) => (
          <Icon name="ios-star" size={25} color={tintColor} />
        )
      }
    },
    MyServices: {
      screen: MyServicesScreen,
      navigationOptions: {
        tabBarLabel: "My services",
        tabBarIcon: ({ tintColor }) => (
          <Icon name="ios-basket" size={25} color={tintColor}
        )
      }
    },
```

- **MainNavigator.js:**
  - The main navigator is a stack navigator which contains the following : the tab navigator, and any other screen that's supposed to be outside the tab navigator or just an inner screen of a screen that already exists in the tab navigator .

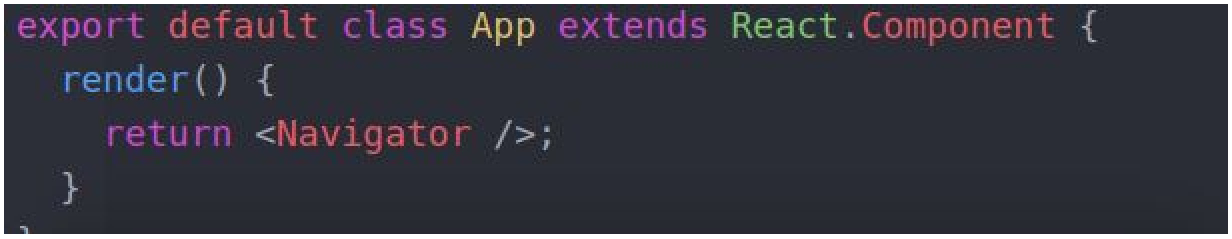
- ■ For now we have the following routes in the main navigator as shown in the image below :
  - 

```
const MainNavigator = createStackNavigator(
  {
    TabNavigator: BottomTabNavigator,
    Login: LoginScreen,
    Profile: ProfileScreen
  },
```

- ■ Then depending on which screen the user is in..we change the title and styling of the stack navigation header ..example of the code used is below :
  - 

```
if (screen === "TabNavigator") {
  const { routes, index } = navigation.state;
  let tabScreen = routes[index].routeName;
  // Navigation options for each tab screen with respect to stack navigation
  switch (tabScreen) {
    case "Featured":
      headerTitle = "Featured page";
      headerStyle = {
        backgroundColor: "green"
      };
      break;
    case "MyServices":
      headerTitle = "My services";
      headerStyle = {
        backgroundColor: "green"
      };
      break;
```

- Finally in app.js we imported the main navigator and rendered it:

- 

```
export default class App extends React.Component {
  render() {
    return <Navigator />;
  }
```

Application Backend

Overview

The backend part of the project is currently handling the registered users and the services that they provide or accept, we have implemented some admin functionalities like getting all the users registered on our app, get details about a specific user, update user or create new user and the same functionality also goes with the users' services.

The backend structure consists of four parts:

1) Server.js
Which is the entry point to the backend and contains:
 1. Initialization of express framework.
 2. Connection to MongoDB which is a NOSQL database.
3. Routes for the users API and the service API.
4. Some dependencies that help facilitates the backend functionality.

2) Models
Models handle the interaction process with the database. The basic models are the User model and Service model.

3) Controllers
Controllers are working with the data given from or to the model
1.For the users controller we implemented
getAllUsers()
getUser()
createUser()
updateUser()
2.For the service controller we implemented
getAllServices()
getService()
createService()
updateService()

4) Routes
Routes connect controllers, models and requests together.
1.Users routes Connect user model with its controller.
- get all users using GET request => router.get('/', UsersController.getAllUsers);
- create user using POST request => router.post('/', UsersController.setUser);

- get one user using GET request => router.get('/:id', UsersController.getUser);
- update user using PATCH request => router.patch('/:id', UsersController.UpdateUser);
2. Services routes connect service model with its controller
- get all services using GET request => router.get('/', ServicesController.getAllServices);
- create service using POST request => router.post('/', ServicesController.setService);
- get one service using GET request => router.get('/:id', ServicesController.getService);
- update service using PATCH request => router.patch('/:id', ServicesController.UpdateService);

Uploading Photos

Why uploading photos?

Having a profile picture for every user is an essential part of a successful app, so in Givantk app we made uploading of profile picture mandatory.
This approach was highly inspired by the study performed by the researcher Stijn Baert from Ghent university, Belgium. In his article, which is called “Facebook profile picture appearance

affects recruiters’ first hiring decisions” he discussed the effect of facebook profile pictures on the hiring choices.

The purpose of the study was to investigate whether the publicly available information on Facebook about job applicants affects employers’ hiring decisions.

The study concluded that Candidates with the most beneficial Facebook picture obtain approximately 38% more job interview invitations compared to candidates with the least beneficial picture.

So this study points out to the major importance of pictures on building trust, and obtaining higher hire rates.

Where to save images?

First we thought of saving the images on our backend server, but we had a technical issue, because whenever the server sleeps, we found that all the images are deleted because the server that we are using depends on what’s called ephemeral filesystem, where each dyno gets its own ephemeral filesystem, with a fresh copy of the most recently deployed code.

During the dyno’s lifetime its running processes can use the filesystem as a temporary scratchpad, but no files that are written are visible to processes in any other dyno and any files written will be discarded the moment the dyno is stopped or restarted.

 For example, this occurs any time a dyno is replaced due to application deployment and approximately once a day as part of normal dyno management.

And as you can see this is an unreliable process, because all of the images will be deleted after few hours, and users will have to upload it again, which will make the user very frustrated.

The alternative for this was using Amazon Simple Storage Service (Amazon S3), which is an object storage service that offers industry-leading scalability, data availability, security, and performance.
This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics.
Amazon S3 provides easy-to-use management features so you can organize your data and configure finely-tuned access

controls to meet your specific business, organizational, and compliance requirements. And it’s also designed for 99.999999999% (11 9's) of durability, and stores data for millions of applications for companies all around the world.

How we integrated it with front and backend?

In the front-end we took the image from gallery or access the camera with the aid of image picker module from expo package. This step happens during the making of the user profile.

Using file system module imported from expo package, we check whether the image is smaller than 5 mega bytes

The next step was to send it to the backend as a “multi/part form data”

In the backend we used the multer package to integrate with amazon s3

The pictures above demonstrate the following:

We first set the amazon aws configurations, by updating the secret access key, and the access key id.

Then we create a new s3 instance, and use it inside multer function, and set the name of bucket in which we store the pictures in, and the access control list type, which in this case is “public-read” as everybody is supposed to see these photos.

After that we send the following parameter to the router of making profile, which is (upload.single(‘avatar’)), this will result in the appearance of a file property in the request handled by the controller.
Also this file property has another property called location which holds the url on the Amazon Aws Server which displays this photo, we simply access it, then save it in the database in order to send it back to the frontend whenever a user image is to be displayed.

Search System

Why did we implement a search bar? it's a very crucial technology that must be implemented in any app and to make it a user friendly app or the user will ignore the app. That being said we decided to make the search bar stunning on both the front-end and back-end.

So how did we do it, we used react native styling for the front-end with a placeholder to make it easier for the user to identify the search bar. This styling consists of a view, a text Input and a button all styled according to the main theme of the app which is blue and grey.

For the back-end of the search bar we used node.js and mongo db to access the database and retrieve whatever the user searched for and present it in a new view screen made only for search purposes and separate it from the main screen where all the services can be found, if the user searched for a service that is not found the search screen will not load anything and notify the user that the service is not found.

this search feature was added to search for services. The search key words are meant for only the title of the service (not the description). The search is not case-sensitive, the user doesn't have to type the exact words to match up to a title but he can write a small keyword and the server will find out if this keyword is included in the title of the service.

e.g.: if a user type the word "shoes" the server will output all the services that have "shoes" in the title or more than this word.

The search route is found in the services routes (/routes/services.js):

```
// @route  GET api/service/search
// @desc   search for a service
// @access Private
// @errors noservice error
router.get(
  '/search',
  passport.authenticate('jwt', { session: false }),
  serviceController.search
);
```

The search controller is found in the services controller (/controllers/serviceController/search.js):

```
const mongoose = require('mongoose');

// Models
const Service = mongoose.model('service');

module.exports = search = (req, res) => {
  const errors = {};
  Service.find({ name: new RegExp('^' + req.params.name + '$',
'i') })
    .sort({ date: -1 })
    .then((services) => {
      if (services.length === 0) {
        errors.noservices = 'No services found';
        return res.status(404).json(errors);
      }
      return res.json(services);
    })
    .catch((err) => {
      errors.error = 'Error fetching services from database';
      res.status(500).json({ ...errors, ...err });
    });
};



class SearchResultsScreen extends Component {
  static navigationOptions = ({ navigation }) => ({
    headerTitle: 'Search Results',
    headerStyle: {
      backgroundColor: colors.primary,
    },
    headerTitleStyle: {
      color: colors.white,
    },
  });

  render() {
    const { navigation, searchedServices,
getSearchedServicesLoading } = this.props;
    return (
      <View style={styles.wrapper}>
        {searchedServices && (
          <ServicesList
            services={searchedServices}
            loading={getSearchedServicesLoading}
            navigation={navigation}
          />
        )}
      </View>
    );
  }
}
```

Chat System

To make the user experience more engaging and responsive we decided to make a chatting system where the asker can speak with the owner and vice versa about the service that the owner posted and discuss everything about the service and how the owner want the service to be implemented.
Though the chatting system will be between the owner and the asker, the admins of the app can detect any inappropriate language and any agreement made outside the service will also be detected by the admins.
The chatting system is established between two users only, it's a private chat between the service owner and the service helper.

```
const express = require('express');
const router = express.Router();
const passport = require('passport');

const ChatController =
require('../controllers/chatController/index');

// @route  POST api/profile/id1+id2
// @desc   Open chat between 2 users
// @access Private
// @errors noprofiles error
router.post(
  '/:id1+:id2',
  passport.authenticate('jwt', { session: false }),
  ChatController.openChat
);

module.exports = router;
```

We used socket.io package to take information from the client and send it to the server then the server distributes the message to the client on the other end.

```
const io = socket(server);
io.on('connection', function(socket){ // waiting for connection
with a client
        console.log('socket connection made and the socket id is ' +
socket.id);
        socket.on('chat', function(data){
        // waiting for a data to be send from a client
        io.sockets.emit('chat', data)
        // send the message back from the server to the rest of the
clients
    });
});
```

The chat history will be stored in mongoDB database and it consists of first user id, second user id, message of the first user and message of the second user.

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const ChatSchema = new Schema({
 userID1: {
  type: mongoose.Schema.Types.ObjectId,
  ref: 'user',
  required: true
 },

 msg1: {
  type: String,
  required: true
 },

 userID2: {
  type: mongoose.Schema.Types.ObjectId,
  ref: 'user',
  required: true
 },
  msg2: {
    type: String,
    required: true
  }
});

const Chat = mongoose.model('Chat', ChatSchema);
module.exports = Chat;
```

As for the front-end of the chatting system we decided to make it more modern and user-friendly so we made the current user messages with the green color and the message of the user on the other end with the grey color, also we used our normal app theme color which is blue and grey for the input text and the send button.
The send button position is not affected by the appearance of the keyboard on the screen and its position will be moved upward until the end of the keyboard.

The messages that are typed by both users have the ability to scroll down on its own so that the user does not have to scroll it down and became a frustrating experience for him.

```
      <View style={styles.wrapper}>
```

```
        <ScrollView ref={ref => this.scrollView = ref}
          onContentSizeChange={(contentWidth,
contentHeight)=>{
              this.scrollView.scrollToEnd({animated: true});
          }}
        >
          <View>{chatHistory}</View>
          <View>{chatMessages}</View>
        </ScrollView>

        <KeyboardAvoidingView behavior="padding"
keyboardVerticalOffset={85}>
          <ChatInputItem
            autoCorrect={false}
            value={this.state.chatMessage}
            onPress={() => this.submitChatMessage()}
            onChangeText={(chatMessage) => {
              this.setState({ chatMessage });
            }}
          />
        </KeyboardAvoidingView>

      </View>
```

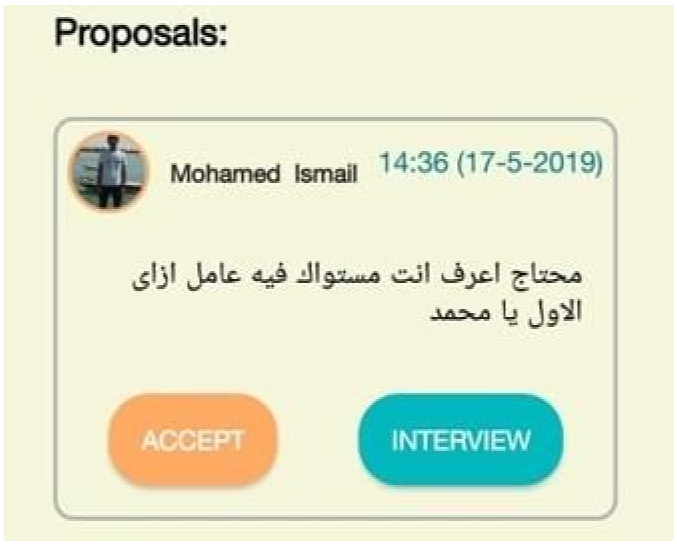
Linking Chat to Services

After Ending the development of the chat, we found that it’s better to link chat to a specific service, and not allowing users to talk with each other without linking this chat to a certain service, as the core of the app is about services, not social networking with other users.

This approach is also followed by the largest freelancing companies such that Upwork and Freelancer.

How this looks like in the app?

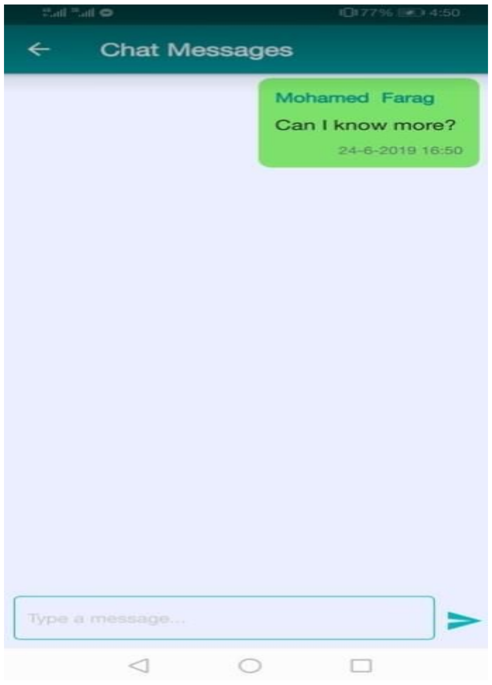
Whenever a user requests a service, and other users apply to this service, the asker has two options as in the image below:


Proposals:

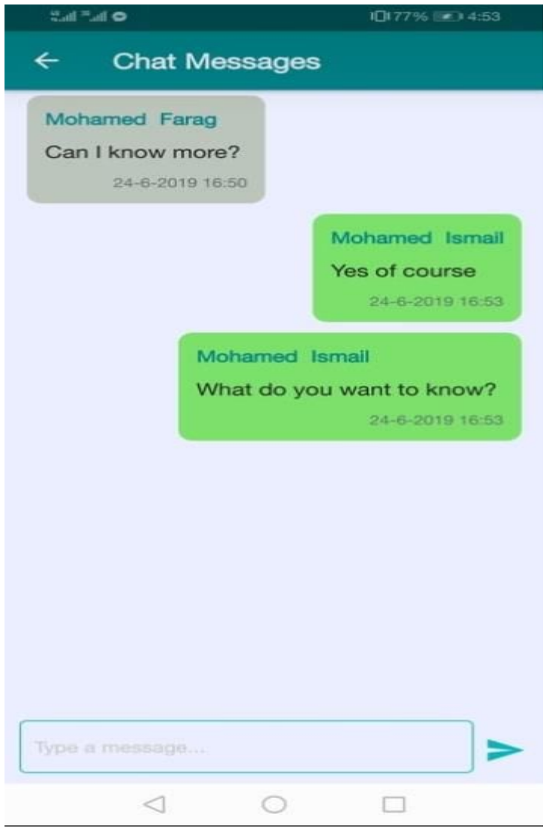
Mohamed Ismail 14:36 (17-5-2019)

محتاج اعرف انت مستواك فيه عامل ازاى الاول يا محمد

Here we see that this is a proposal for a service, where the asker is asking about someone who can train him in Autocad software , and we can find here that there are two options, the first is to accept this proposal, and the other one is “interview”, this option guarantees a way of communication between the asker and the proposal maker, before hiring them.
We also wanted to make this interview a private one, so we didn’t add it as a group of comments, but we preferred to make it a private chat.
Note that the interview option doesn’t appear to the proposal maker, but we let the initiative to be made by the asker.
Whenever the asker clicks on the interview button, A chat between the two of them (the asker and the helper) is constructed, and a push notification is sent to the helper, whenever the asker sends him a message.



Mohamed Farag
Can I know more?
24-6-2019 16:50



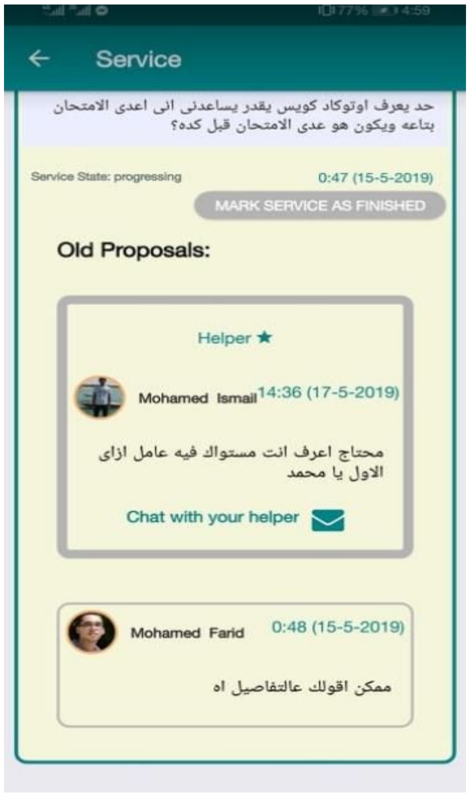
Mohamed Farag
Can I know more?
24-6-2019 16:50

Mohamed Ismail
Yes of course
24-6-2019 16:53

Mohamed Ismail
What do you want to know?
24-6-2019 16:53

In the images above we see that the asker sent a message to the proposal maker, and that the proposal maker responded to him asking more questions.
This will be a very efficient way to test services providers, before accepting them as service helpers.

Once an agreement happens, the asker can click on the accept button, then as the image below indicates, the “interview” button no longer appears.



حد يعرف اوتوكاد كويس يقدر يساعدنى انى اعدى الامتحان بتاعه ويكون هو عدى الامتحان قبل كده؟

Service State: progressing 0:47 (15-5-2019)

Old Proposals:

Helper ★

Mohamed Ismail 14:36 (17-5-2019)

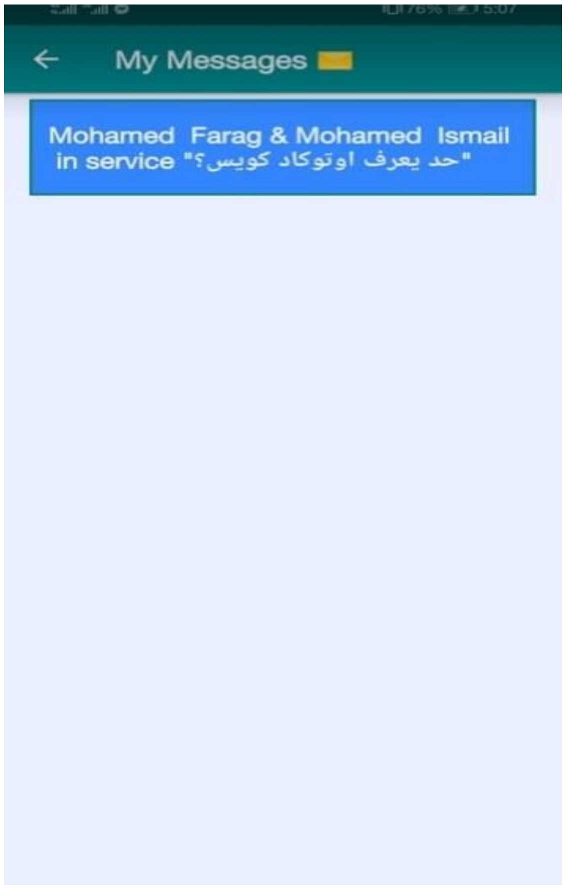
محتاج اعرف انت مستواك فيه عامل ازاى الاول يا محمد

Mohamed Farid 0:48 (15-5-2019)

ممكن اقولك عالتفاصيل اه

As we see hare a “chat with your helper” button is added, instead of the interview button, we also realize that the interview button also disappeared from other proposals, as interviews can no longer be proceeded while there’s a helper for the service.
So the chat can be accessed by two ways: the first way is from the service itself, from a button attached to the proposal, but the second way is from the user profile, where the user can see all the chat that he was involved in since he joined the app.
Each chat in the chat list is displayed in the following format: “The name of the first user” & “The name of the second user” in the service “the name of the service”.

And here’s an example below for such format:



Mohamed Farag & Mohamed Ismail in service "حد يعرف اوتوكاد كويس؟"

Admin Website

Admin website is built using React JS.

The website consists of these main pages:

1-Home page:

This page consists of the most important statistics of the app such as :

- Number of users.
- Number of services.
- Number of messages sent by users of the app to the admins.
- The app total income.

It also contains the most important tables that hold the information of users and services such as:

- Users personal information.
- Users points and skills.
- Finished Services.
- Unfinished Services.

2-Users Page:

It consists of the detailed tables that have information related to the users.

All the tables have search feature where they can search for all the results that match the keyword that they have entered.

The tables can handle big data, and it’s also provided with paging to avoid disturbing the admin or lagging the page with rendering such amount of data that will be existed within the app database.

Admin can take several actions on users from table such as banning users.

The tables consist of tables related to the user such as:

- personal information like: first name, last name, email, location, date.

- Different skills of users, and their points either they are givantk points (free points) or money points.

- A table linking the users to services, and it contains the number of services that the users have done, the number of services the user have asked for, and the number of unfinished services of the user. This table merely gives statistical information

- A table that that links every user id with a column that contains a button that leads to a page that contains the services that the user has done.

- A table that that links every user id with a column that contains a button that leads to a page that contains the services that the user has asked for.

3-Services page:

It contains the same system of tables that exist in the user page but with different type of information that is concerned mainly with services, such as:

- Services table that contains the id of the service, the title of the service ,the identity of asker, the identity of applicants to services, date of the service , state of the service, and a button that leads to a page that contains the detailed information of the service

- A table contains the information of unfinished services such as id, date, and link to page that contains list of them.

- A table contains the information of finished services such as id, date, and link to page that contains list of them.

All these tables as we said before are provided with search to facilitate extracting such amount of big data.

4-Statistics page:

It contains all the statistics of the app such as:

- Number of all users.
- Number of approved users.
- Number of banned users.
- Number of users registered today.
- Number of services.
- Number of services done.
- Number of undone services.
- Number of pending services.
- Number of customers’ messages.
- Number of transactions.
- Transactions value.
- Today’s app income.

All these information and more are provided inside cards that show the information in elegant way.

5-Messages Page:

This page contains a list of all customer messages and it shows the email of each sender, when the admin clicks on it a card expands with the content of the message, the admin can then reply to the message and send it by clicking on a button.

6-Activities:

This page consists of the most recent activities with the ability to delete it, it contains a list, and each component of this list contains the photo of the one who did the activity and also a description of the user name and which activity did he do, and when clicking on it a card appears that contains the content of this activity, when the admin sees an inappropriate activity he clicks on the delete activity button.

7-Settings:

In this page the admin determines the main settings that affects the app, with a simple form. He sets all the options that he needs to set, then simply clicks on submit button to send all theses changes to the back-end.

From the settings examples:

- Enabling services publishing or not.
- Enabling Registration.
- Displaying ads or not.
- Limiting number of published services per hour.
- Limiting number of advertisements per page.

8-Announcements Page:

It contains a text area where the user can write an announcement then send it to be displayed in the app informing users of recent news.

The users are notified by the announcement with a push notification, whenever the admin clicks on the send announcement button, as the announcement transfers to the backend, then it's displayed to all the users in the app.

Python Recommender System Overview

We need a recommender system that recommends the new services to users who may be interested to do these services .
Real life examples of recommender system :
Netflix , Linkin , Youtube .

Netflix recommender system recommend a movie for a certain user depends on the Movie's rates and user's rates for the movies that he has seen before .
Then depends on these two factors , the system can guess if the user will like a movie or dislike it.

Can we use Netflix recommender system? Why?

No, Netflix recommender system dependes on two factors :
Movie's rates and user's rates for movies that he has seen before

. In our application we will have user's rates for services that he did before but we will not have a rates for the new services . Then what the solution ?
We need to build system that can find the similarity between the new services and the previous services .
Then according to the similarity between the services, and the user's rates of services that he did , we can suggest a new suitable service for this user .

We will build this system the following steps :
1. Dataset : Find dataset that fits **our application**.
2. Preprocess the Data : Preprocessing the data is an essential step in natural language process. We will convert our class labels to binary values using the LabelEncoder from sklearn, replace email addresses, URLs, phone numbers, and other symbols by using regular expressions, remove stop words, and extract word stems.
3. Generating Features : Feature engineering is the process of using domain knowledge of the data to create features for machine learning algorithms.
4. Scikit-Learn Classifiers with NLTK : We'll need to import each algorithm we plan on using from sklearn. We also need to import some performance metrics, such as accuracy_score and classification_report.

Main JavaScript packages used and their description

Back-End Packages

1.1) Mongoose:
Getting Started
First be sure you have MongoDB and Node.js installed.
Next install Mongoose from the command line using npm:

```
$ npm install mongoose
```

Now say we like fuzzy kittens and want to record every kitten we ever meet in MongoDB. The first thing we need to do is include mongoose in our project and open a connection to the test database on our locally running instance of MongoDB.

```
// getting-started.js
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test', {useNewUrlParser: true});
```

We have a pending connection to the test database running on localhost. We now need to get notified if we connect successfully or if a connection error occurs:

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

Once our connection opens, our callback will be called. For brevity, let's assume that all following code is within this callback.
With Mongoose, everything is derived from a Schema. Let's get a reference to it and define our kittens.

```
var kittySchema = new mongoose.Schema({
  name: String
});
```

So far so good. We've got a schema with one property, name, which will be a String. The next step is compiling our schema into a Model.

```
var Kitten = mongoose.model('Kitten', kittySchema);
```

A model is a class with which we construct documents. In this case, each document will be a kitten with properties and behaviors as declared in our schema. Let's create a kitten document representing the little guy we just met on the sidewalk outside:

```
var silence = new Kitten({ name: 'Silence' });
console.log(silence.name); // 'Silence'
```

Kittens can meow, so let's take a look at how to add "speak" functionality to our documents:

```
// NOTE: methods must be added to the schema before compiling it with mongoose.model()
```

```
kittySchema.methods.speak = function () {
  var greeting = this.name
    ? "Meow name is " + this.name
    : "I don't have a name";
  console.log(greeting);
}

var Kitten = mongoose.model('Kitten', kittySchema);
```
Functions added to the methods property of a schema get compiled into the Model prototype and exposed on each document instance:
```
var fluffy = new Kitten({ name: 'fluffy' });
fluffy.speak(); // "Meow name is fluffy"
```
We have talking kittens! But we still haven't saved anything to MongoDB. Each document can be saved to the database by calling its save method. The first argument to the callback will be an error if any occurred.
```
  fluffy.save(function (err, fluffy) {
    if (err) return console.error(err);
    fluffy.speak();
  });
```
Say time goes by and we want to display all the kittens we've seen. We can access all of the kitten documents through our Kitten model.
```
Kitten.find(function (err, kittens) {
  if (err) return console.error(err);
  console.log(kittens);
})
```
We just logged all of the kittens in our db to the console. If we want to filter our kittens by name, Mongoose supports MongoDBs rich querying syntax.
```
Kitten.find({ name: /^fluff/ }, callback);
```
This performs a search for all documents with a name property that begins with "Fluff" and returns the result as an array of kittens to the callback.

Congratulations

That's the end of our quick start. We created a schema, added a custom document method, saved and queried kittens in MongoDB using Mongoose. Head over to the guide, or API docs for more.

1.2) bcryptjs

Usage

The library is compatible with CommonJS and AMD loaders and is exposed globally as dcodeIO.bcrypt if neither is available.

node.js

On node.js, the inbuilt crypto module's randomBytes interface is used to obtain secure random numbers.
```
npm install bcryptjs
var bcrypt = require('bcryptjs');
```

```
...
```

Browser

In the browser, bcrypt.js relies on Web Crypto API's getRandomValues interface to obtain secure random numbers. If no cryptographically secure source of randomness is available, you may specify one through bcrypt.setRandomFallback.

```
var bcrypt = dcodeIO.bcrypt;
...
```

or

```
require.config({
    paths: { "bcrypt": "/path/to/bcrypt.js" }
});
require(["bcrypt"], function(bcrypt) {
    ...
});
```

Usage - Sync

To hash a password:

```
var bcrypt = require('bcryptjs');
var salt = bcrypt.genSaltSync(10);
var hash = bcrypt.hashSync("B4c0/\/", salt);
// Store hash in your password DB.
```

To check a password:

```
// Load hash from your password DB.
bcrypt.compareSync("B4c0/\/", hash); // true
bcrypt.compareSync("not_bacon", hash); // false
```

Auto-gen a salt and hash:

```
var hash = bcrypt.hashSync('bacon', 8);
```

Usage - Async

To hash a password:

```
var bcrypt = require('bcryptjs');
bcrypt.genSalt(10, function(err, salt) {
    bcrypt.hash("B4c0/\/", salt, function(err, hash) {
        // Store hash in your password DB.
    });
});
```

To check a password:

```
// Load hash from your password DB.
bcrypt.compare("B4c0/\/", hash, function(err, res) {
    // res === true
});
bcrypt.compare("not_bacon", hash, function(err, res) {
    // res === false
});
```

```
// As of bcryptjs 2.4.0, compare returns a promise if callback is omitted:
bcrypt.compare("B4c0/\/", hash).then((res) => {
   // res === true
});
```
Auto-gen a salt and hash:
```
bcrypt.hash('bacon', 8, function(err, hash) {
});
```
Note: Under the hood, asynchronisation splits a crypto operation into small chunks. After the completion of a chunk, the execution of the next chunk is placed on the back of JS event loop queue, thus efficiently sharing the computational resources with the other operations in the queue.

API

setRandomFallback(random)

Sets the pseudo random number generator to use as a fallback if neither node's cryptomodule nor the Web Crypto API is available. Please note: It is highly important that the PRNG used is cryptographically secure and that it is seeded properly!

Parameter	Type	Description
random	function(number):!Array.<number>	Function taking the number of bytes to generate as its sole argument, returning the corresponding array of cryptographically secure random byte values.
@see	http://nodejs.org/api/crypto.html	
@see	http://www.w3.org/TR/WebCryptoAPI/	

Hint: You might use isaac.js as a CSPRNG but you still have to make sure to seed it properly.

genSaltSync(rounds=, seed_length=)

Synchronously generates a salt.

Parameter	Type	Description
rounds	number	Number of rounds to use, defaults to 10 if omitted
seed_length	number	Not supported.
@returns	string	Resulting salt
@throws	Error	If a random fallback is required but not set

genSalt(rounds=, seed_length=, callback)

Asynchronously generates a salt.

Parameter	Type	Description
rounds	number | function(Error, string=)	Number of rounds to use, defaults to 10 if omitted
seed_length	number | function(Error, string=)	Not supported.
callback	function(Error, string=)	Callback receiving the error, if any, and the resulting salt
@returns	Promise	If callback has been omitted
@throws	Error	If callback is present but not a function

hashSync(s, salt=)

Synchronously generates a hash for the given string.

Parameter	Type	Description
s	string	String to hash

salt	number | string	Salt length to generate or salt to use, default to 10
@returns	string	Resulting hash

hash(s, salt, callback, progressCallback=)

Asynchronously generates a hash for the given string.

Parameter	Type	Description
s	string	String to hash
salt	number | string	Salt length to generate or salt to use
callback	function(Error, string=)	Callback receiving the error, if any, and the resulting hash
progressCallback	function(number)	Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX_EXECUTION_TIME = 100 ms.
@returns	Promise	If callback has been omitted
@throws	Error	If callback is present but not a function

compareSync(s, hash)

Synchronously tests a string against a hash.

Parameter	Type	Description
s	string	String to compare
hash	string	Hash to test against
@returns	boolean	true if matching, otherwise false
@throws	Error	If an argument is illegal

compare(s, hash, callback, progressCallback=)

Asynchronously compares the given data against the given hash.

Parameter	Type	Description
s	string	Data to compare
hash	string	Data to be compared to
callback	function(Error, boolean)	Callback receiving the error, if any, otherwise the result
progressCallback	function(number)	Callback successively called with the percentage of rounds completed (0.0 - 1.0), maximally once per MAX_EXECUTION_TIME = 100 ms.
@returns	Promise	If callback has been omitted
@throws	Error	If callback is present but not a function

getRounds(hash)

Gets the number of rounds used to encrypt the specified hash.

Parameter	Type	Description
hash	string	Hash to extract the used number of rounds from
@returns	number	Number of rounds used
@throws	Error	If hash is not a string

getSalt(hash)

Gets the salt portion from a hash. Does not validate the hash.

Parameter	Type	Description
hash	string	Hash to extract the salt from
@returns	string	Extracted salt part
@throws	Error	If hash is not a string or otherwise invalid

1.3) passport

Overview

Passport is authentication middleware for Node. It is designed to serve a singular purpose: authenticate requests. When writing modules, encapsulation is a virtue, so Passport delegates all other functionality to the application. This separation of concerns keeps code clean and maintainable, and makes Passport extremely easy to integrate into an application.

In modern web applications, authentication can take a variety of forms. Traditionally, users log in by providing a username and password. With the rise of social networking, single sign-on using an OAuth provider such as Facebook or Twitter has become a popular authentication method. Services that expose an API often require token-based credentials to protect access.

Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as strategies, are packaged as individual modules. Applications can choose which strategies to employ, without creating unnecessary dependencies.

Despite the complexities involved in authentication, code does not have to be complicated.

```
app.post('/login', passport.authenticate('local', { successRedirect: '/',
                                                    failureRedirect: '/login' }));
```

Install

```
$ npm install passport
```

Authenticate

Authenticating requests is as simple as calling passport.authenticate() and specifying which strategy to employ. authenticate()'s function signature is standard Connect middleware, which makes it convenient to use as route middleware in Express applications.

```
app.post('/login',
  passport.authenticate('local'),
  function(req, res) {
    // If this function gets called, authentication was successful.
    // `req.user` contains the authenticated user.
    res.redirect('/users/' + req.user.username);
  });
```

By default, if authentication fails, Passport will respond with a 401 Unauthorized status, and any additional route handlers will not be invoked. If authentication succeeds, the next handler will be invoked and the req.user property will be set to the authenticated user.

Note: Strategies must be configured prior to using them in a route. Continue reading the chapter on configuration for details.

Redirects

A redirect is commonly issued after authenticating a request.

```
app.post('/login',
  passport.authenticate('local', { successRedirect: '/',
                                   failureRedirect: '/login' }));
```

In this case, the redirect options override the default behavior. Upon successful authentication, the user will be redirected to the home page. If authentication fails, the user will be redirected back to the login page for another attempt.

Flash Messages

Redirects are often combined with flash messages in order to display status information to the user.

```
app.post('/login',
  passport.authenticate('local', { successRedirect: '/',
                                   failureRedirect: '/login',
                                   failureFlash: true })
);
```

Setting the failureFlash option to true instructs Passport to flash an error message using the message given by the strategy's verify callback, if any. This is often the best approach, because the verify callback can make the most accurate determination of why authentication failed.

Alternatively, the flash message can be set specifically.

```
passport.authenticate('local', { failureFlash: 'Invalid username or password.' });
```

A successFlash option is available which flashes a success message when authentication succeeds.

```
passport.authenticate('local', { successFlash: 'Welcome!' });
```

Note: Using flash messages requires a req.flash() function. Express 2.x provided this functionality, however it was removed from Express 3.x. Use of connect-flash middleware is recommended to provide this functionality when using Express 3.x.

Disable Sessions

After successful authentication, Passport will establish a persistent login session. This is useful for the common scenario of users accessing a web application via a browser. However, in some cases, session support is not necessary. For example, API servers typically require credentials to be supplied with each request. When this is the case, session support can be safely disabled by setting the session option to false.

```
app.get('/api/users/me',
  passport.authenticate('basic', { session: false }),
  function(req, res) {
    res.json({ id: req.user.id, username: req.user.username });
  });
```

Custom Callback

If the built-in options are not sufficient for handling an authentication request, a custom callback can be provided to allow the application to handle success or failure.

```
app.get('/login', function(req, res, next) {
  passport.authenticate('local', function(err, user, info) {
    if (err) { return next(err); }
    if (!user) { return res.redirect('/login'); }
    req.logIn(user, function(err) {
      if (err) { return next(err); }
      return res.redirect('/users/' + user.username);
    });
  })(req, res, next);
});
```

In this example, note that authenticate() is called from within the route handler, rather than being used as route middleware. This gives the callback access to the req and res objects through closure.

If authentication failed, user will be set to false. If an exception occurred, err will be set. An optional info argument will be passed, containing additional details provided by the strategy's verify callback.

The callback can use the arguments supplied to handle the authentication result as desired. Note that when using a custom callback, it becomes the application's responsibility to establish a session (by calling req.login()) and send a response.

Configure

Three pieces need to be configured to use Passport for authentication:

- Authentication strategies
- Application middleware
- Sessions (optional)

Strategies

Passport uses what are termed strategies to authenticate requests. Strategies range from verifying a username and password, delegated authentication using OAuth or federated authentication using OpenID.

Before asking Passport to authenticate a request, the strategy (or strategies) used by an application must be configured.

Strategies, and their configuration, are supplied via the use() function. For example, the following uses the LocalStrategy for username/password authentication.

```
var passport = require('passport')
  , LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (!user.validPassword(password)) {
        return done(null, false, { message: 'Incorrect password.' });
      }
      return done(null, user);
    });
  }
));
```

Verify Callback

This example introduces an important concept. Strategies require what is known as a verify callback. The purpose of a verify callback is to find the user that possesses a set of credentials. When Passport authenticates a request, it parses the credentials contained in the request. It then invokes the verify callback with those credentials as arguments, in this case username and password. If the credentials are valid, the verify callback invokes done to supply Passport with the user that authenticated.

```
return done(null, user);
```

If the credentials are not valid (for example, if the password is incorrect), done should be invoked with false instead of a user to indicate an authentication failure.

```
return done(null, false);
```

An additional info message can be supplied to indicate the reason for the failure. This is useful for displaying a flash message prompting the user to try again.

```
return done(null, false, { message: 'Incorrect password.' });
```

Finally, if an exception occurred while verifying the credentials (for example, if the database is not available), done should be invoked with an error, in conventional Node style.

```
return done(err);
```

Note that it is important to distinguish the two failure cases that can occur. The latter is a server exception, in which err is set to a non-null value. Authentication failures are natural conditions, in which the server is operating normally. Ensure that err remains null, and use the final argument to pass additional details.

By delegating in this manner, the verify callback keeps Passport database agnostic. Applications are free to choose how user information is stored, without any assumptions imposed by the authentication layer.

Middleware

In a Connect or Express-based application, passport.initialize() middleware is required to initialize Passport. If your application uses persistent login sessions, passport.session() middleware must also be used.

```
app.configure(function() {
  app.use(express.static('public'));
  app.use(express.cookieParser());
  app.use(express.bodyParser());
  app.use(express.session({ secret: 'keyboard cat' }));
  app.use(passport.initialize());
  app.use(passport.session());
  app.use(app.router);
});
```

Note that enabling session support is entirely optional, though it is recommended for most applications. If enabled, be sure to use session() before passport.session() to ensure that the login session is restored in the correct order.

In Express 4.x, the Connect middleware is no longer included in the Express core, and the app.configure() method has been removed. The same middleware can be found in their npm module equivalents.

```
var session = require("express-session"),
    bodyParser = require("body-parser");

app.use(express.static("public"));
app.use(session({ secret: "cats" }));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(passport.initialize());
app.use(passport.session());
```
Sessions

In a typical web application, the credentials used to authenticate a user will only be transmitted during the login request. If authentication succeeds, a session will be established and maintained via a cookie set in the user's browser.

Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session. In order to support login sessions, Passport will serialize and deserialize user instances to and from the session.
```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});
```
In this example, only the user ID is serialized to the session, keeping the amount of data stored within the session small. When subsequent requests are received, this ID is used to find the user, which will be restored to req.user.

The serialization and deserialization logic is supplied by the application, allowing the application to choose an appropriate database and/or object mapper, without imposition by the authentication layer.

Username & Password

The most widely used way for websites to authenticate users is via a username and password. Support for this mechanism is provided by the passport-local module.

Install
```
$ npm install passport-local
```
Configuration
```
var passport = require('passport')
  , LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function(err, user) {
      if (err) { return done(err); }
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (!user.validPassword(password)) {
        return done(null, false, { message: 'Incorrect password.' });
      }
      return done(null, user);
    });
  }
));
```

The verify callback for local authentication accepts username and password arguments, which are submitted to the application via a login form.

Form

A form is placed on a web page, allowing the user to enter their credentials and log in.

```
<form action="/login" method="post">
    <div>
        <label>Username:</label>
        <input type="text" name="username"/>
    </div>
    <div>
        <label>Password:</label>
        <input type="password" name="password"/>
    </div>
    <div>
        <input type="submit" value="Log In"/>
    </div>
</form>
```

Route

The login form is submitted to the server via the POST method. Using authenticate() with the localstrategy will handle the login request.

```
app.post('/login',
  passport.authenticate('local', { successRedirect: '/',
                                   failureRedirect: '/login',
                                   failureFlash: true })
);
```

Setting the failureFlash option to true instructs Passport to flash an error message using the messageoption set by the verify callback above. This is helpful when prompting the user to try again.

Parameters

By default, LocalStrategy expects to find credentials in parameters named username and password. If your site prefers to name these fields differently, options are available to change the defaults.

```
passport.use(new LocalStrategy({
    usernameField: 'email',
    passwordField: 'passwd'
  },
  function(username, password, done) {
    // ...
  }
));
```

OpenID

OpenID is an open standard for federated authentication. When visiting a website, users present their OpenID to sign in. The user then authenticates with their chosen OpenID provider, which issues an assertion to confirm the user's identity. The website verifies this assertion in order to sign the user in.

Support for OpenID is provided by the passport-openid module.

Install

```
$ npm install passport-openid
```

Configuration

When using OpenID, a return URL and realm must be specified. The returnURL is the URL to which the user will be redirected after authenticating with their OpenID provider. realm indicates the part of URL-space for which authentication is valid. Typically this will be the root URL of the website.

```
var passport = require('passport')
  , OpenIDStrategy = require('passport-openid').Strategy;

passport.use(new OpenIDStrategy({
    returnURL: 'http://www.example.com/auth/openid/return',
    realm: 'http://www.example.com/'
  },
  function(identifier, done) {
    User.findOrCreate({ openId: identifier }, function(err, user) {
      done(err, user);
    });
  }
));
```

The verify callback for OpenID authentication accepts an identifier argument containing the user's claimed identifier.

Form

A form is placed on a web page, allowing the user to enter their OpenID and sign in.

```
<form action="/auth/openid" method="post">
    <div>
        <label>OpenID:</label>
        <input type="text" name="openid_identifier"/><br/>
    </div>
    <div>
        <input type="submit" value="Sign In"/>
```

```
  </div>
</form>
```

Routes

Two routes are required for OpenID authentication. The first route accepts the form submission containing an OpenID identifier. During authentication, the user will be redirected to their OpenID provider. The second route is the URL to which the user will be returned after authenticating with their OpenID provider.

```
// Accept the OpenID identifier and redirect the user to their OpenID
// provider for authentication.  When complete, the provider will redirect
// the user back to the application at:
//     /auth/openid/return
app.post('/auth/openid', passport.authenticate('openid'));

// The OpenID provider has redirected the user back to the application.
// Finish the authentication process by verifying the assertion.  If valid,
// the user will be logged in.  Otherwise, authentication has failed.
app.get('/auth/openid/return',
  passport.authenticate('openid', { successRedirect: '/',
                                    failureRedirect: '/login' }));
```

Profile Exchange

OpenID can optionally be configured to retrieve profile information about the user being authenticated. Profile exchange is enabled by setting the profile option to true.

```
passport.use(new OpenIDStrategy({
    returnURL: 'http://www.example.com/auth/openid/return',
    realm: 'http://www.example.com/',
    profile: true
  },
  function(identifier, profile, done) {
    // ...
  }
));
```

When profile exchange is enabled, the function signature of the verify callback accepts an additional profile argument containing user profile information provided by the OpenID provider; refer to User Profile for further information.

OAuth

OAuth is a standard protocol that allows users to authorize API access to web and desktop or mobile applications. Once access has been granted, the authorized application can utilize the API on behalf of the user. OAuth has also emerged as a popular mechanism for delegated authentication.

OAuth comes in two primary flavors, both of which are widely deployed.

The initial version of OAuth was developed as an open standard by a loosely organized collective of web developers. Their work resulted in OAuth 1.0, which was superseded by OAuth 1.0a. This work has now been standardized by the IETF as RFC 5849.

Recent efforts undertaken by the Web Authorization Protocol Working Group have focused on defining OAuth 2.0. Due to the lengthy standardization effort, providers have proceeded to deploy implementations conforming to various drafts, each with slightly different semantics. Thankfully, Passport shields an application from the complexities of dealing with OAuth variants. In many cases, a provider-specific strategy can be used instead of the generic OAuth strategies described below. This cuts down on the necessary configuration, and accommodates any provider-specific quirks. See Facebook, Twitter or the list of providers for preferred usage.
Support for OAuth is provided by the passport-oauth module.
Install
```
$ npm install passport-oauth
```
OAuth 1.0
OAuth 1.0 is a delegated authentication strategy that involves multiple steps. First, a request token must be obtained. Next, the user is redirected to the service provider to authorize access. Finally, after authorization has been granted, the user is redirected back to the application and the request token can be exchanged for an access token. The application requesting access, known as a consumer, is identified by a consumer key and consumer secret.
Configuration
When using the generic OAuth strategy, the key, secret, and endpoints are specified as options.
```
var passport = require('passport')
  , OAuthStrategy = require('passport-oauth').OAuthStrategy;

passport.use('provider', new OAuthStrategy({
    requestTokenURL: 'https://www.provider.com/oauth/request_token',
    accessTokenURL: 'https://www.provider.com/oauth/access_token',
    userAuthorizationURL: 'https://www.provider.com/oauth/authorize',
    consumerKey: '123-456-789',
    consumerSecret: 'shhh-its-a-secret'
    callbackURL: 'https://www.example.com/auth/provider/callback'
  },
  function(token, tokenSecret, profile, done) {
    User.findOrCreate(..., function(err, user) {
      done(err, user);
    });
  }
));
```
The verify callback for OAuth-based strategies accepts token, tokenSecret, and profile arguments.token is the access token and tokenSecret is its corresponding secret. profile will contain user profile information provided by the service provider; refer to User Profile for additional information.
Routes
Two routes are required for OAuth authentication. The first route initiates an OAuth transaction and redirects the user to the service provider. The second route is the URL to which the user will be redirected after authenticating with the provider.

```
// Redirect the user to the OAuth provider for authentication.  When
// complete, the provider will redirect the user back to the application at
//     /auth/provider/callback
app.get('/auth/provider', passport.authenticate('provider'));

// The OAuth provider has redirected the user back to the application.
// Finish the authentication process by attempting to obtain an access
// token.  If authorization was granted, the user will be logged in.
// Otherwise, authentication has failed.
app.get('/auth/provider/callback',
  passport.authenticate('provider', { successRedirect: '/',
                                      failureRedirect: '/login' }));
```

Link

A link or button can be placed on a web page, which will start the authentication process when clicked.

```
<a href="/auth/provider">Log In with OAuth Provider</a>
```

OAuth 2.0

OAuth 2.0 is the successor to OAuth 1.0, and is designed to overcome perceived shortcomings in the earlier version. The authentication flow is essentially the same. The user is first redirected to the service provider to authorize access. After authorization has been granted, the user is redirected back to the application with a code that can be exchanged for an access token. The application requesting access, known as a client, is identified by an ID and secret.

Configuration

When using the generic OAuth 2.0 strategy, the client ID, client secret, and endpoints are specified as options.

```
var passport = require('passport')
  , OAuth2Strategy = require('passport-oauth').OAuth2Strategy;

passport.use('provider', new OAuth2Strategy({
    authorizationURL: 'https://www.provider.com/oauth2/authorize',
    tokenURL: 'https://www.provider.com/oauth2/token',
    clientID: '123-456-789',
    clientSecret: 'shhh-its-a-secret'
    callbackURL: 'https://www.example.com/auth/provider/callback'
  },
  function(accessToken, refreshToken, profile, done) {
    User.findOrCreate(..., function(err, user) {
      done(err, user);
    });
  }
));
```

The verify callback for OAuth 2.0-based strategies accepts accessToken, refreshToken, and profilearguments. refreshToken can be used to obtain new access tokens, and may be undefined if the provider does not issue refresh tokens. profile will contain user profile information provided by the service provider; refer to User Profile for additional information.

Routes

Two routes are required for OAuth 2.0 authentication. The first route redirects the user to the service provider. The second route is the URL to which the user will be redirected after authenticating with the provider.

```
// Redirect the user to the OAuth 2.0 provider for authentication.  When
// complete, the provider will redirect the user back to the application at
//     /auth/provider/callback
app.get('/auth/provider', passport.authenticate('provider'));

// The OAuth 2.0 provider has redirected the user back to the application.
// Finish the authentication process by attempting to obtain an access
// token.  If authorization was granted, the user will be logged in.
// Otherwise, authentication has failed.
app.get('/auth/provider/callback',
  passport.authenticate('provider', { successRedirect: '/',
                                      failureRedirect: '/login' }));
```

Scope

When requesting access using OAuth 2.0, the scope of access is controlled by the scope option.

```
app.get('/auth/provider',
  passport.authenticate('provider', { scope: 'email' })
);
```

Multiple scopes can be specified as an array.

```
app.get('/auth/provider',
  passport.authenticate('provider', { scope: ['email', 'sms'] })
);
```

Values for the scope option are provider-specific. Consult the provider's documentation for details regarding supported scopes.

Link

A link or button can be placed on a web page, which will start the authentication process when clicked.

```
<a href="/auth/provider">Log In with OAuth 2.0 Provider</a>
```

User Profile

When authenticating using a third-party service such as Facebook or Twitter, user profile information will often be available. Each service tends to have a different way of encoding this information. To make integration easier, Passport normalizes profile information to the extent possible.

Normalized profile information conforms to the contact schema established by [Joseph Smarr][schema-author]. The common fields available are outlined in the following table.

provider {String}

The provider with which the user authenticated (facebook, twitter, etc.).

id {String}
A unique identifier for the user, as generated by the service provider.
displayName {String}
The name of this user, suitable for display.
name {Object}
familyName {String}
The family name of this user, or "last name" in most Western languages.
givenName {String}
The given name of this user, or "first name" in most Western languages.
middleName {String}
The middle name of this user.
emails {Array} [n]
value {String}
The actual email address.
type {String}
The type of email address (home, work, etc.).
photos {Array} [n]
value {String}
The URL of the image.
Note that not all of the above fields are available from every service provider. Some providers may contain additional information not described here. Consult the provider-specific documentation for further details.
Facebook
The Facebook strategy allows users to log in to a web application using their Facebook account. Internally, Facebook authentication works using OAuth 2.0.
Support for Facebook is implemented by the passport-facebook module.
Install

```
$ npm install passport-facebook
```

Configuration
In order to use Facebook authentication, you must first create an app at Facebook Developers. When created, an app is assigned an App ID and App Secret. Your application must also implement a redirect URL, to which Facebook will redirect users after they have approved access for your application.

```
var passport = require('passport')
  , FacebookStrategy = require('passport-facebook').Strategy;

passport.use(new FacebookStrategy({
    clientID: FACEBOOK_APP_ID,
    clientSecret: FACEBOOK_APP_SECRET,
    callbackURL: "http://www.example.com/auth/facebook/callback"
  },
  function(accessToken, refreshToken, profile, done) {
    User.findOrCreate(..., function(err, user) {
      if (err) { return done(err); }
      done(null, user);
    });
  }
));
```
The verify callback for Facebook authentication accepts accessToken, refreshToken, and profilearguments. profile will contain user profile information provided by Facebook; refer to User Profile for additional information.
Note: For security reasons, the redirection URL must reside on the same host that is registered with Facebook.
Routes
Two routes are required for Facebook authentication. The first route redirects the user to Facebook. The second route is the URL to which Facebook will redirect the user after they have logged in.
```
// Redirect the user to Facebook for authentication.  When complete,
// Facebook will redirect the user back to the application at
//     /auth/facebook/callback
app.get('/auth/facebook', passport.authenticate('facebook'));

// Facebook will redirect the user to this URL after approval.  Finish the
// authentication process by attempting to obtain an access token.  If
// access was granted, the user will be logged in.  Otherwise,
// authentication has failed.
app.get('/auth/facebook/callback',
  passport.authenticate('facebook', { successRedirect: '/',
                                      failureRedirect: '/login' }));
```
Note that the URL of the callback route matches that of the callbackURL option specified when configuring the strategy.
Permissions
If your application needs extended permissions, they can be requested by setting the scope option.
```
app.get('/auth/facebook',
  passport.authenticate('facebook', { scope: 'read_stream' })
);
```
Multiple permissions can be specified as an array.
```
app.get('/auth/facebook',
  passport.authenticate('facebook', { scope: ['read_stream', 'publish_actions'] })
);
```
Link
A link or button can be placed on a web page, allowing one-click login with Facebook.
```
<a href="/auth/facebook">Login with Facebook</a>
```
Twitter

The Twitter strategy allows users to sign in to a web application using their Twitter account. Internally, Twitter authentication works using OAuth 1.0a.
Support for Twitter is implemented by the passport-twitter module.
Install
```
$ npm install passport-twitter
```
Configuration
In order to use Twitter authentication, you must first create an application at Twitter Developers. When created, an application is assigned a consumer key and consumer secret. Your application must also implement a callback URL, to which Twitter will redirect users after they have approved access for your application.
```
var passport = require('passport')
  , TwitterStrategy = require('passport-twitter').Strategy;

passport.use(new TwitterStrategy({
    consumerKey: TWITTER_CONSUMER_KEY,
    consumerSecret: TWITTER_CONSUMER_SECRET,
    callbackURL: "http://www.example.com/auth/twitter/callback"
  },
  function(token, tokenSecret, profile, done) {
    User.findOrCreate(..., function(err, user) {
      if (err) { return done(err); }
      done(null, user);
    });
  }
));
```
The verify callback for Twitter authentication accepts token, tokenSecret, and profile arguments. profilewill contain user profile information provided by Twitter; refer to User Profile for additional information.
Routes
Two routes are required for Twitter authentication. The first route initiates an OAuth transaction and redirects the user to Twitter. The second route is the URL to which Twitter will redirect the user after they have signed in.
```
// Redirect the user to Twitter for authentication.  When complete, Twitter
// will redirect the user back to the application at
//   /auth/twitter/callback
app.get('/auth/twitter', passport.authenticate('twitter'));

// Twitter will redirect the user to this URL after approval.  Finish the
// authentication process by attempting to obtain an access token.  If
// access was granted, the user will be logged in.  Otherwise,
// authentication has failed.
app.get('/auth/twitter/callback',
  passport.authenticate('twitter', { successRedirect: '/',
                                failureRedirect: '/login' }));
```
Note that the URL of the callback route matches that of the callbackURL option specified when configuring the strategy.

Link

A link or button can be placed on a web page, allowing one-click sign in with Twitter.

```
<a href="/auth/twitter">Sign in with Twitter</a>
```

Google

The Google strategy allows users to sign in to a web application using their Google account. Google used to support OpenID internally, but it now works based on OpenID Connect and supports oAuth 1.0 and oAuth 2.0.

Support for Google is implemented by the passport-google-oauth module.

Install

```
$ npm install passport-google-oauth
```

Configuration

The Client Id and Client Secret needed to authenticate with Google can be set up from the Google Developers Console. You may also need to enable Google+ API in the developer console, otherwise user profile data may not be fetched. Google supports authentication with both oAuth 1.0 and oAuth 2.0.

oAuth 1.0

The Google OAuth 1.0 authentication strategy authenticates users using a Google account and OAuth tokens. The strategy requires a verify callback, which accepts these credentials and calls done providing a user, as well as options specifying a consumer key, consumer secret, and callback URL.

Configuration

```
var passport = require('passport');
var GoogleStrategy = require('passport-google-oauth').OAuthStrategy;

// Use the GoogleStrategy within Passport.
//   Strategies in passport require a `verify` function, which accept
//   credentials (in this case, a token, tokenSecret, and Google profile), and
//   invoke a callback with a user object.
passport.use(new GoogleStrategy({
    consumerKey: GOOGLE_CONSUMER_KEY,
    consumerSecret: GOOGLE_CONSUMER_SECRET,
    callbackURL: "http://www.example.com/auth/google/callback"
  },
  function(token, tokenSecret, profile, done) {
      User.findOrCreate({ googleId: profile.id }, function (err, user) {
        return done(err, user);
      });
  }
));
```
Routes

Use passport.authenticate(), specifying the 'google' strategy, to authenticate requests. Authentication with Google requires an extra scope parameter. For information, go here.

```
// GET /auth/google
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  The first step in Google authentication will involve redirecting
//   the user to google.com.  After authorization, Google will redirect the user
//   back to this application at /auth/google/callback
app.get('/auth/google',
  passport.authenticate('google', { scope: 'https://www.google.com/m8/feeds' });

// GET /auth/google/callback
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  If authentication fails, the user will be redirected back to the
//   login page.  Otherwise, the primary route function function will be called,
//   which, in this example, will redirect the user to the home page.
app.get('/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/');
  });
```

oAuth 2.0

The Google OAuth 2.0 authentication strategy authenticates users using a Google account and OAuth 2.0 tokens. The strategy requires a verify callback, which accepts these credentials and calls done providing a user, as well as options specifying a client ID, client secret, and callback URL.

Configuration

```
var passport = require('passport');
var GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;

// Use the GoogleStrategy within Passport.
//   Strategies in Passport require a `verify` function, which accept
//   credentials (in this case, an accessToken, refreshToken, and Google
//   profile), and invoke a callback with a user object.
passport.use(new GoogleStrategy({
    clientID: GOOGLE_CLIENT_ID,
    clientSecret: GOOGLE_CLIENT_SECRET,
    callbackURL: "http://www.example.com/auth/google/callback"
  },
  function(accessToken, refreshToken, profile, done) {
       User.findOrCreate({ googleId: profile.id }, function (err, user) {
         return done(err, user);
       });
  }
));
```

Routes

Use passport.authenticate(), specifying the 'google' strategy, to authenticate requests.
Authentication with Google requires an extra scope parameter. For information, go here.

```
// GET /auth/google
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  The first step in Google authentication will involve
//   redirecting the user to google.com.  After authorization, Google
//   will redirect the user back to this application at /auth/google/callback
app.get('/auth/google',
  passport.authenticate('google', { scope: ['https://www.googleapis.com/auth/plus.login'] }));

// GET /auth/google/callback
//   Use passport.authenticate() as route middleware to authenticate the
//   request.  If authentication fails, the user will be redirected back to the
//   login page.  Otherwise, the primary route function function will be called,
//   which, in this example, will redirect the user to the home page.
app.get('/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/');
  });
```

Link

A link or button can be placed on a web page, allowing one-click sign in with Google.

```
<a href="/auth/google">Sign In with Google</a>
```

Basic & Digest

Along with defining HTTP's authentication framework, RFC 2617 also defined the Basic and Digest authentications schemes. These two schemes both use usernames and passwords as credentials to authenticate users, and are often used to protect API endpoints.

It should be noted that relying on username and password creditials can have adverse security impacts, especially in scenarios where there is not a high degree of trust between the server and client. In these situations, it is recommended to use an authorization framework such as OAuth 2.0.

Support for Basic and Digest schemes is provided by the passport-http module.

Install

```
$ npm install passport-http
```

Basic

The Basic scheme uses a username and password to authenticate a user. These credentials are transported in plain text, so it is advised to use HTTPS when implementing this scheme.

Configuration

```
passport.use(new BasicStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      if (!user.validPassword(password)) { return done(null, false); }
      return done(null, user);
    });
  }
));
```
The verify callback for Basic authentication accepts username and password arguments.
Protect Endpoints
```
app.get('/api/me',
  passport.authenticate('basic', { session: false }),
  function(req, res) {
    res.json(req.user);
  });
```
Specify passport.authenticate() with the basic strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.
Digest
The Digest scheme uses a username and password to authenticate a user. Its primary benefit over Basic is that it uses a challenge-response paradigm to avoid sending the password in the clear.
Configuration
```
passport.use(new DigestStrategy({ qop: 'auth' },
  function(username, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      return done(null, user, user.password);
    });
  },
  function(params, done) {
    // validate nonces as necessary
    done(null, true)
  }
));
```
The Digest strategy utilizes two callbacks, the second of which is optional.
The first callback, known as the "secret callback" accepts the username and calls done supplying a user and the corresponding secret password. The password is used to compute a hash, and authentication fails if it does not match that contained in the request.
The second "validate callback" accepts nonce related params, which can be checked to avoid replay attacks.
Protect Endpoints
```
app.get('/api/me',
  passport.authenticate('digest', { session: false }),
  function(req, res) {
    res.json(req.user);
  });
```
Specify passport.authenticate() with the digest strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

OAuth

OAuth (formally specified by RFC 5849) provides a means for users to grant third-party applications access to their data without exposing their password to those applications.

The protocol greatly improves the security of web applications, in particular, and OAuth has been important in bringing attention to the potential dangers of exposing passwords to external services.

While OAuth 1.0 is still widely used, it has been superseded by OAuth 2.0. It is recommended to base new implementations on OAuth 2.0.

When using OAuth to protect API endpoints, there are three distinct steps that that must be performed:

- The application requests permission from the user for access to protected resources.
- A token is issued to the application, if permission is granted by the user.
- The application authenticates using the token to access protected resources.

Issuing Tokens

OAuthorize, a sibling project to Passport, provides a toolkit for implementing OAuth service providers.

The authorization process is a complex sequence that involves authenticating both the requesting application and the user, as well as prompting the user for permission, ensuring that enough detail is provided for the user to make an informed decision.

Additionally, it is up to the implementor to determine what limits can be placed on the application regarding scope of access, as well as subsequently enforcing those limits.

As a toolkit, OAuthorize does not attempt to make implementation decisions. This guide does not cover these issues, but does highly recommend that services deploying OAuth have a complete understanding of the security considerations involved.

Authenticating Tokens

Once issued, OAuth tokens can be authenticated using the passport-http-oauth module.

Install
```
$ npm install passport-http-oauth
```
Configuration
```
passport.use('token', new TokenStrategy(
  function(consumerKey, done) {
    Consumer.findOne({ key: consumerKey }, function (err, consumer) {
      if (err) { return done(err); }
      if (!consumer) { return done(null, false); }
      return done(null, consumer, consumer.secret);
    });
  },
  function(accessToken, done) {
    AccessToken.findOne({ token: accessToken }, function (err, token) {
      if (err) { return done(err); }
      if (!token) { return done(null, false); }
      Users.findById(token.userId, function(err, user) {
        if (err) { return done(err); }
        if (!user) { return done(null, false); }
        // fourth argument is optional info.  typically used to pass
        // details needed to authorize the request (ex: `scope`)
        return done(null, user, token.secret, { scope: token.scope });
      });
    });
  },
  function(timestamp, nonce, done) {
    // validate the timestamp and nonce as necessary
    done(null, true)
  }
));
```

In contrast to other strategies, there are two callbacks required by OAuth. In OAuth, both an identifier for the requesting application and the user-specific token are encoded as credentials. The first callback is known as the "consumer callback", and is used to find the application making the request, including the secret assigned to it. The second callback is the "token callback", which is used to indentify the user as well as the token's corresponding secret. The secrets supplied by the consumer and token callbacks are used to compute a signature, and authentication fails if it does not match the request signature.

A final "validate callback" is optional, which can be used to prevent replay attacks by checking the timestamp and nonce used in the request.

Protect Endpoints

```
app.get('/api/me',
  passport.authenticate('token', { session: false }),
  function(req, res) {
    res.json(req.user);
  });
```

Specify passport.authenticate() with the token strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

OAuth 2.0

OAuth 2.0 (formally specified by RFC 6749) provides an authorization framework which allows users to authorize access to third-party applications. When authorized, the application is issued a token to use as an authentication credential. This has two primary security benefits:

- The application does not need to store the user's username and password.
- The token can have a restricted scope (for example: read-only access).

These benefits are particularly important for ensuring the security of web applications, making OAuth 2.0 the predominant standard for API authentication.

When using OAuth 2.0 to protect API endpoints, there are three distinct steps that must be performed:

- The application requests permission from the user for access to protected resources.
- A token is issued to the application, if permission is granted by the user.
- The application authenticates using the token to access protected resources.

Issuing Tokens

OAuth2orize, a sibling project to Passport, provides a toolkit for implementing OAuth 2.0 authorization servers.

The authorization process is a complex sequence that involves authenticating both the requesting application and the user, as well as prompting the user for permission, ensuring that enough detail is provided for the user to make an informed decision.

Additionally, it is up to the implementor to determine what limits can be placed on the application regarding scope of access, as well as subsequently enforcing those limits.

As a toolkit, OAuth2orize does not attempt to make implementation decisions. This guide does not cover these issues, but does highly recommend that services deploying OAuth 2.0 have a complete understanding of the security considerations involved.

Authenticating Tokens

OAuth 2.0 provides a framework, in which an arbitrarily extensible set of token types can be issued. In practice, only specific token types have gained widespread use.

Bearer Tokens

Bearer tokens are the most widely issued type of token in OAuth 2.0. So much so, in fact, that many implementations assume that bearer tokens are the only type of token issued.

Bearer tokens can be authenticated using the passport-http-bearer module.

Install

```
$ npm install passport-http-bearer
```

Configuration

```
passport.use(new BearerStrategy(
  function(token, done) {
    User.findOne({ token: token }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      return done(null, user, { scope: 'read' });
    });
  }
));
```

The verify callback for bearer tokens accepts the token as an argument. When invoking done, optional info can be passed, which will be set by Passport at req.authInfo. This is typically used to convey the scope of the token, and can be used when making access control checks.

Protect Endpoints

```
app.get('/api/me',
  passport.authenticate('bearer', { session: false }),
  function(req, res) {
    res.json(req.user);
  });
```
Specify passport.authenticate() with the bearer strategy to protect API endpoints. Sessions are not typically needed by APIs, so they can be disabled.

API Schemes

The following is a list of strategies that implement authentication schemes used when protecting API endpoints.

Scheme	Specification	Developer
Anonymous	N/A	Jared Hanson
Bearer	RFC 6750	Jared Hanson
Basic	RFC 2617	Jared Hanson
Digest	RFC 2617	Jared Hanson
Hash	N/A	Yuri Karadzhov
Hawk	hueniverse/hawk	José F. Romaniello
Local API Key	N/A	Sudhakar Mani
OAuth	RFC 5849	Jared Hanson
OAuth 2.0 Client Password	RFC 6749	Jared Hanson
OAuth 2.0 JWT Client Assertion	draft-jones-oauth-jwt-bearer	xTuple
OAuth 2.0 Public Client	RFC 6749	Tim Shadel

Log In

Passport exposes a login() function on req (also aliased as logIn()) that can be used to establish a login session.

```
req.login(user, function(err) {
  if (err) { return next(err); }
  return res.redirect('/users/' + req.user.username);
});
```
When the login operation completes, user will be assigned to req.user.

Note: passport.authenticate() middleware invokes req.login() automatically. This function is primarily used when users sign up, during which req.login() can be invoked to automatically log in the newly registered user.

Log Out

Passport exposes a logout() function on req (also aliased as logOut()) that can be called from any route handler which needs to terminate a login session. Invoking logout() will remove the req.userproperty and clear the login session (if any).

```
app.get('/logout', function(req, res){
  req.logout();
  res.redirect('/');
});
```
Authorize

An application may need to incorporate information from multiple third-party services. In this case, the application will request the user to "connect", for example, both their Facebook and Twitter accounts.

When this occurs, a user will already be authenticated with the application, and any subsequent third-party accounts merely need to be authorized and associated with the user. Because

authentication and authorization in this situation are similar, Passport provides a means to accommodate both.
Authorization is performed by calling passport.authorize(). If authorization is granted, the result provided by the strategy's verify callback will be assigned to req.account. The existing login session and req.userwill be unaffected.

```
app.get('/connect/twitter',
  passport.authorize('twitter-authz', { failureRedirect: '/account' })
);

app.get('/connect/twitter/callback',
  passport.authorize('twitter-authz', { failureRedirect: '/account' }),
  function(req, res) {
    var user = req.user;
    var account = req.account;

    // Associate the Twitter account with the logged-in user.
    account.userId = user.id;
    account.save(function(err) {
      if (err) { return self.error(err); }
      self.redirect('/');
    });
  }
);
```

In the callback route, you can see the use of both req.user and req.account. The newly connected account is associated with the logged-in user and saved to the database.
Configuration
Strategies used for authorization are the same as those used for authentication. However, an application may want to offer both authentication and authorization with the same third-party service. In this case, a named strategy can be used, by overriding the strategy's default name in the call to use().

```
passport.use('twitter-authz', new TwitterStrategy({
    consumerKey: TWITTER_CONSUMER_KEY,
    consumerSecret: TWITTER_CONSUMER_SECRET,
    callbackURL: "http://www.example.com/connect/twitter/callback"
  },
  function(token, tokenSecret, profile, done) {
    Account.findOne({ domain: 'twitter.com', uid: profile.id }, function(err, account) {
      if (err) { return done(err); }
      if (account) { return done(null, account); }

      var account = new Account();
      account.domain = 'twitter.com';
      account.uid = profile.id;
      var t = { kind: 'oauth', token: token, attributes: { tokenSecret: tokenSecret } };
      account.tokens.push(t);
      return done(null, account);
    });
  }
));
```
In the above example, you can see that the twitter-authz strategy is finding or creating an Accountinstance to store Twitter account information. The result will be assigned to req.account, allowing the route handler to associate the account with the authenticated user.

Association in Verify Callback

One downside to the approach described above is that it requires two instances of the same strategy and supporting routes.

To avoid this, set the strategy's passReqToCallback option to true. With this option enabled, req will be passed as the first argument to the verify callback.
```
passport.use(new TwitterStrategy({
    consumerKey: TWITTER_CONSUMER_KEY,
    consumerSecret: TWITTER_CONSUMER_SECRET,
    callbackURL: "http://www.example.com/auth/twitter/callback",
    passReqToCallback: true
  },
  function(req, token, tokenSecret, profile, done) {
    if (!req.user) {
      // Not logged-in. Authenticate based on Twitter account.
    } else {
      // Logged in. Associate Twitter account with user.  Preserve the login
      // state by supplying the existing user after association.
      // return done(null, req.user);
    }
  }
));
```
With req passed as an argument, the verify callback can use the state of the request to tailor the authentication process, handling both authentication and authorization using a single strategy instance and set of routes. For example, if a user is already logged in, the newly "connected" account can be associated. Any additional application-specific properties set on req, including req.session, can be used as well.

1.4) Socket.IO

Introduction

Writing a chat application with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it’s a lot slower than it should be.

Sockets have traditionally been the solution around which most real-time chat systems are architected, providing a bi-directional communication channel between a client and a server.

This means that the server can push messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients.

The web framework

The first goal is to setup a simple HTML webpage that serves out a form and a list of messages. We’re going to use the Node.JS web framework express to this end. Make sure Node.JS is installed.

First let’s create a package.json manifest file that describes our project. I recommend you place it in a dedicated empty directory (I’ll call mine chat-example).

```
{
  "name": "socket-chat-example",
  "version": "0.0.1",
  "description": "my first socket.io app",
  "dependencies": {}
}
```

Now, in order to easily populate the dependencies with the things we need, we’ll use npm install --save:

```
npm install --save express@4.15.2
```

Now that express is installed we can create an index.js file that will setup our application.

```
var app = require('express')();
var http = require('http').createServer(app);

app.get('/', function(req, res){
  res.send('<h1>Hello world</h1>');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

This translates into the following:

- Express initializes app to be a function handler that you can supply to an HTTP server (as seen in line 2).
- We define a route handler / that gets called when we hit our website home.
- We make the http server listen on port 3000.

If you run node index.js you should see the following:

And if you point your browser to http://localhost:3000:

Serving HTML

So far in index.js we’re calling res.send and pass it a HTML string. Our code would look very confusing if we just placed our entire application’s HTML there. Instead, we’re going to create a index.html file and serve it.

Let’s refactor our route handler to use sendFile instead:

```
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
```
And populate index.html with the following:
```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <style>
      * { margin: 0; padding: 0; box-sizing: border-box; }
      body { font: 13px Helvetica, Arial; }
      form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }
      form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }
      form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }
      #messages { list-style-type: none; margin: 0; padding: 0; }
      #messages li { padding: 5px 10px; }
      #messages li:nth-child(odd) { background: #eee; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form action="">
      <input id="m" autocomplete="off" /><button>Send</button>
    </form>
  </body>
</html>
```
If you restart the process (by hitting Control+C and running node index again) and refresh the page it should look like this:

Integrating Socket.IO

Socket.IO is composed of two parts:

- A server that integrates with (or mounts on) the Node.JS HTTP Server: socket.io
- A client library that loads on the browser side: socket.io-client

During development, socket.io serves the client automatically for us, as we'll see, so for now we only have to install one module:
```
npm install --save socket.io
```
That will install the module and add the dependency to package.json. Now let's edit index.js to add it:
```
var app = require('express')();
var http = require('http').createServer(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function(socket){
  console.log('a user connected');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Notice that I initialize a new instance of socket.io by passing the http (the HTTP server) object.
Then I listen on the connection event for incoming sockets, and I log it to the console.
Now in index.html I add the following snippet before the </body>:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();
</script>
```

That’s all it takes to load the socket.io-client, which exposes a io global (and the endpoint GET /socket.io/socket.io.js), and then connect.
If you would like to use the local version of the client-side JS file, you can find it at node_modules/socket.io-client/dist/socket.io.js.
Notice that I’m not specifying any URL when I call io(), since it defaults to trying to connect to the host that serves the page.
If you now reload the server and the website you should see the console print “a user connected”.
Try opening several tabs, and you’ll see several messages:

Each socket also fires a special disconnect event:

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

Then if you refresh a tab several times you can see it in action:

Emitting events
The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.
Let’s make it so that when the user types in a message, the server gets it as a chat message event. The script section in index.html should now look as follows:

```
<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
```

```
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(e){
      e.preventDefault(); // prevents page reloading
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
  });
</script>
```
And in index.js we print out the chat message event:
```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    console.log('message: ' + msg);
  });
});
```
The result should be like the following video:

Broadcasting

The next goal is for us to emit the event from the server to the rest of the users.

In order to send an event to everyone, Socket.IO gives us the io.emit:
```
io.emit('some event', { for: 'everyone' });
```
If you want to send a message to everyone except for a certain socket, we have the broadcast flag:
```
io.on('connection', function(socket){
  socket.broadcast.emit('hi');
});
```
In this case, for the sake of simplicity we'll send the message to everyone, including the sender.
```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```
And on the client side when we capture a chat message event we'll include it in the page. The total client-side JavaScript code now amounts to:
```
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(e){
      e.preventDefault(); // prevents page reloading
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
    socket.on('chat message', function(msg){
      $('#messages').append($('<li>').text(msg));
    });
  });
</script>
```

1.5)Validator
A library of string validators and sanitizers.
Strings only
This library validates and sanitizes strings only.
If you're not sure if your input is a string, coerce it using input + ''. Passing anything other than a string is an error.
Installation and Usage
Server-side usage
Install the library with npm install validator
No ES6

```
var validator = require('validator');

validator.isEmail('foo@bar.com'); //=> true
```

ES6

```
import validator from 'validator';
```

Or, import only a subset of the library:

```
import isEmail from 'validator/lib/isEmail';
```

Client-side usage
The library can be loaded either as a standalone script, or through an AMD-compatible loader

```
<script type="text/javascript" src="validator.min.js"></script>
<script type="text/javascript">
  validator.isEmail('foo@bar.com'); //=> true
</script>
```

The library can also be installed through bower

```
$ bower install validator-js
```

Validators
Here is a list of the validators currently available.

Validator	Description
contains(str, seed)	check if the string contains the seed.
equals(str, comparison)	check if the string matches the comparison.
isAfter(str [, date])	check if the string is a date that's after the specified date (defaults to now).
isAlpha(str [, locale])	check if the string contains only letters (a-zA-Z). Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK',

'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphaLocales.
isAlphanumeric(str [, locale]) check if the string contains only letters and numbers.

Locale is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'el-GR', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']) and defaults to en-US. Locale list is validator.isAlphanumericLocales.
isAscii(str)   check if the string contains ASCII chars only.
isBase32(str) check if a string is base32 encoded.
isBase64(str) check if a string is base64 encoded.
isBefore(str [, date])   check if the string is a date that's before the specified date.
isBoolean(str) check if a string is a boolean.
isByteLength(str [, options])   check if the string's length (in UTF-8 bytes) falls in a range.

options is an object which defaults to {min:0, max: undefined}.
isCreditCard(str)   check if the string is a credit card.
isCurrency(str [, options])   check if the string is a valid currency amount.

options is an object which defaults to {symbol: '$', require_symbol: false, allow_space_after_symbol: false, symbol_after_digits: false, allow_negatives: true, parens_for_negatives: false, negative_sign_before_digits: false, negative_sign_after_digits: false, allow_negative_sign_placeholder: false, thousands_separator: ',', decimal_separator: '.', allow_decimal: true, require_decimal: false, digits_after_decimal: [2], allow_space_after_digits: false}.
Note: The array digits_after_decimal is filled with the exact number of digits allowed not a range, for example a range 1 to 3 will be given as [1, 2, 3].
isDataURI(str) check if the string is a data uri format.
isMagnetURI(str)   check if the string is a magnet uri format.
isDecimal(str [, options])   check if the string represents a decimal number, such as 0.1, .3, 1.1, 1.00003, 4.0, etc.

options is an object which defaults to {force_decimal: false, decimal_digits: '1,', locale: 'en-US'}

locale determine the decimal separator and is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'ku-IQ', nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA'].

Note: decimal_digits is given as a range like '1,3', a specific value like '3' or min like '1,'.
isDivisibleBy(str, number)	check if the string is a number that's divisible by another.
isEmail(str [, options]) check if the string is an email.

options is an object which defaults to { allow_display_name: false, require_display_name: false, allow_utf8_local_part: true, require_tld: true, allow_ip_domain: false, domain_specific_validation: false }. If allow_display_name is set to true, the validator will also match Display Name <email-address>. If require_display_name is set to true, the validator will reject strings without the format Display Name <email-address>. If allow_utf8_local_part is set to false, the validator will not allow any non-English UTF8 character in email address' local part. If require_tld is set to false, e-mail addresses without having TLD in their domain will also be matched. If ignore_max_length is set to true, the validator will not check for the standard max length of an email. If allow_ip_domain is set to true, the validator will allow IP addresses in the host part. If domain_specific_validation is true, some additional validation will be enabled, e.g. disallowing certain syntactically valid email addresses that are rejected by GMail.
isEmpty(str [, options])	check if the string has a length of zero.

options is an object which defaults to { ignore_whitespace:false }.
isFQDN(str [, options])	check if the string is a fully qualified domain name (e.g. domain.com).

options is an object which defaults to { require_tld: true, allow_underscores: false, allow_trailing_dot: false }.
isFloat(str [, options]) check if the string is a float.

options is an object which can contain the keys min, max, gt, and/or lt to validate the float is within boundaries (e.g. { min: 7.22, max: 9.55 }) it also has locale as an option.

min and max are equivalent to 'greater or equal' and 'less or equal', respectively while gt and lt are their strict counterparts.

locale determine the decimal separator and is one of ['ar', 'ar-AE', 'ar-BH', 'ar-DZ', 'ar-EG', 'ar-IQ', 'ar-JO', 'ar-KW', 'ar-LB', 'ar-LY', 'ar-MA', 'ar-QA', 'ar-QM', 'ar-SA', 'ar-SD', 'ar-SY', 'ar-TN', 'ar-YE', 'bg-BG', 'cs-CZ', 'da-DK', 'de-DE', 'en-AU', 'en-GB', 'en-HK', 'en-IN', 'en-NZ', 'en-US', 'en-ZA', 'en-ZM', 'es-ES', 'fr-FR', 'hu-HU', 'it-IT', 'nb-NO', 'nl-NL', 'nn-NO', 'pl-PL', 'pt-BR', 'pt-PT', 'ru-RU', 'sl-SI', 'sr-RS', 'sr-RS@latin', 'sv-SE', 'tr-TR', 'uk-UA']. Locale list is validator.isFloatLocales.
isFullWidth(str)	check if the string contains any full-width chars.
isHalfWidth(str)	check if the string contains any half-width chars.
isHash(str, algorithm) check if the string is a hash of type algorithm.

Algorithm is one of ['md4', 'md5', 'sha1', 'sha256', 'sha384', 'sha512', 'ripemd128', 'ripemd160', 'tiger128', 'tiger160', 'tiger192', 'crc32', 'crc32b']

isHexColor(str)	check if the string is a hexadecimal color.
isHexadecimal(str)	check if the string is a hexadecimal number.
isIdentityCard(str [, locale])	check if the string is a valid identity card code.

locale is one of ['ES'] OR 'any'. If 'any' is used, function will check if any of the locals match.

Defaults to 'any'.
isIP(str [, version])	check if the string is an IP (version 4 or 6).
isIPRange(str)	check if the string is an IP Range(version 4 only).
isISBN(str [, version])	check if the string is an ISBN (version 10 or 13).
isISSN(str [, options])	check if the string is an ISSN.

options is an object which defaults to { case_sensitive: false, require_hyphen: false }. If case_sensitiveis true, ISSNs with a lowercase 'x' as the check digit are rejected.
isISIN(str)	check if the string is an ISIN (stock/security identifier).
isISO8601(str)	check if the string is a valid ISO 8601 date; for additional checks for valid dates, e.g. invalidates dates like 2009-02-29, pass options object as a second parameter with options.strict = true.
isRFC3339(str)	check if the string is a valid RFC 3339 date.
isISO31661Alpha2(str)	check if the string is a valid ISO 3166-1 alpha-2 officially assigned country code.
isISO31661Alpha3(str)	check if the string is a valid ISO 3166-1 alpha-3 officially assigned country code.
isISRC(str)	check if the string is a ISRC.
isIn(str, values)	check if the string is in a array of allowed values.
isInt(str [, options])	check if the string is an integer.

options is an object which can contain the keys min and/or max to check the integer is within boundaries (e.g. { min: 10, max: 99 }). options can also contain the key allow_leading_zeroes, which when set to false will disallow integer values with leading zeroes (e.g. { allow_leading_zeroes: false }). Finally, options can contain the keys gt and/or lt which will enforce integers being greater than or less than, respectively, the value provided (e.g. {gt: 1, lt: 4} for a number between 1 and 4).
isJSON(str)	check if the string is valid JSON (note: uses JSON.parse).
isJWT(str)	check if the string is valid JWT token.
isLatLong(str)	check if the string is a valid latitude-longitude coordinate in the format lat,long or lat, long.
isLength(str [, options])	check if the string's length falls in a range.

options is an object which defaults to {min:0, max: undefined}. Note: this function takes into account surrogate pairs.
isLowercase(str)	check if the string is lowercase.
isMACAddress(str)	check if the string is a MAC address.

options is an object which defaults to {no_colons: false}. If no_colons is true, the validator will allow MAC addresses without the colons.
isMD5(str)	check if the string is a MD5 hash.
isMimeType(str)	check if the string matches to a valid MIME type format
isMobilePhone(str [, locale [, options]])	check if the string is a mobile phone number,

(locale is either an array of locales (e.g ['sk-SK', 'sr-RS']) OR one of ['ar-AE', 'ar-DZ', 'ar-EG', 'ar-IQ', ar-JO', 'ar-KW', 'ar-SA', 'ar-SY', 'ar-TN', 'be-BY', 'bg-BG', 'bn-BD', 'cs-CZ', 'de-DE', 'da-DK', 'el-GR', 'en-AU', 'en-CA', 'en-GB', 'en-GH', 'en-HK', 'en-IE', 'en-IN', 'en-KE', 'en-MU', en-NG', 'en-NZ', 'en-RW', 'en-SG', 'en-UG', 'en-US', 'en-TZ', 'en-ZA', 'en-ZM', 'en-PK', 'es-ES', 'es-MX', 'es-PY', 'es-UY', 'et-EE', 'fa-IR', 'fi-FI', 'fr-FR', 'he-IL', 'hu-HU', 'id-ID', 'it-IT', 'ja-JP', 'kk-KZ', 'ko-KR', 'lt-LT', 'ms-MY', 'nb-NO', 'nn-NO', 'pl-PL', 'pt-PT', 'pt-BR', 'ro-RO', 'ru-RU', 'sl-SI', 'sk-SK', 'sr-RS', 'sv-SE', 'th-TH', 'tr-TR', 'uk-UA', 'vi-VN', 'zh-CN', 'zh-HK', 'zh-TW'] OR defaults to 'any'. If 'any' or a falsey value is used, function will check if any of the locales match).

options is an optional object that can be supplied with the following keys: strictMode, if this is set to true, the mobile phone number must be supplied with the country code and therefore must start with +. Locale list is validator.isMobilePhoneLocales.
isMongoId(str)	check if the string is a valid hex-encoded representation of a MongoDB ObjectId.
isMultibyte(str)	check if the string contains one or more multibyte chars.
isNumeric(str [, options])	check if the string contains only numbers.

options is an object which defaults to {no_symbols: false}. If no_symbols is true, the validator will reject numeric strings that feature a symbol (e.g. +, -, or .).
isPort(str)	check if the string is a valid port number.
isPostalCode(str, locale)	check if the string is a postal code,

(locale is one of ['AD', 'AT', 'AU', 'BE', 'BG', 'CA', 'CH', 'CZ', 'DE', 'DK', 'DZ', 'EE', 'ES', 'FI', 'FR', 'GB', 'GR', 'HR', 'HU', 'ID', 'IL', 'IN', 'IS', 'IT', 'JP', 'KE', 'LI', 'LT', 'LU', 'LV', 'MX', 'NL', 'NO', 'PL', 'PT', 'RO', 'RU', 'SA', 'SE', 'SI', 'TN', 'TW', 'UA', 'US', 'ZA', 'ZM'] OR 'any'. If 'any' is used, function will check if any of the locals match. Locale list is validator.isPostalCodeLocales.).
isSurrogatePair(str)	check if the string contains any surrogate pairs chars.
isURL(str [, options])	check if the string is an URL.

options is an object which defaults to { protocols: ['http','https','ftp'], require_tld: true, require_protocol: false, require_host: true, require_valid_protocol: true, allow_underscores: false, host_whitelist: false, host_blacklist: false, allow_trailing_dot: false, allow_protocol_relative_urls: false, disallow_auth: false }.
isUUID(str [, version])	check if the string is a UUID (version 3, 4 or 5).
isUppercase(str)	check if the string is uppercase.
isVariableWidth(str)	check if the string contains a mixture of full and half-width chars.
isWhitelisted(str, chars)	checks characters if they appear in the whitelist.

matches(str, pattern [, modifiers])	check if string matches the pattern. Either matches('foo', /foo/i) or matches('foo', 'foo', 'i').

Sanitizers

Here is a list of the sanitizers currently available.

Sanitizer	Description
blacklist(input, chars)	remove characters that appear in the blacklist. The characters are used in a RegExp and so you will need to escape some chars, e.g. blacklist(input, '\\[\\]').
escape(input)	replace <, >, &, ', " and / with HTML entities.
unescape(input)	replaces HTML encoded entities with <, >, &, ', " and /.
ltrim(input [, chars])	trim characters from the left-side of the input.
normalizeEmail(email [, options])	canonicalizes an email address. (This doesn't validate that the input is an email, if you want to validate the email use isEmail beforehand) options is an object with the following keys and default values:

 all_lowercase: true - Transforms the local part (before the @ symbol) of all email addresses to lowercase. Please note that this may violate RFC 5321, which gives providers the possibility to treat the local part of email addresses in a case sensitive way (although in practice most - yet not all - providers don't). The domain part of the email address is always lowercased, as it's case insensitive per RFC 1035.

 gmail_lowercase: true - GMail addresses are known to be case-insensitive, so this switch allows lowercasing them even when all_lowercase is set to false. Please note that when all_lowercase is true, GMail addresses are lowercased regardless of the value of this setting.

 gmail_remove_dots: true: Removes dots from the local part of the email address, as GMail ignores them (e.g. "john.doe" and "johndoe" are considered equal).

 gmail_remove_subaddress: true: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. "foo+bar@gmail.com" becomes "foo@gmail.com").

 gmail_convert_googlemaildotcom: true: Converts addresses with domain @googlemail.com to @gmail.com, as they're equivalent.

 outlookdotcom_lowercase: true - Outlook.com addresses (including Windows Live and Hotmail) are known to be case-insensitive, so this switch allows lowercasing them even when all_lowercase is set to false. Please note that when all_lowercase is true, Outlook.com addresses are lowercased regardless of the value of this setting.

 outlookdotcom_remove_subaddress: true: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. "foo+bar@outlook.com" becomes "foo@outlook.com").

 yahoo_lowercase: true - Yahoo Mail addresses are known to be case-insensitive, so this switch allows lowercasing them even when all_lowercase is set to false. Please note that when all_lowercase is true, Yahoo Mail addresses are lowercased regardless of the value of this setting.

	yahoo_remove_subaddress: true: Normalizes addresses by removing "sub-addresses", which is the part following a "-" sign (e.g. "foo-bar@yahoo.com" becomes "foo@yahoo.com").
	icloud_lowercase: true - iCloud addresses (including MobileMe) are known to be case-insensitive, so this switch allows lowercasing them even when all_lowercase is set to false. Please note that when all_lowercase is true, iCloud addresses are lowercased regardless of the value of this setting.
	icloud_remove_subaddress: true: Normalizes addresses by removing "sub-addresses", which is the part following a "+" sign (e.g. "foo+bar@icloud.com" becomes "foo@icloud.com").
rtrim(input [, chars])   trim characters from the right-side of the input.
stripLow(input [, keep_new_lines])   remove characters with a numerical value < 32 and 127, mostly control characters. If keep_new_lines is true, newline characters are preserved (\n and \r, hex 0xA and 0xD). Unicode-safe in JavaScript.
toBoolean(input [, strict])   convert the input string to a boolean. Everything except for '0', 'false' and '' returns true. In strict mode only '1' and 'true' return true.
toDate(input)   convert the input string to a date, or null if the input is not a date.
toFloat(input)   convert the input string to a float, or NaN if the input is not a float.
toInt(input [, radix])   convert the input string to an integer, or NaN if the input is not an integer.
trim(input [, chars])   trim characters (whitespace by default) from both sides of the input.
whitelist(input, chars)   remove characters that do not appear in the whitelist. The characters are used in a RegExp and so you will need to escape some chars, e.g. whitelist(input, '\\[\\]').

1.6) Multer:
Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency.
NOTE: Multer will not process any form which is not multipart (multipart/form-data).
Translations
This README is also available in other languages:
简体中文 (Chinese)
한국어 (Korean)
Installation
$ npm install --save multer
Usage
Multer adds a body object and a file or files object to the request object. The bodyobject contains the values of the text fields of the form, the file or files object contains the files uploaded via the form.
Basic usage example:
Don't forget the enctype="multipart/form-data" in your form.
```
<form action="/profile" method="post" enctype="multipart/form-data">
  <input type="file" name="avatar" />
</form>
var express = require('express')
var multer  = require('multer')
var upload = multer({ dest: 'uploads/' })

var app = express()

app.post('/profile', upload.single('avatar'), function (req, res, next) {
  // req.file is the `avatar` file
  // req.body will hold the text fields, if there were any
})

app.post('/photos/upload', upload.array('photos', 12), function (req, res, next) {
  // req.files is array of `photos` files
  // req.body will contain the text fields, if there were any
})

var cpUpload = upload.fields([{ name: 'avatar', maxCount: 1 }, { name: 'gallery', maxCount: 8 }])
app.post('/cool-profile', cpUpload, function (req, res, next) {
  // req.files is an object (String -> Array) where fieldname is the key, and the value is array of
files
  //
  // e.g.
  //  req.files['avatar'][0] -> File
  //  req.files['gallery'] -> Array
  //
  // req.body will contain the text fields, if there were any
})
```

In case you need to handle a text-only multipart form, you should use the .none() method:

```
var express = require('express')
var app = express()
var multer  = require('multer')
var upload = multer()

app.post('/profile', upload.none(), function (req, res, next) {
  // req.body contains the text fields
})
```

API

File information

Each file contains the following information:

Key	Description	Note
fieldname	Field name specified in the form	
originalname	Name of the file on the user's computer	
encoding	Encoding type of the file	
mimetype	Mime type of the file	

size	Size of the file in bytes	
destination	The folder to which the file has been saved	DiskStorage
filename	The name of the file within the destination	DiskStorage
path	The full path to the uploaded file	DiskStorage
buffer	A Buffer of the entire file	MemoryStorage

multer(opts)
Multer accepts an options object, the most basic of which is the dest property, which tells Multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to disk.
By default, Multer will rename the files so as to avoid naming conflicts. The renaming function can be customized according to your needs.
The following are the options that can be passed to Multer.

Key	Description
dest or storage	Where to store the files
fileFilter	Function to control which files are accepted
limits	Limits of the uploaded data
preservePath	Keep the full path of files instead of just the base name

In an average web app, only dest might be required, and configured as shown in the following example.

```
var upload = multer({ dest: 'uploads/' })
```

If you want more control over your uploads, you'll want to use the storage option instead of dest. Multer ships with storage engines DiskStorage and MemoryStorage; More engines are available from third parties.
.single(fieldname)
Accept a single file with the name fieldname. The single file will be stored in req.file.
.array(fieldname[, maxCount])
Accept an array of files, all with the name fieldname. Optionally error out if more than maxCount files are uploaded. The array of files will be stored in req.files.
.fields(fields)
Accept a mix of files, specified by fields. An object with arrays of files will be stored in req.files.
fields should be an array of objects with name and optionally a maxCount. Example:

```
[
  { name: 'avatar', maxCount: 1 },
  { name: 'gallery', maxCount: 8 }
]
```

.none()
Accept only text fields. If any file upload is made, error with code "LIMIT_UNEXPECTED_FILE" will be issued.
.any()
Accepts all files that comes over the wire. An array of files will be stored in req.files.
WARNING: Make sure that you always handle the files that a user uploads. Never add multer as a global middleware since a malicious user could upload files to a route that you didn't anticipate. Only use this function on routes where you are handling the uploaded files.

storage

DiskStorage

The disk storage engine gives you full control on storing files to disk.

```
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, '/tmp/my-uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})

var upload = multer({ storage: storage })
```

There are two options available, destination and filename. They are both functions that determine where the file should be stored.

destination is used to determine within which folder the uploaded files should be stored. This can also be given as a string (e.g. '/tmp/uploads'). If no destination is given, the operating system's default directory for temporary files is used.

Note: You are responsible for creating the directory when providing destination as a function. When passing a string, multer will make sure that the directory is created for you.

filename is used to determine what the file should be named inside the folder. If no filename is given, each file will be given a random name that doesn't include any file extension.

Note: Multer will not append any file extension for you, your function should return a filename complete with an file extension.

Each function gets passed both the request (req) and some information about the file (file) to aid with the decision.

Note that req.body might not have been fully populated yet. It depends on the order that the client transmits fields and files to the server.

MemoryStorage

The memory storage engine stores the files in memory as Buffer objects. It doesn't have any options.

```
var storage = multer.memoryStorage()
var upload = multer({ storage: storage })
```

When using memory storage, the file info will contain a field called buffer that contains the entire file.

WARNING: Uploading very large files, or relatively small files in large numbers very quickly, can cause your application to run out of memory when memory storage is used.

limits

An object specifying the size limits of the following optional properties. Multer passes this object into busboy directly, and the details of the properties can be found on busboy's page.

The following integer values are available:

Key	Description	Default
fieldNameSize	Max field name size	100 bytes

fieldSize	Max field value size	1MB
fields	Max number of non-file fields	Infinity
fileSize	For multipart forms, the max file size (in bytes)	Infinity
files	For multipart forms, the max number of file fields	Infinity
parts	For multipart forms, the max number of parts (fields + files)	Infinity
headerPairs	For multipart forms, the max number of header key=>value pairs to parse	2000

Specifying the limits can help protect your site against denial of service (DoS) attacks.

fileFilter

Set this to a function to control which files should be uploaded and which should be skipped.
The function should look like this:

```
function fileFilter (req, file, cb) {

  // The function should call `cb` with a boolean
  // to indicate if the file should be accepted

  // To reject this file pass `false`, like so:
  cb(null, false)

  // To accept the file pass `true`, like so:
  cb(null, true)

  // You can always pass an error if something goes wrong:
  cb(new Error('I don\'t have a clue!'))

}
```

Error handling

When encountering an error, Multer will delegate the error to Express. You can display a nice error page using the standard express way.

If you want to catch errors specifically from Multer, you can call the middleware function by yourself. Also, if you want to catch only the Multer errors, you can use the MulterError class that is attached to the multer object itself (e.g. err instanceof multer.MulterError).

```
var multer = require('multer')
var upload = multer().single('avatar')

app.post('/profile', function (req, res) {
  upload(req, res, function (err) {
    if (err instanceof multer.MulterError) {
      // A Multer error occurred when uploading.
    } else if (err) {
      // An unknown error occurred when uploading.
    }

    // Everything went fine.
  })
})
```

1.7) Stripe:

Usage

The package needs to be configured with your account's secret key which is available in your Stripe Dashboard. Require it with the key's value:

```
const stripe = require('stripe')('sk_test_...');

const customer = await stripe.customers.create({
  email: 'customer@example.com',
});
```

Or using ES modules, this looks more like:

```
import Stripe from 'stripe';
const stripe = Stripe('sk_test_...');
//…
```

On older versions of Node, you can use promises or callbacks instead of async/await.

Usage with TypeScript

Stripe does not currently maintain typings for this package, but there are community typings available from DefinitelyTyped.

To install:

```
npm install --dev @types/stripe
```

To use:

```
// Note `* as` and `new Stripe` for TypeScript:
import * as Stripe from 'stripe';
const stripe = new Stripe('sk_test_...');

const customer: Promise<
  Stripe.customers.ICustomer
> = stripe.customers.create(/* ... */);
```

Using Promises

Every method returns a chainable promise which can be used instead of a regular callback:

```
// Create a new customer and then a new charge for that customer:
stripe.customers
  .create({
    email: 'foo-customer@example.com',
  })
  .then((customer) => {
    return stripe.customers.createSource(customer.id, {
      source: 'tok_visa',
    });
  })
  .then((source) => {
    return stripe.charges.create({
      amount: 1600,
      currency: 'usd',
      customer: source.customer,
    });
  })
  .then((charge) => {
    // New charge created on a new customer
  })
  .catch((err) => {
    // Deal with an error
  });
```

Using callbacks

On versions of Node.js prior to v7.9:

```
var stripe = require('stripe')('sk_test_...');

stripe.customers.create(
  {
    email: 'customer@example.com',
  },
  function(err, customer) {
    if (err) {
      // Deal with an error (will be `null` if no error occurred).
    }

    // Do something with created customer object
    console.log(customer.id);
  }
);
```

Configuring Timeout

Request timeout is configurable (the default is Node's default of 120 seconds):

```
stripe.setTimeout(20000); // in ms (this is 20 seconds)
```

Configuring For Connect

A per-request Stripe-Account header for use with Stripe Connect can be added to any method:

```
// Retrieve the balance for a connected account:
stripe.balance
  .retrieve({
    stripe_account: 'acct_foo',
  })
  .then((balance) => {
    // The balance object for the connected account
  })
  .catch((err) => {
    // Error
  });
```
Configuring a Proxy
An https-proxy-agent can be configured with setHttpAgent.
To use stripe behind a proxy you can pass to sdk:
```
if (process.env.http_proxy) {
  const ProxyAgent = require('https-proxy-agent');
  stripe.setHttpAgent(new ProxyAgent(process.env.http_proxy));
}
```
Network retries
Automatic network retries can be enabled with setMaxNetworkRetries. This will retry requests n times with exponential backoff if they fail due to an intermittent network problem.Idempotency keys are added where appropriate to prevent duplication.
```
// Retry a request once before giving up
stripe.setMaxNetworkRetries(1);
```
Examining Responses
Some information about the response which generated a resource is available with the lastResponse property:
```
charge.lastResponse.requestId; // see: https://stripe.com/docs/api/node#request_ids
charge.lastResponse.statusCode;
```
request and response events
The Stripe object emits request and response events. You can use them like this:
```
const stripe = require('stripe')('sk_test_...');

const onRequest = (request) => {
  // Do something.
};

// Add the event handler function:
stripe.on('request', onRequest);

// Remove the event handler function:
stripe.off('request', onRequest);
```
request object
```
{
  api_version: 'latest',
  account: 'acct_TEST',       // Only present if provided
  idempotency_key: 'abc123',  // Only present if provided
  method: 'POST',
  path: '/v1/charges'
}
```
response object
```
{
  api_version: 'latest',
  account: 'acct_TEST',       // Only present if provided
  idempotency_key: 'abc123',  // Only present if provided
  method: 'POST',
  path: '/v1/charges',
  status: 402,
  request_id: 'req_Ghc9r26ts73DRf',
  elapsed: 445                // Elapsed time in milliseconds
}
```

Webhook signing

Stripe can optionally sign the webhook events it sends to your endpoint, allowing you to validate that they were not sent by a third-party. You can read more about it here.

Please note that you must pass the raw request body, exactly as received from Stripe, to the constructEvent() function; this will not work with a parsed (i.e., JSON) request body.

You can find an example of how to use this with Express in the examples/webhook-signingfolder, but here's what it looks like:

```
const event = stripe.webhooks.constructEvent(
  webhookRawBody,
  webhookStripeSignatureHeader,
  webhookSecret
);
```

Testing Webhook signing

You can use stripe.webhooks.generateTestHeaderString to mock webhook events that come from Stripe:

```
const payload = {
  id: 'evt_test_webhook',
  object: 'event',
};

const payloadString = JSON.stringify(payload, null, 2);
const secret = 'whsec_test_secret';

const header = stripe.webhooks.generateTestHeaderString({
  payload: payloadString,
  secret,
});

const event = stripe.webhooks.constructEvent(payloadString, header, secret);

// Do something with mocked signed event
expect(event.id).to.equal(payload.id);
```

Writing a Plugin

If you're writing a plugin that uses the library, we'd appreciate it if you identified using stripe.setAppInfo():

```
stripe.setAppInfo({
  name: 'MyAwesomePlugin',
  version: '1.2.34', // Optional
  url: 'https://myawesomeplugin.info', // Optional
});
```

This information is passed along when the library makes calls to the Stripe API.

Auto-pagination

As of stripe-node 6.11.0, you may auto-paginate list methods. We provide a few different APIs for this to aid with a variety of node versions and styles.

Async iterators (for-await-of)

If you are in a Node environment that has support for async iteration, such as Node 10+ or babel, the following will auto-paginate:

```
for await (const customer of stripe.customers.list()) {
  doSomething(customer);
  if (shouldStop()) {
    break;
  }
}
```

autoPagingEach

If you are in a Node environment that has support for await, such as Node 7.9 and greater, you may pass an async function to .autoPagingEach:

```
await stripe.customers.list().autoPagingEach(async (customer) => {
  await doSomething(customer);
  if (shouldBreak()) {
    return false;
  }
});
console.log('Done iterating.');
```

Equivalently, without await, you may return a Promise, which can resolve to false to break:

```
stripe.customers
  .list()
  .autoPagingEach((customer) => {
    return doSomething(customer).then(() => {
      if (shouldBreak()) {
        return false;
      }
    });
  })
  .then(() => {
    console.log('Done iterating.');
  })
  .catch(handleError);
```

If you prefer callbacks to promises, you may also use a next callback and a second onDonecallback:

```
stripe.customers.list().autoPagingEach(
  function onItem(customer, next) {
    doSomething(customer, function(err, result) {
      if (shouldStop(result)) {
        next(false); // Passing `false` breaks out of the loop.
      } else {
        next();
      }
    });
  },
  function onDone(err) {
    if (err) {
      console.error(err);
    } else {
      console.log('Done iterating.');
    }
  }
);
```

If your onItem function does not accept a next callback parameter or return a Promise, the return value is used to decide whether to continue (false breaks, anything else continues).
autoPagingToArray
This is a convenience for cases where you expect the number of items to be relatively small; accordingly, you must pass a limit option to prevent runaway list growth from consuming too much memory.
Returns a promise of an array of all items across pages for a list request.

```
const allNewCustomers = await stripe.customers
  .list({created: {gt: lastMonth}})
  .autoPagingToArray({limit: 10000});
```

Front-End Packages

2.1) Axios
Installing
Using npm:

```
$ npm install axios
```

Using bower:

```
$ bower install axios
```

Using yarn:

```
$ yarn add axios
```

Using cdn:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Example

Performing a GET request

```
const axios = require('axios');

// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .finally(function () {
    // always executed
  });

// Optionally the request above could also be done as
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  })
  .then(function () {
    // always executed
  });

// Want to use async/await? Add the `async` keyword to your outer function/method.
async function getUser() {
  try {
    const response = await axios.get('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```
NOTE: async/await is part of ECMAScript 2017 and is not supported in Internet Explorer and older browsers, so use with caution.

Performing a POST request
```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```
Performing multiple concurrent requests
```
function getUserAccount() {
  return axios.get('/user/12345');
}

function getUserPermissions() {
  return axios.get('/user/12345/permissions');
}

axios.all([getUserAccount(), getUserPermissions()])
  .then(axios.spread(function (acct, perms) {
    // Both requests are now complete
  }));
```
axios API

Requests can be made by passing the relevant config to axios.

axios(config)
```
// Send a POST request
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
});
// GET request for remote image
axios({
  method: 'get',
  url: 'http://bit.ly/2mTM3nY',
  responseType: 'stream'
})
  .then(function (response) {
    response.data.pipe(fs.createWriteStream('ada_lovelace.jpg'))
  });
axios(url[, config])
// Send a GET request (default method)
axios('/user/12345');
```

Request method aliases

For convenience aliases have been provided for all supported request methods.

```
axios.request(config)
axios.get(url[, config])
axios.delete(url[, config])
axios.head(url[, config])
axios.options(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.patch(url[, data[, config]])
```

NOTE

When using the alias methods url, method, and data properties don't need to be specified in config.

Concurrency

Helper functions for dealing with concurrent requests.

```
axios.all(iterable)
axios.spread(callback)
```

Creating an instance

You can create a new instance of axios with a custom config.

```
axios.create([config])
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,
  headers: {'X-Custom-Header': 'foobar'}
});
```

Instance methods

The available instance methods are listed below. The specified config will be merged with the instance config.

```
axios#request(config)
axios#get(url[, config])
axios#delete(url[, config])
axios#head(url[, config])
axios#options(url[, config])
```

axios#post(url[, data[, config]])
axios#put(url[, data[, config]])
axios#patch(url[, data[, config]])
axios#getUri([config])
Request Config
These are the available config options for making requests. Only the url is required. Requests will default to GET if methodis not specified.

```
{
  // `url` is the server URL that will be used for the request
  url: '/user',

  // `method` is the request method to be used when making the request
  method: 'get', // default

  // `baseURL` will be prepended to `url` unless `url` is absolute.
  // It can be convenient to set `baseURL` for an instance of axios to pass relative URLs
  // to methods of that instance.
  baseURL: 'https://some-domain.com/api/',

  // `transformRequest` allows changes to the request data before it is sent to the server
  // This is only applicable for request methods 'PUT', 'POST', 'PATCH' and 'DELETE'
  // The last function in the array must return a string or an instance of Buffer, ArrayBuffer,
  // FormData or Stream
  // You may modify the headers object.
  transformRequest: [function (data, headers) {
    // Do whatever you want to transform the data

    return data;
  }],

  // `transformResponse` allows changes to the response data to be made before
  // it is passed to then/catch
  transformResponse: [function (data) {
    // Do whatever you want to transform the data

    return data;
  }],

  // `headers` are custom headers to be sent
  headers: {'X-Requested-With': 'XMLHttpRequest'},

  // `params` are the URL parameters to be sent with the request
  // Must be a plain object or a URLSearchParams object
```

```
params: {
  ID: 12345
},

// `paramsSerializer` is an optional function in charge of serializing `params`
// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
paramsSerializer: function (params) {
  return Qs.stringify(params, {arrayFormat: 'brackets'})
},

// `data` is the data to be sent as the request body
// Only applicable for request methods 'PUT', 'POST', and 'PATCH'
// When no `transformRequest` is set, must be of one of the following types:
// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
// - Browser only: FormData, File, Blob
// - Node only: Stream, Buffer
data: {
  firstName: 'Fred'
},

// `timeout` specifies the number of milliseconds before the request times out.
// If the request takes longer than `timeout`, the request will be aborted.
timeout: 1000, // default is `0` (no timeout)

// `withCredentials` indicates whether or not cross-site Access-Control requests
// should be made using credentials
withCredentials: false, // default

// `adapter` allows custom handling of requests which makes testing easier.
// Return a promise and supply a valid response (see lib/adapters/README.md).
adapter: function (config) {
  /* ... */
},

// `auth` indicates that HTTP Basic auth should be used, and supplies credentials.
// This will set an `Authorization` header, overwriting any existing
// `Authorization` custom headers you have set using `headers`.
// Please note that only HTTP Basic auth is configurable through this parameter.
// For Bearer tokens and such, use `Authorization` custom headers instead.
auth: {
  username: 'janedoe',
  password: 's00pers3cret'
},
// `responseType` indicates the type of data that the server will respond with
// options are: 'arraybuffer', 'document', 'json', 'text', 'stream'
//   browser only: 'blob'
responseType: 'json', // default

// `responseEncoding` indicates encoding to use for decoding responses
// Note: Ignored for `responseType` of 'stream' or client-side requests
responseEncoding: 'utf8', // default

// `xsrfCookieName` is the name of the cookie to use as a value for xsrf token
xsrfCookieName: 'XSRF-TOKEN', // default

// `xsrfHeaderName` is the name of the http header that carries the xsrf token value
xsrfHeaderName: 'X-XSRF-TOKEN', // default

// `onUploadProgress` allows handling of progress events for uploads
onUploadProgress: function (progressEvent) {
  // Do whatever you want with the native progress event
},

// `onDownloadProgress` allows handling of progress events for downloads
onDownloadProgress: function (progressEvent) {
  // Do whatever you want with the native progress event
},

// `maxContentLength` defines the max size of the http response content in bytes allowed
maxContentLength: 2000,

// `validateStatus` defines whether to resolve or reject the promise for a given
// HTTP response status code. If `validateStatus` returns `true` (or is set to `null`
// or `undefined`), the promise will be resolved; otherwise, the promise will be
// rejected.
validateStatus: function (status) {
  return status >= 200 && status < 300; // default
},

// `maxRedirects` defines the maximum number of redirects to follow in node.js.
// If set to 0, no redirects will be followed.
maxRedirects: 5, // default

// `socketPath` defines a UNIX Socket to be used in node.js.
// e.g. '/var/run/docker.sock' to send requests to the docker daemon.
```

```
  // Only either `socketPath` or `proxy` can be specified.
  // If both are specified, `socketPath` is used.
  socketPath: null, // default

  // `httpAgent` and `httpsAgent` define a custom agent to be used when performing http
  // and https requests, respectively, in node.js. This allows options to be added like
  // `keepAlive` that are not enabled by default.
  httpAgent: new http.Agent({ keepAlive: true }),
  httpsAgent: new https.Agent({ keepAlive: true }),

  // 'proxy' defines the hostname and port of the proxy server.
  // You can also define your proxy using the conventional `http_proxy` and
  // `https_proxy` environment variables. If you are using environment variables
  // for your proxy configuration, you can also define a `no_proxy` environment
  // variable as a comma-separated list of domains that should not be proxied.
  // Use `false` to disable proxies, ignoring environment variables.
  // `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and
  // supplies credentials.
  // This will set an `Proxy-Authorization` header, overwriting any existing
  // `Proxy-Authorization` custom headers you have set using `headers`.
  proxy: {
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'mikeymike',
      password: 'rapunz3l'
    }
  },

  // `cancelToken` specifies a cancel token that can be used to cancel the request
  // (see Cancellation section below for details)
  cancelToken: new CancelToken(function (cancel) {
  })
}
```

Response Schema

The response for a request contains the following information.

```
{
  // `data` is the response that was provided by the server
  data: {},

  // `status` is the HTTP status code from the server response
  status: 200,
  // `statusText` is the HTTP status message from the server response
  statusText: 'OK',

  // `headers` the headers that the server responded with
  // All header names are lower cased
  headers: {},

  // `config` is the config that was provided to `axios` for the request
  config: {},

  // `request` is the request that generated this response
  // It is the last ClientRequest instance in node.js (in redirects)
  // and an XMLHttpRequest instance the browser
  request: {}
}
```

When using then, you will receive the response as follows:

```
axios.get('/user/12345')
  .then(function (response) {
    console.log(response.data);
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.headers);
    console.log(response.config);
  });
```

When using catch, or passing a rejection callback as second parameter of then, the response will be available through the error object as explained in the Handling Errors section.

Config Defaults

You can specify config defaults that will be applied to every request.

Global axios defaults

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

Custom instance defaults

```
// Set config defaults when creating the instance
const instance = axios.create({
  baseURL: 'https://api.example.com'
});

// Alter defaults after instance has been created
instance.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

Config order of precedence

Config will be merged with an order of precedence. The order is library defaults found in lib/defaults.js, then defaultsproperty of the instance, and finally config argument for the request. The latter will take precedence over the former. Here's an example.

```
// Create an instance using the config defaults provided by the library
// At this point the timeout config value is `0` as is the default for the library
const instance = axios.create();

// Override timeout default for the library
// Now all requests using this instance will wait 2.5 seconds before timing out
instance.defaults.timeout = 2500;

// Override timeout for this request as it's known to take a long time
instance.get('/longRequest', {
  timeout: 5000
});
```

Interceptors

You can intercept requests or responses before they are handled by then or catch.

```
// Add a request interceptor
axios.interceptors.request.use(function (config) {
    // Do something before request is sent
    return config;
  }, function (error) {
    // Do something with request error
    return Promise.reject(error);
  });

// Add a response interceptor
axios.interceptors.response.use(function (response) {
    // Do something with response data
    return response;
  }, function (error) {
    // Do something with response error
    return Promise.reject(error);
  });
```

If you may need to remove an interceptor later you can.

```
const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

You can add interceptors to a custom instance of axios.

```
const instance = axios.create();
instance.interceptors.request.use(function () {/*...*/});
```

Handling Errors

```
axios.get('/user/12345')
  .catch(function (error) {
    if (error.response) {
      // The request was made and the server responded with a status code
      // that falls out of the range of 2xx
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // The request was made but no response was received
      // `error.request` is an instance of XMLHttpRequest in the browser and an instance of
      // http.ClientRequest in node.js
      console.log(error.request);
    } else {
      // Something happened in setting up the request that triggered an Error
      console.log('Error', error.message);
    }
    console.log(error.config);
  });
```

You can define a custom HTTP status code error range using the validateStatus config option.

```
axios.get('/user/12345', {
  validateStatus: function (status) {
    return status < 500; // Reject only if the status code is greater than or equal to 500
  }
})
```

Cancellation

You can cancel a request using a cancel token.

The axios cancel token API is based on the withdrawn cancelable promises proposal.

You can create a cancel token using the CancelToken.source factory as shown below:

```
const CancelToken = axios.CancelToken;
const source = CancelToken.source();

axios.get('/user/12345', {
  cancelToken: source.token
}).catch(function (thrown) {
  if (axios.isCancel(thrown)) {
    console.log('Request canceled', thrown.message);
  } else {
    // handle error
  }
});

axios.post('/user/12345', {
  name: 'new name'
}, {
  cancelToken: source.token
})

// cancel the request (the message parameter is optional)
source.cancel('Operation canceled by the user.');
```

You can also create a cancel token by passing an executor function to the CancelToken constructor:

```
const CancelToken = axios.CancelToken;
let cancel;

axios.get('/user/12345', {
  cancelToken: new CancelToken(function executor(c) {
    // An executor function receives a cancel function as a parameter
    cancel = c;
  })
});

// cancel the request
cancel();
```

Note: you can cancel several requests with the same cancel token.

Using application/x-www-form-urlencoded format

By default, axios serializes JavaScript objects to JSON. To send data in the application/x-www-form-urlencoded format instead, you can use one of the following options.

Browser

In a browser, you can use the URLSearchParams API as follows:

```
const params = new URLSearchParams();
params.append('param1', 'value1');
params.append('param2', 'value2');
axios.post('/foo', params);
```

Note that URLSearchParams is not supported by all browsers (see caniuse.com), but there is a polyfill available (make sure to polyfill the global environment).

Alternatively, you can encode data using the qs library:

```
const qs = require('qs');
axios.post('/foo', qs.stringify({ 'bar': 123 }));
```

Or in another way (ES6),

```
import qs from 'qs';
const data = { 'bar': 123 };
const options = {
  method: 'POST',
  headers: { 'content-type': 'application/x-www-form-urlencoded' },
  data: qs.stringify(data),
  url,
};
```

```
axios(options);
```
Node.js
In node.js, you can use the querystring module as follows:
```
const querystring = require('querystring');
axios.post('http://something.com/', querystring.stringify({ foo: 'bar' }));
```
You can also use the qs library.
NOTE
The qs library is preferable if you need to stringify nested objects, as the querystring method has known issues with that use case (https://github.com/nodejs/node-v0.x-archive/issues/1665).
Semver
Until axios reaches a 1.0 release, breaking changes will be released with a new minor version. For example 0.5.1, and 0.5.4 will have the same API, but 0.6.0 will have breaking changes.
Promises
axios depends on a native ES6 Promise implementation to be supported. If your environment doesn't support ES6 Promises, you can polyfill.
TypeScript
axios includes TypeScript definitions.
```
import axios from 'axios';
axios.get('/user?ID=12345');
```

2.2)jwt-decode
Usage
```
var token = 'eyJ0eXAiO.../// jwt token';

var decoded = jwt_decode(token);
console.log(decoded);

/* prints:
 * { foo: "bar",
 *   exp: 1393286893,
 *   iat: 1393268893  }
 */
```
Note: A falsy token will throw an error.
Can also be used with browserify or webpack by doing npm install jwt-decode and requiring:
```
var jwtDecode = require('jwt-decode');
```
Polymer Web Component
Can also be installed and used with Polymer-based wrapper.

2.3) Native Base
1. What is NativeBase?
NativeBase is a sleek, ingenious and dynamic front-end framework created by passionate React Loving team at Geekyants.com to build cross platform Android & iOS mobile apps using ready to use generic components of React Native.

2. Why NativeBase?
What is really great with NativeBase is that you can use shared UI cross-platform components, which will drastically increase your productivity. When using NativeBase, you can use any native third-party libraries out of the box.
Recommended by Microsoft
If you're looking to get started with React Native + CodePush, and are looking for an awesome starter kit, you should check out Native Starter Pro - Microsoft's react-native-code-push repo
Recommended by Awesome React Native
NativeBase added into the list of Frameworks of Awesome React Native and are also used by many other React lovers across the world.
4. Getting Started
a. Setup with pure React Native app
Install NativeBase

```
npm install native-base --save
```

Install Peer Dependencies
The peer dependencies included from any npm packages does not automatically get installed. Your application will not depend on it explicitly.

```
react-native link
```

You've successfully setup NativeBase with your React Native app. Your React Native app is now all set to run on iOS and Android simulator.
b. Setup with Expo
Expo helps you make React Native apps with no build configuration. It works on macOS, Windows, and Linux.
Refer this link for additional information on Expo
Install NativeBase

```
npm install native-base --save
```

Note
NativeBase uses some custom fonts that can be loaded using Font.loadAsync. Check out the Expo Font documentation.
Syntax

```
// At the top of your file
import { Font } from 'expo';
import { Ionicons } from '@expo/vector-icons';

// Later on in your component
async componentDidMount() {
  await Font.loadAsync({
    'Roboto': require('native-base/Fonts/Roboto.ttf'),
    'Roboto_medium': require('native-base/Fonts/Roboto_medium.ttf'),
    ...Ionicons.font,
  });
```

}

Check out the KitchenSink with Expo for an example of the implementation.
Find the KitchenSink repo here
c. Setup with ignite-native-base-boilerplate
You can run the following command to create the boilerplate, provided you have Ignite CLIinstalled.

```
ignite new appname --boilerplate native-base-boilerplate
```

Go to app location

```
cd appname
```

For iOS run

```
react-native run-ios
```

For Android run

```
react-native run-android
```

Refer ignite-native-base-boilerplate page for additional information
5. Components
NativeBase is made from effective building blocks referred to as components. The Components are constructed in pure React Native platform along with some JavaScript functionality with rich set of customisable properties. These components allow you to quickly build the perfect interface.
6. NativeBase for Web
NativeBase is now available for React web lovers. Check the demo Find the repo here
7. Compatibility Versions

NativeBase	React Native
v0.1.1	v0.22 to v0.23
v0.2.0 to v0.3.1	v0.24 to v0.25
v0.4.6 to v0.4.9	v0.26.0 - v0.27.1
v0.5.0 to v0.5.15	v0.26.0 - v0.37.0
v0.5.16 to v0.5.20	v0.38.0 - v0.39.0
v2.0.0-alpha1 to v2.1.3	v0.38.0 to v0.43.0
v2.1.4 to v2.1.5	v0.44.0 to v0.45.0
v2.2.0	v0.44.0 to v0.45.0
v2.2.1	v0.46.0 and above
v2.3.0 to 2.6.1	v0.46.0 and above (does not support React 16.0.0-alpha.13)
v2.7.0	v0.56.0 and above

8. React Native Seed
React Native Seed provides you React Native starter kits for your base app with the technologies that you love.
Based on the feedback we received from our users, people had trouble sorting out the right boilerplate for them with the desired technologies and contacted us to enquire. We realized that many people were particular about the technologies they want in the app and that a minimal, neat solution was required to solve this, and hence, React Native Seed.

React Native Seed is for learners and professionals alike, those who want to experiment, learn all aspects and those who already know enough, just want a starter kit to quickly start working on their project.
9. NativeBase Market
Having tried with the free version, Native Starter Kit and appreciate our product?
Get on the mobile fast track with the featured apps of NativeBase, to build high-quality iOS and Android mobile apps.
A marketplace for premium React Native app themes to build high-quality iOS and Android mobile apps.

2.4) Redux
Getting Started with Redux
Redux is a predictable state container for JavaScript apps.
It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.
You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.
Installation
Redux is available as a package on NPM for use with a module bundler or in a Node application:

```
npm install --save redux
```

Copy
It is also available as a precompiled UMD package that defines a window.Redux global variable. The UMD package can be used as a <script> tag directly.
For more details, see the Installation page.
Redux Starter Kit
Redux itself is small and unopinionated. We also have a separate package called redux-starter-kit, which includes some opinionated defaults that help you use Redux more effectively.
It helps simplify a lot of common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.
Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, redux-starter-kit can help you make your Redux code better.
Basic Example
The whole state of your app is stored in an object tree inside a single store.
The only way to change the state tree is to emit an action, an object describing what happened.
To specify how the actions transform the state tree, you write pure reducers.
That's it!

```
import { createStore } from 'redux'

/**
```

```
 * This is a reducer, a pure function with (state, action) => state signature.
 * It describes how an action transforms the state into the next state.
 *
 * The shape of the state is up to you: it can be a primitive, an array, an object,
 * or even an Immutable.js data structure. The only important part is that you should
 * not mutate the state object, but return a new object if the state changes.
 *
 * In this example, we use a `switch` statement and strings, but you can use a helper that
 * follows a different convention (such as function maps) if it makes sense for your
 * project.
 */
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
let store = createStore(counter)

// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.
// However it can also be handy to persist the current state in the localStorage.

store.subscribe(() => console.log(store.getState()))

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'INCREMENT' })
// 1
store.dispatch({ type: 'INCREMENT' })
// 2
store.dispatch({ type: 'DECREMENT' })
// 1
```
Copy

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called actions. Then you write a special function called a reducer to decide how every action transforms the entire application's state.
In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.
This architecture might seem like an overkill for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

Result

After the initial basic marketing campaign, we have 25 enrolled users.

Services initiated by those users vary a lot, from normal services to anonymous services, and from very general ones to very specific ones.

We still haven’t reached everyone, we are still working on it. Facebook advertisement is a great option which we are considering.

Number of users (including both askers and helpers): 25

Number of active services: 17

Discussion

What is inevitable?

We found that it is inevitable to have an incentive system in an app like ours.

In the marketing world, an “incentive” is something that motivates an individual to perform an action, such as making a purchase, completing a survey or signing up for a mailing list. In other words, it’s an “enticement” to get customers and prospects to do what you want them to do, so in our case incentives will play a critical role in encouraging users to participate more in the app, and help others in services.

Our incentive system consists of two types of incentives:
1-Givantk points.
2-Money Score.

1-Givantk points are points that helpers take for helping in free services, they may exchange it later to discounts in stores.
To know how givantk points work, we will discuss the following example:

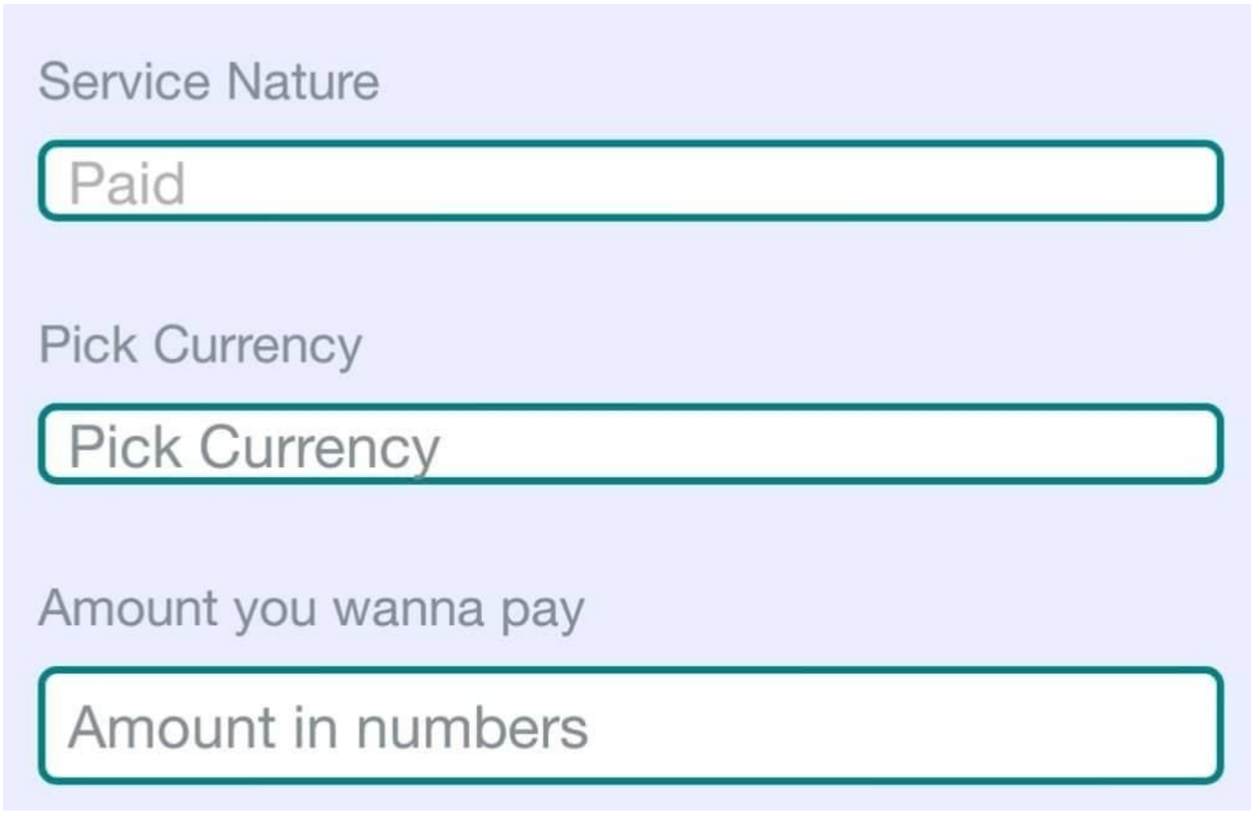
“User A registered in the app, and he will find that he has an amount of 100 givantk points in his account.
He can use them to order as much free services as he wants, any free service requires at least 1 givantk point. So he asked for a service with 10 points.
If User B helped User A in his service, and the service is finished successfully. User B will take the 10 points, and they will be added to his account.
User A account now will only has 90 points.
If he finished his points, he is encouraged to help in more free services to get more points, but he can also get a random number of points if he wants to ask for new free service, but this random number of points will not exceed 10 points.”

2-Money Score: this score is equivalent to the real money that the user charges his account by.
And here’s an example to the way that is used in our app:
“User A wants to ask for a paid service, he initially doesn’t have any money in his account in the app, so he goes to his account tab and charge it with visa or Vodafone cache, then his money appears in his account.
User A asks for a service for 100 EGP, then the app takes a share of this price, and the service is being published for 91 EGP.
If no user is accepted in this service, User A can archive the service and take all his money back in his account (100 EGP).

If a user is accepted in this service, the money will be in hold until the service is finished.
If the service is finished successfully, User B takes 91 EGP."

How all of these look in the app?

1-Here's we ask the user for the nature and price of service before publishing it.
Depending on his choice of the nature of the service the price will differs, if the user chooses free services, he will deal with money score, else he will deal with givantk points.


Service Nature

Paid

Pick Currency

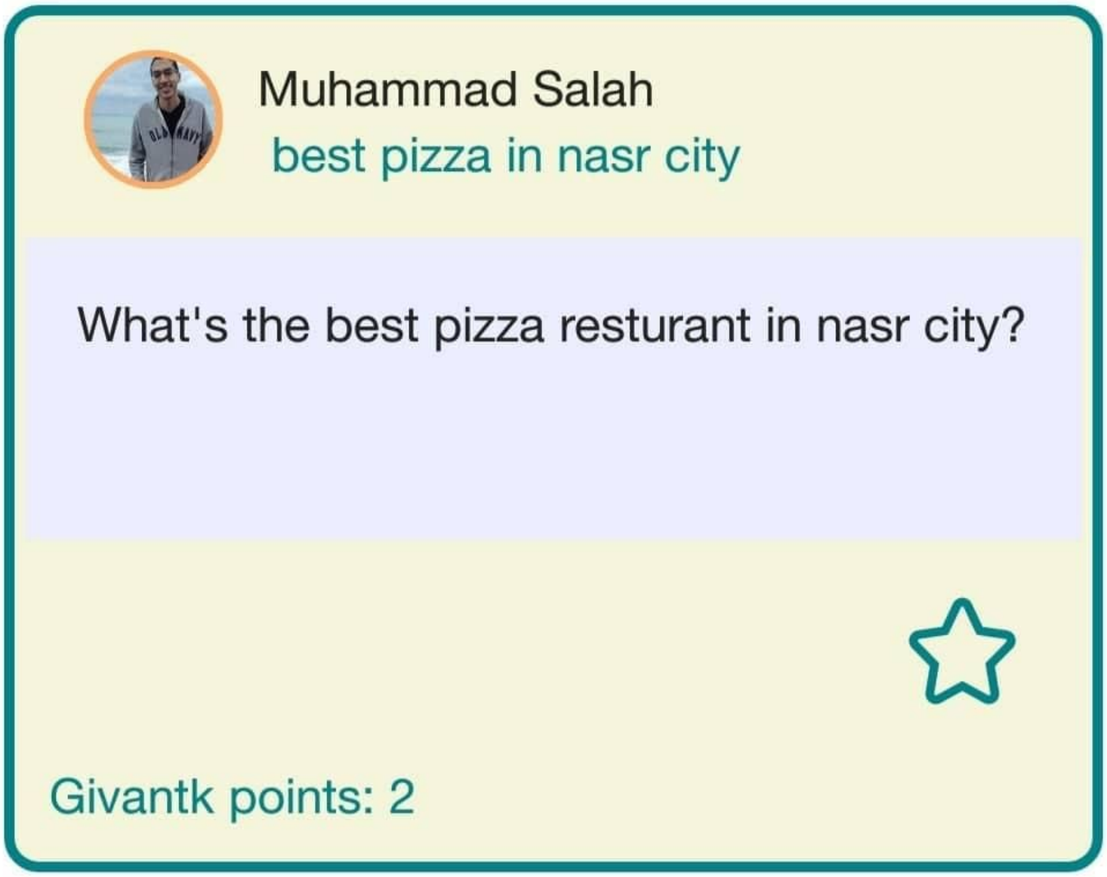
Pick Currency

Amount you wanna pay

Amount in numbers

2-Example for free services, and the number of points being displayed in the service card.

So this will make it easy for the helpers to surf the services easily knowing how much exactly does each service cost.



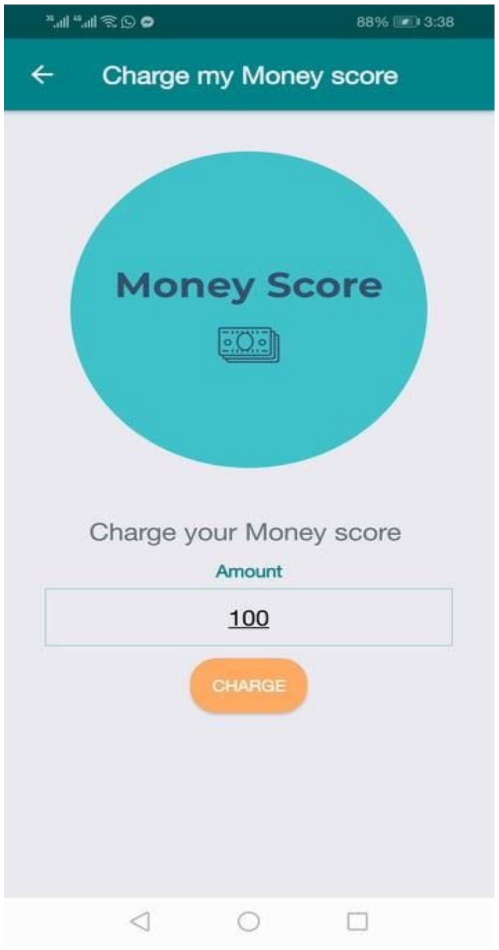
Muhammad Salah

best pizza in nasr city

What's the best pizza resturant in nasr city?

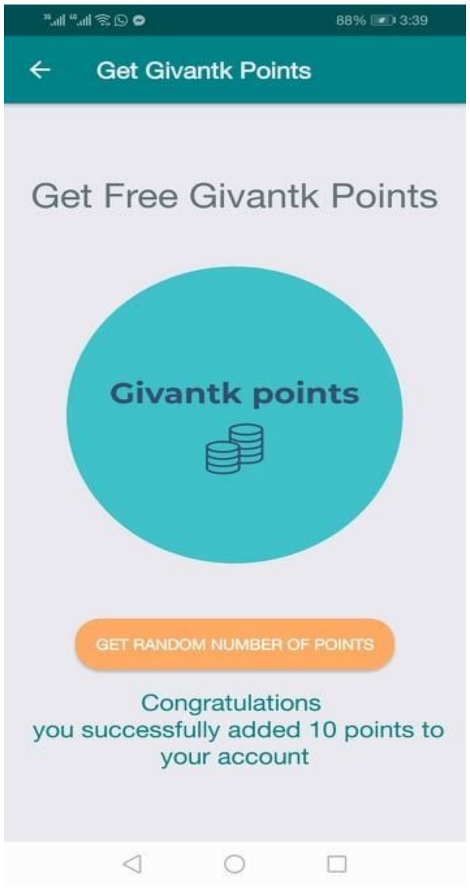
Givantk points: 2

3-Example for charging money score with money:
This screen acts as an interface, where the user will enter the amount of money by which he wants to charge his account, and then the app deals with the payment gateway, and if it received a success token, the money will be added to the user account.



4-Example for getting random number of givantk points:
The point of making this is to make sure that users can always ask for free services, even if they have spent all their free points, also another thing to point out is that this adds the element of

gamification to the app by getting the user excited about getting new random points.

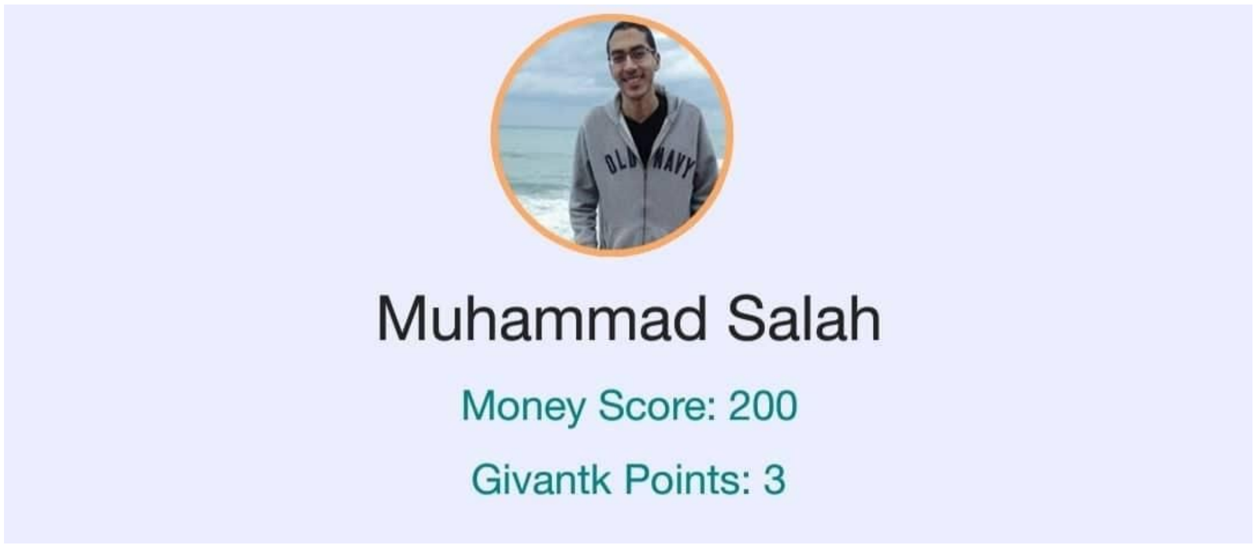


Congratulations
you successfully added 10 points to your account

5-Example of displaying givantk points and money score in the account tab:

Note that the money score is displayed in the account screen only, but not in the profile, while givantk points (free points) are displayed inside the profile normally.

This ensures the privacy of the user, by hiding how much money score does the user have on the app.



Muhammad Salah

Money Score: 200

Givantk Points: 3