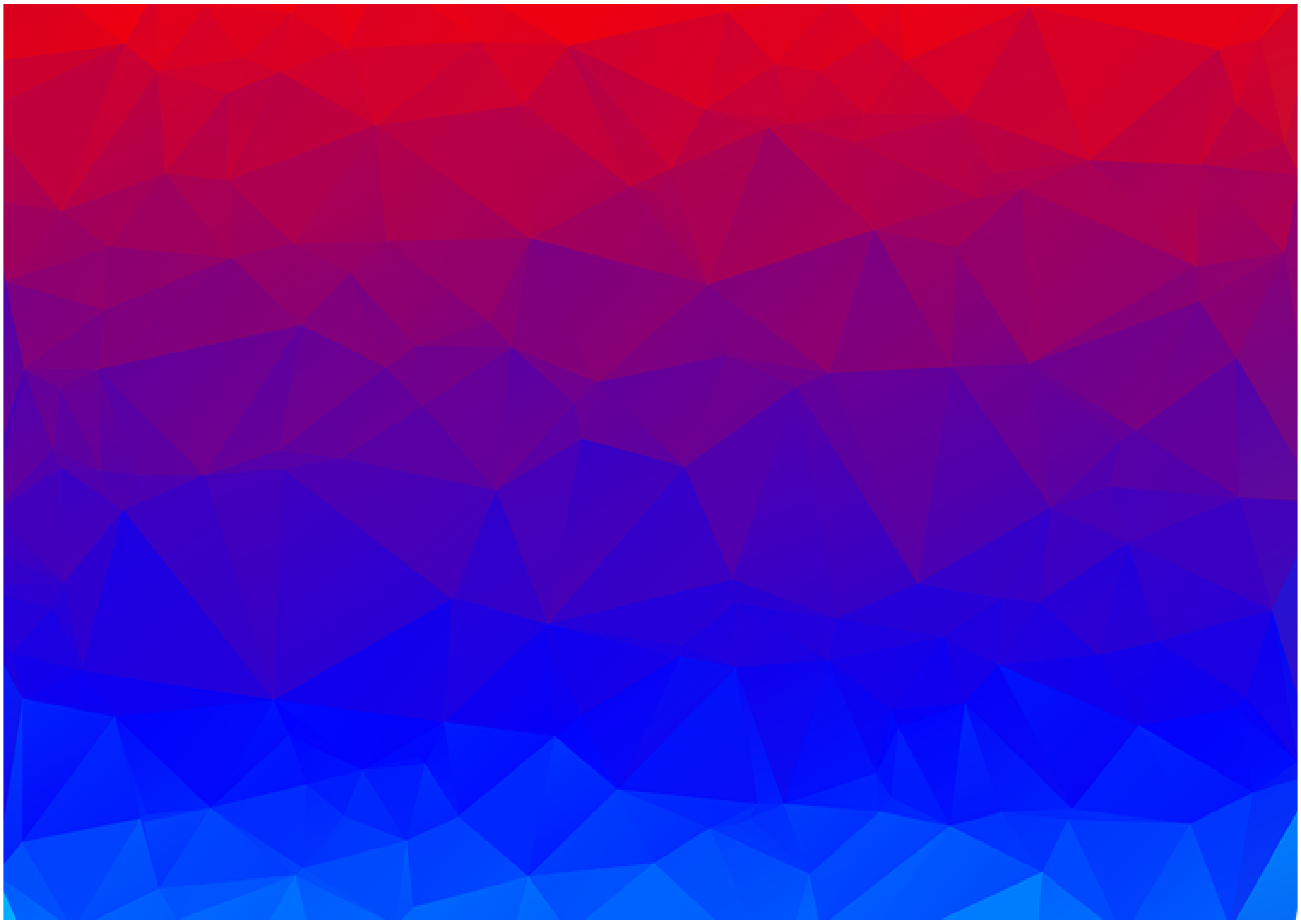


Project X.

FULL DEVELOPMENT DOCUMENTATION

by Matan Yosef

Contents

Design Statement:

I am designing a web-application that will encourage people to plan and complete their bucket-list of experiences. This social bucket-list network solves the issue of failure of completion, and failure to find people to complete it with.
Please see here for more information.

Planning Documentation:

Databases:

Tables

User		
id	int	pk, auto-inc
name	string(160)	null=false
email	string(240)	unique, null=false
password	string()*[1]	Null = false
birthday	string(10)	-
country	string(2)*[2]	-
role	string(20)	default=urole*[3]
image_file	string(20)	Default =default.jpg*[4]

DB Relationship (to List): Back-reference of 'owner' i.e. they own the list.

*Relevant Justifications:

1. The attribute 'password' has unspecified string length due to variation in length of passwords (i.e. the longer they are, the more secure they are - a max length serves no purpose).
2. I will create a dictionary field for country so that once a country is selected, only the 2-character code is stored (i.e. when choosing New Zealand, NZ gets stored.)
3. Sets users as having the role 'urole' by default. This means that unless otherwise (i.e. registered as a service), they only have permissions to access user related fields.
4. The 'default.jpg' file will be a blank 'filler' image for those who do not wish to have a profile picture.

List		
id	int	pk, auto-inc
item	string(1000)	null=false
user_id	int	foreign key (user.id), null=false
completed	int	default=0*[1]

*Relevant Justifications:

1. The default value of completed is stored as an int of 0, which corresponds to false.

Wall		
id	int	pk, auto-inc
list_id	int	foreign key (list.id), null=false
image_file	string(20)	null=false, default=defaultWall.jpg[1]
caption	string(2000)	-
date_completed	string(10)	-
location_completed	string(160)	-

*Relevant Justifications:

1. Although a photo must be uploaded, the ‘defaultWall.jpg’ file will serve as a blank ‘filler’ image for images that fail to upload.

Circle		
id	int	pk, auto-inc
user_id	int	foreign key (user.id), null=false
friend_user_id	int	foreign key (user.id), null=false

Incremental Table Additions (i.e. after User's is completed):

Service		
id	int	pk, auto-inc
user_id	int	foreign key (user.id), null=false
name	string(160)	null=false
email	string(240)	unique, null=false
password	string()	null=false
adress_number	int(6)*[1]	
address_street	string(240)	null=false
address_subrub	string(240)	-
address_city	string(240)	null=false
address_country	string(160)	null=false
description	string(2000)	null=false
category*[2]	int	null=false
web_link	string(2000)	

DB Relationship (to Service_Post): Back-reference of company i.e. they write the posts.

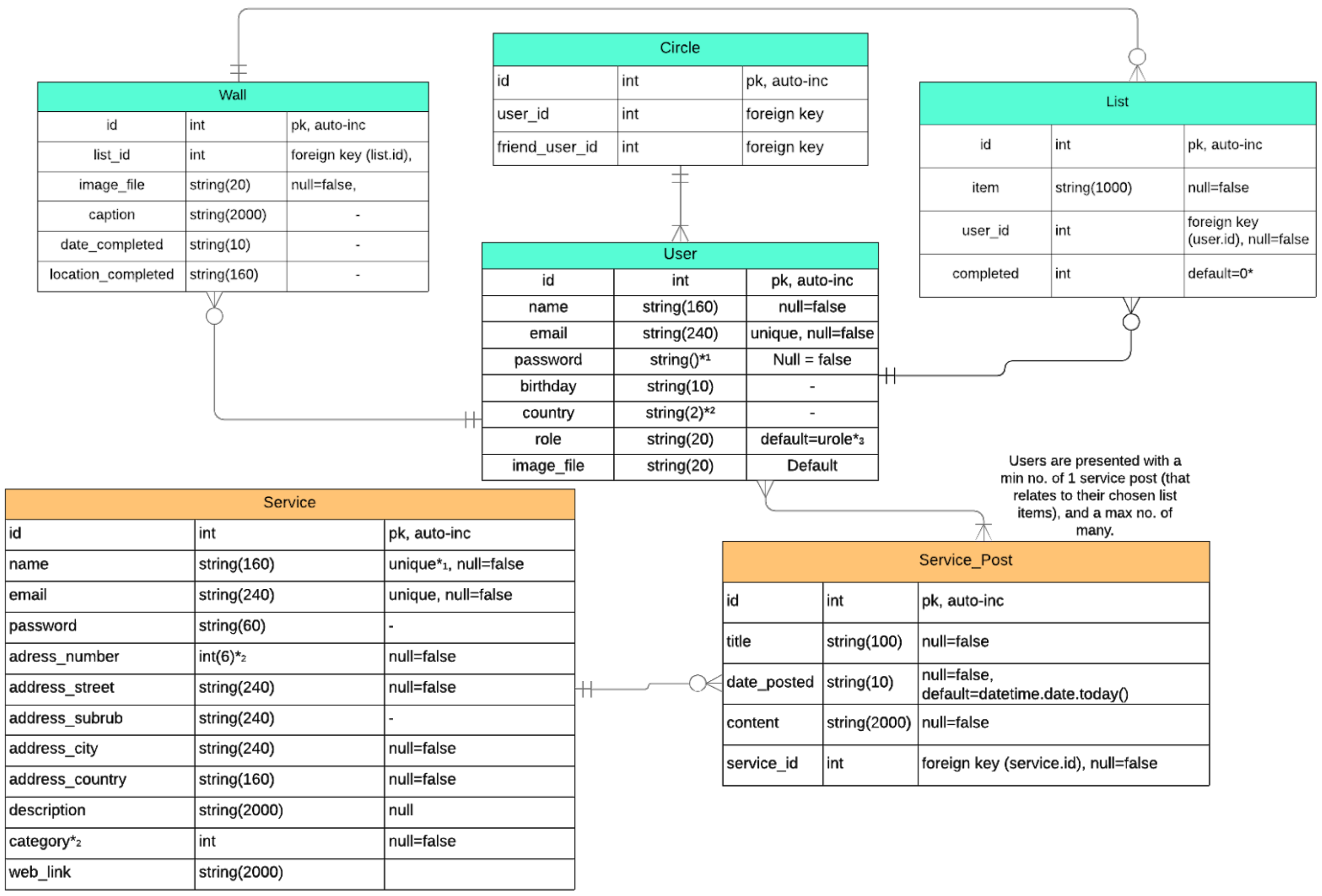
*Relevant Justifications:

1. The longest address no. is '21114 Yonge Street,' having a total of 5 characters. Adding another character prevents mistaken characters and fake numbers.
2. Category will help sort what to display to users.

Service_Post		
id	int	pk, auto-inc
title	string(100)	null=false
date_posted	string(10)	null=false, default=datetime.date.today()

content	string(2000)	null=false
post_picture	string(20)	default='defaultService.jpg'

ER Diagram



Wall
id | int | pk, auto-inc
list_id | int | foreign key (list.id),
image_file | string(20) | null=false,
caption | string(2000) | -
date_completed | string(10) | -
location_completed | string(160) | -
Circle
id | int | pk, auto-inc
user_id | int | foreign key
friend_user_id | int | foreign key
List
id | int | pk, auto-inc
item | string(1000) | null=false
user_id | int | foreign key (user.id), null=false
completed | int | default=0*
User
id | int | pk, auto-inc
name | string(160) | null=false
email | string(240) | unique, null=false
password | string()*1 | Null = false
birthday | string(10) | -
country | string(2)*2 | -
role | string(20) | default=urole*3
image_file | string(20) | Default
Users are presented with a min no. of 1 service post (that relates to their chosen list items), and a max no. of many.
Service
id | int | pk, auto-inc
name | string(160) | unique*1, null=false
email | string(240) | unique, null=false
password | string(60) | -
adress_number | int(6)*2 | null=false
address_street | string(240) | null=false
address_subrub | string(240) | -
address_city | string(240) | null=false
address_country | string(160) | null=false
description | string(2000) | null
category*2 | int | null=false
web_link | string(2000) |
Service_Post
id | int | pk, auto-inc
title | string(100) | null=false
date_posted | string(10) | null=false, default=datetime.date.today()
content | string(2000) | null=false
service_id | int | foreign key (service.id), null=false

Example SQL Queries and Corresponding SQL Alchemy Translations

User Registration Query

SQL Query:	**SQLAlchemy Translation:**
`INSERT INTO user (name, email, password, birthday, country, image_file) VALUES (?, ?, ?, ?, ?, ?)]`	`new_user = User(name=form.name.data, email=form.email.data, password=somehashedpassword, birthday=form.birthday.data, country=form.country.data,) db.session.add(new_user)`

(since it is a form entry, the above question marks are used with the following example parameters being used once submitted) `[parameters: ('Matan Yosef', 'yoyomat13@gmail.com', 'somehashedpassword, datetime.date(2002, 1, 13), 'NZ', 'default.jpg')]`	`db.session.commit()`

Service Registration Query

SQL Query:	**SQLAlchemy Translation:**

`INSERT INTO service(name, email, password, address_number, address_street, address_subrub, address_city, address_country, description) VALUES (?, ?, ?, ?, ?, ?, ?, ?)]`
(since it is a form entry, the above question marks are used with the following example parameters being used once submitted)
`[parameters: ('CompanyName1', 'companyemail@gmail.com', 'somehashedpassword, ‘151’, ‘Memorial Avenue’, ‘Burnside’, ‘Christchurch’, 'NZ', 'Lorem ipsum dolor sit amet, consectetur adipiscing elit')]`
	`new_service = Service(name=form.name.data, email=form.email.data, password=somehashedpassword, address_number=form.address_number.data, address_street=form.address_street.data, address_subrub=form.address_subrub.data, address_city=form.address_city.data, address_country=form.address_country.data, description=from.description.data)`
	`db.session.add(new_service)`
	`db.session.commit()`

User Login Query - Checks if email is recognised

SQL Query:	**SQLAlchemy Translation:**

`SELECT * FROM User WHERE email=? LIMIT 1;`
`[parameters: (‘formdata’)]`
	`User.query.filter_by(email=formdata).first()`

Service Login Query - Checks if Service needs details filled.

SQL Query:	**SQLAlchemy Translation:**
`SELECT Service.id AS user_id FROM Service WHERE service.email = ? [parameters: ('formdata')]`	`db.session.query(Service.id).filter_by(email=form.email.data).scalar()`

Service Query - Retrieves All Post data belonging to them.

SQL Query:	**SQLAlchemy Translation:**
`SELECT * FROM Service_Post WHERE company = ? [parameters: ('loggedincompany')]`	`Service_Post.query.filter_by(company=loggedincompany).all()`

List Query - Retrieves All List Data

SQL Query:	**SQLAlchemy Translation:**
`SELECT * FROM List WHERE user_id = ? LIMIT 1; [parameters: (currentuserid)]`	`List.query.filter_by(user_id=currentuserid).first()`

Estimated models.py Translation

```
class User(db.Model):
  __tablename__ = 'user'

  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(160), nullable=False)
  email = db.Column(db.String(240), unique=True, nullable=False )
  password = db.Column(db.String, nullable=False)
  birthday = db.Column(db.String(10))
  country = db.Column(db.String(2))
  image_file = db.Column(db.String(20), default="default.jpg")
  bucket_list = db.relationship('List', backref='owner', lazy=True)

  def __repr__(self):
      return f"User: ('{self.name}', '{self.email}', '{self.image_file}')"
```

```
class List(db.Model):
  __tablename__ = 'list'
  id = db.Column(db.Integer, primary_key=True)
  item = db.Column(db.String(1000), nullable=False)
  user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
  completed = db.Column(db.Integer, default=0)

  def __repr__(self):
      return f"List Item: ('{self.item}', '{self.user_id}')"

class Wall(db.Model):
  __tablename__ = 'wall'
  id = db.Column(db.Integer, primary_key=True)
  list_id = db.Column(db.Integer, db.ForeignKey('list.id'),
nullable=False)
  image_file = db.Column(db.String(20), nullable=False,
default="defaultWall.jpg")
  caption = db.Column(db.String(2000))
  date_completed = db.Column(db.Date)
  location_completed = db.Column(db.String(160))

  def __repr__(self):
      return f"Wall Item: ('{self.list_id}', '{self.image_file}',
'{self.date_completed}')"

class Circle(db.Model):
  __tablename__ = 'circle'
  id = db.Column(db.Integer, primary_key=True)
  user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=false)
  friend_user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=false)

  def __repr__(self):
      return f"Circle: ('{self.item}', '{self.friend_user_id}')"

class Service(db.Model):
  __tablename__ = 'service'
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(160), nullable=False)
  email = db.Column(db.String(240), unique=True, nullable=False )
  password = db.Column(db.String, nullable=False)  address_number =
db.Column(db.Integer(6), nullable=False)
  address_street = db.Column(db.String(240), nullable=False)

  address_suburb = db.Column(db.String(240))
  address_city = db.Column(db.String(240), nullable=False)
  address_country = db.Column(db.String(160), nullable=False)
  description = db.Column(db.String(2000), nullable=False)
  category=db.Column(db.Integer, nullable=False)
  posts = db.relationship('Service_Post', backref='company', lazy=True)

  def __repr__(self):
      return f"Service: ('{self.name}', '{self.email}',
'{self.address_city}', '{self.address_country}', '{self.description}')"

class Service_Post(db.Model):
  __tablename__ = 'service_post'
  id = db.Column(db.Integer, primary_key=True)
  title = db.Column(db.String(100), nullable=False)
  date_posted = db.Column(db.String(10), nullable=False,
default=datetime.date.today())
  content = db.Column(db.String(2000), nullable=False)
  service_image_file = db.Column(db.String(20),
default='defaultService.jpg')  service_id = db.Column(db.Integer,
db.ForeignKey('service.id'),
nullable=False)

  def __repr__(self):
      return f"Service Post: ('{self.title}', '{self.date_posted}')"
```

Media:

Conceptual Design

Name Ideas (list will continue as idea develops):

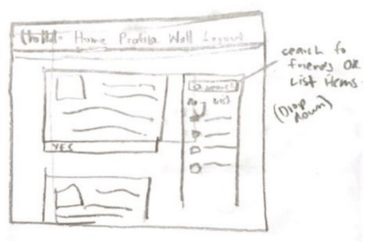
Forlive, Lister, Tick, Threaks, AThrill, Bravvy, Joltrive, Stealtzy, thrillzy,Thrilld.

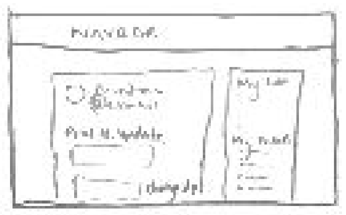
FINAL CHOICE: thrilld. This encapsulates the thrill-seeking aspect of achieving a bucket list, and is a short, sharp, and modern name.

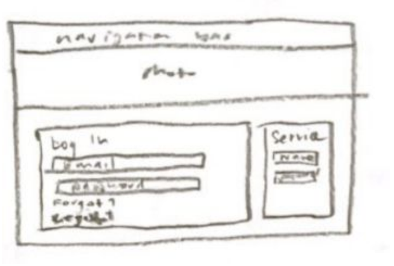
Screen Concepts - Paper

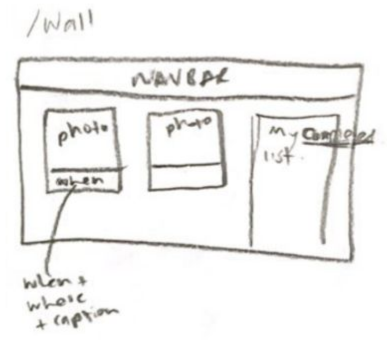
Please See here for full development of paper concepts.

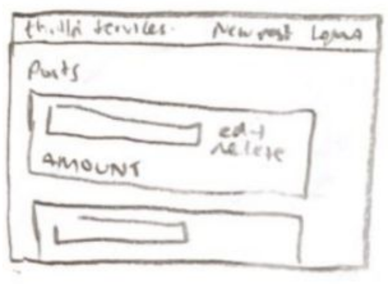
The final paper designs I will be using are as follows:

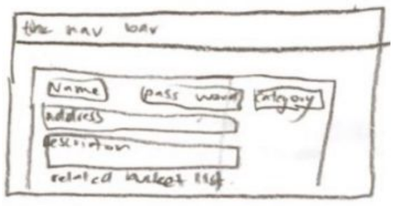

thrilld Home Profile Wall Logout
search for friends OR list items (Drop down)
YES







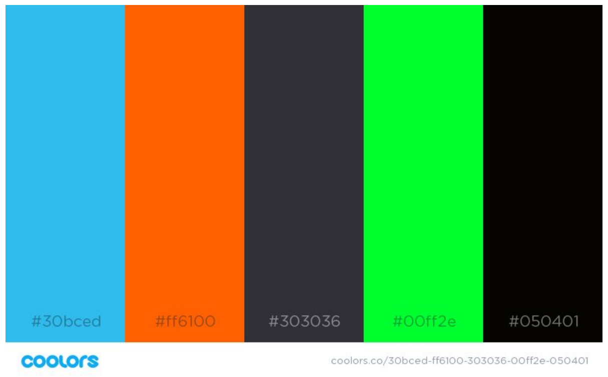




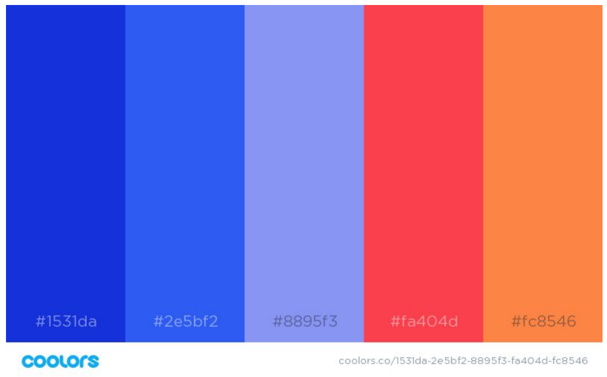
Colour Scheme and Fonts:

Name and Big Heading Font: Horatio Std Bold (thrilld.)
Other Font: Raleway

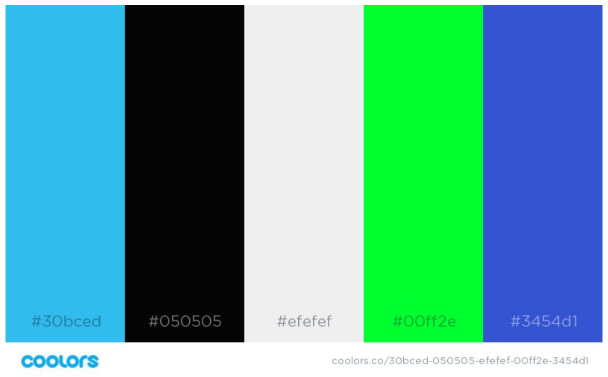
Possible Colour Schemes



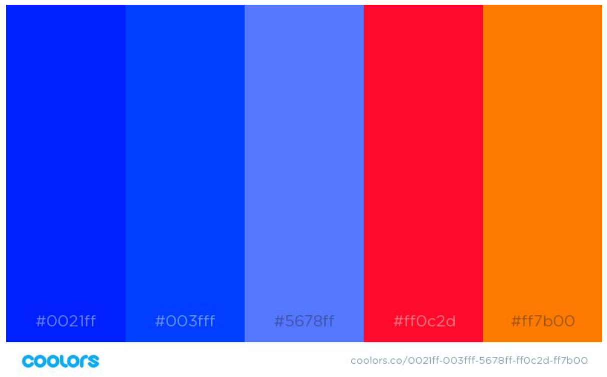
https://coolors.co/30bced-ff6100-303036-00ff2e-050401



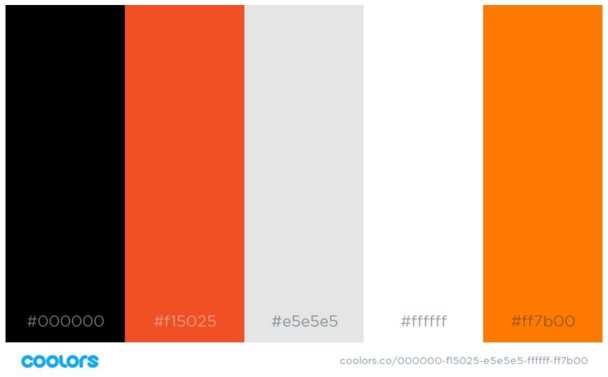
https://coolors.co/1531da-2e5bf2-8895f3-fa404d-fc8546



https://coolors.co/30bced-050505-efefef-00ff2e-3454d1



https://coolors.co/0021ff-003fff-5678ff-ff0c2d-ff7b00

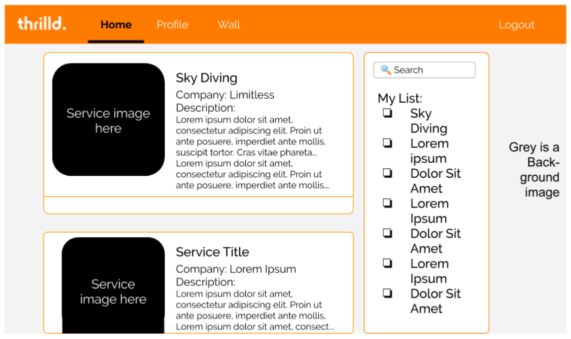


https://coolors.co/000000-f15025-e5e5e5-ffffff-ff7b00

↑ ↑
↑ ↑

FINAL CHOICE: The color scheme shown here This is because orange covers the liveliness nature of the web-app, inspired by a sunset. The palette can also be used with colours such as white and black also means

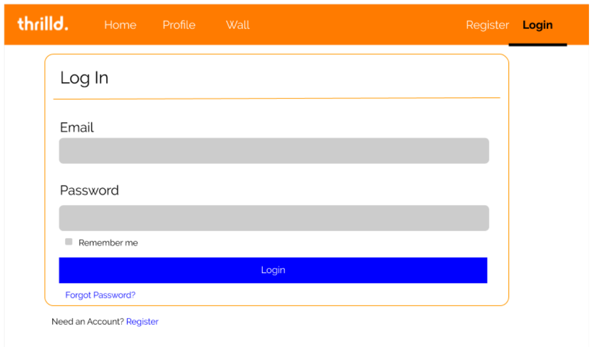
Final Renders



Sky Diving
Company: Limitless
Description:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin ut ante posuere, imperdiet ante mollis, suscipit tortor. Cras vitae phareta... Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin ut ante posuere, imperdiet ante mollis....

Service Title
Company: Lorem Ipsum
Description:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin ut ante posuere, imperdiet ante mollis, Lorem ipsum dolor sit amet, consect...

My List:
- ❏ Sky Diving
- ❏ Lorem ipsum
- ❏ Dolor Sit Amet
- ❏ Lorem Ipsum
- ❏ Dolor Sit Amet
- ❏ Lorem Ipsum
- ❏ Dolor Sit Amet



Log In

Email

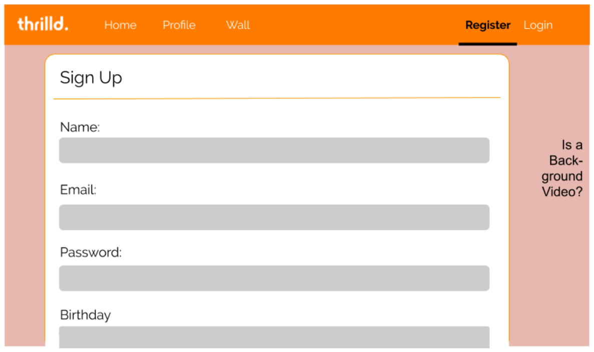
Password

Remember me

Login

Forgot Password?

Need an Account? Register



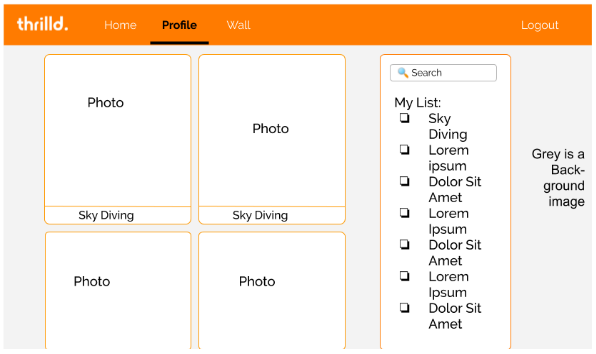
Sign Up

Name:

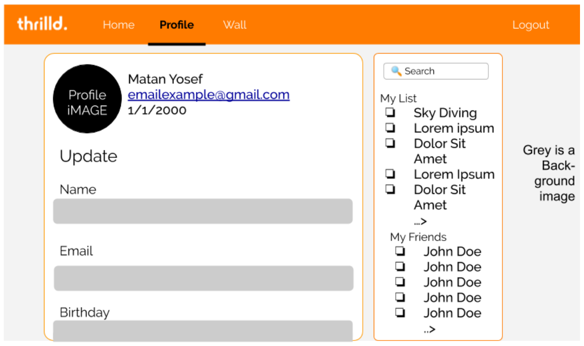
Email:

Password:

Birthday



My List:
- ❏ Sky Diving
- ❏ Lorem ipsum
- ❏ Dolor Sit Amet
- ❏ Lorem Ipsum
- ❏ Dolor Sit Amet
- ❏ Lorem Ipsum
- ❏ Dolor Sit Amet



Matan Yosef
emailexample@gmail.com
1/1/2000

Update

Name

Email

Birthday

My List
- ❏ Sky Diving
- ❏ Lorem ipsum
- ❏ Dolor Sit Amet
- ❏ Lorem Ipsum
- ❏ Dolor Sit Amet

...>

My Friends
- ❏ John Doe
- ❏ John Doe
- ❏ John Doe
- ❏ John Doe
- ❏ John Doe

...>

Development Log:

Please see my git log for additional dates. Documentation is for changes substantial enough to document.

5th June, 2019: Database Change - Keyword Column Added to Service

LOG: This keywords column will be a string field that lets services enter their own keywords in a single string. Originally, I was going to use a category for sorting, however this category would need to be defined for every item a user added to their list. If this list item was user defined, then this would cause an additional category select field relevant to that list (TLDR: it become more complicated). By sorting it with keywords, I can match users with different posts based on the service's keywords, and the contents of the list item (i.e. a skydiving service would have relevant keywords like: "skydiving, sky, dive". A user with 'Go skydiving' or 'Go on a sky dive' on their list would mean a match, and hence they would filter to see that companies posts.

20th June, 2019: Database Change - Tags

LOG: Upon talking to Mr Dunford, I have decided to add a table to my database called 'Tags.' Originally, I intended to only match the users with a post if it had the same keyword that existed in the list-item. This meant separating the keyword string into using '.split()', and then using a for loop that said "for keyword in keywords:" However, I did not realise that I would need two more for loops i.e. for each keyword in keywords and then for each list item and if they matched, within those for loops for each post in posts etc. (TLDR: it created an n^3 problem).

Separating the keywords into separate tags when they add the keywords to the db will mean I can simply query for all the tags at once for a matched company. This means I can just go for each matched company, filter these posts.

15th July, 2019: Database Change - Flagged Column Added to Service_Post

LOG: After much deliberation, to filter posts for a user based on a company's tags, I will add a true or false value in a 'flagged' column of each service post. If the user matches it, it is flagged.

24th July, 2019: Database Change(s) - Date added column to List. Added user_id Column to Wall

LOG: A new notification function - this notification will be a modal box that I'm adding to meet the requirements of my project (encouragement). For version 1.0.0, it will only be a notification to say that if the current date is more than **n** days over the date the list item was added, give the users a notification. I also forgot to add user_id column to the Wall (silly mistake), so I have done so now.

25th July, 2019: Database Change - Many to Many, Category Column Removal

LOG: I have discovered that the flagged column was the incorrect approach to filtering posts. Although this works for one user, problems arise when two users, who could theoretically be logged in at the same time, would start to arise. This is because the function would be running at the same time as the other, causing an internal server error.

After going back to my ER diagram today, I saw the many to many relationships between Users and Posts, and have realised I needed an intermediate table. This will be linked to the service posts, and will handle the viewing of posts for different users. Using db.relationship, and hence a backref, would mean I can simply specify a user_id, and service_post id (i.e. the post(s) that each user can see). These posts can be filtered (i.e. tags → matched companies → matched companies posts).

20th August, 2019: Notifications Route and Table.

LOG: Due to modal styling and javascript issues, I have decided to simply have a route for notifications. This page will display all notifications for a user. For version 1.0.0, it will only be the reminder notification, but the types of notifications can be expanded on later. Similarly, I have decided to specify a notifications table in the database to easily track all of a users notifications.

Data Integrity and Testing Procedures:

Validation and Proofing:

I ran my project through a CSS Validator (W3 Schools), and this returned no errors.

After finishing my website's first stable version, I also sent my website through to 3 different class members and my father, and code testing feedback from all.

Checking URLs on page(s) - Main Pages:

HOME

URL	Expected Result	Actual Result	Pass/Fail

COMPANY

Outcome Check Against Planning - Why

Checks for Relevance, Accuracy and Reliability

In my project, forms the source of how the data is Created, Read, Updated, and Deleted. Throughout the development of my project, I underwent rigorous testing after each form element, relevant models.py code, and validation rules, were added.

My validation testing consisted of applying queries and raising Validation Errors as needed. The general structure was as follows:

1. Apply the relevant query -- did it return the desired value?
   - If no, check the return with a print / the cause of any errors. Fix, then reset.
   - If yes, continue:
2. If the condition is met (e.g. a value exists, != None, len()>n etc) raise a validation error.

Some examples include:

Service Registration - Check if the email is already taken.

```
def check_email(form, email):
        service = Service.query.filter_by(email=email.data).first()
        if service:
          raise ValidationError("That email is already being used. Please choose a 
different one.")
```

Service Registration - Update the current email.

```
def check_email(self, email):
        if email.data != current_user.email:
            service = Service.query.filter_by(email=email.data).first()
            if service:
              raise ValidationError("That email is already being used. Please choose a 
different one.")
```

My database testing consisted of a series of questions within a step by step process that went as follows:

3. Clear the Database of relevant entries
4. Create entries through the forms -- did it commit the desired value?
   - If no, look at commit in SQLite / the cause of any errors raised. Fix, then reset **test**.
   - If yes, continue:
5. Display through website -- Is the data being displayed correctly on the website, and is the user aware they've made a commit?
   - If no, look at commit in SQLite / the cause of any errors raised. Fix, then reset **test**.
   - If yes, continue:
6. If the user is able to update their items, does the data update update and validate correctly without changing the id? Is the user aware they've made an update
   - If no, look at commit in SQLite / the cause of any errors raised. Fix, then reset **test**.
   - If yes, continue:

7. If the user is able to update their items, does the data update update and validate correctly without changing the id? Is the user aware they've made an update
    - If no, look at what was committed / the cause of any errors raised. Fix, then reset **test**.
    - If yes, continue:

Data Access Permissions:

Flask Login and Roles

In my website, I made the decision to have two different roles: either a user, or a service. However, upon looking at the flask-login module, you can only define one type of 'current_user', meaning the system could only log one user type (a single person from the User table rather than one person from User and one from Service). This presented one main question: how would I present different pages for services so they could edit their posts etc?

Since I would only be able to define one user type in flask-login, I decided that I would need to register services into the User table before I they were entered into the Service table. This led me to the column of 'role' in the User table. A person using my site would either be registered as urole (regular user) or srole (service) in this column. This urole or srole entry would depend on the form they submit (i.e. the user-registration form would set a default value of 'urole' upon submission, and the service-registration form would set a default value of srole upon submission).

Upon deciding services needed to be first registered as a user (so that srole would be defined), I also became aware that I would need to create an extended registration page for services. This is because I wanted EVERY SERVICE to require an address, website, etc, but adding the address and website columns to the User table meant that every regular user would be required to enter these as well. Hence, when services were first registered as a user, I would simply only record the Service name and email (the email would also be checked against any existing services OR users), and make the birthday and picture entries nullable.

Furthermore, since I did need some way of recording the required values for the service, I created an extended registration that services would need to complete once they were logged in**.** I ensured that they would not be able to access any other service page until this was completed.
(**once they were logged in because of the 'one user type' flask-login rule stated further above).

For different access permissions specifically, I created my own @login_required(type='srole') decorator.
This was added above each route, and meant you were only be able to access certain pages which required your type.

Queries - SQLAlchemy

Service Post Page Query - Display all the posts that have ‘servicename’ as the author of a Service post
**servicename is a variable passed as a parameter in the route; Users see all of a specific service’s posts.*

SQL Query:	**SQLAlchemy Translation:**

`SELECT * FROM Service_Post WHERE company = ? ORDER BY date_posted DESC;`
`[parameters: (servicename)]`
	`Service_Post.query\`
	`.filter_by(company=servicename)\`
	`.order_by(Service_Post.date_posted.desc())\`
	`.all()`

Notification Page Query - Display all the notifications for a user from newest to oldest.

SQL Query:	**SQLAlchemy Translation:**

`SELECT * FROM notification WHERE user_id = ?;`
`ORDER BY id DESC;`
`[parameters: (current_user.id)]`
	`Notification.query\`
	`.filter_by(user_id=current_user.id)\`
	`.order_by(Notification.id.desc())\`
	`.all()`

List Sidebar View Query - Display a maximum of `’list_view_int’` list items on the sidebar.

SQL Query:	**SQLAlchemy Translation:**

`SELECT * FROM List WHERE user_id = ?;`
`LIMIT = ?;`
`[parameters: (current_user.id), (list_view_int)]`
	`List.query.filter_by(user_id=current_user.id)\`
	`.limit(list_view_int).all()`

Legal, Ethical And/Or Moral Requirements

Ethical and Moral: Users Data and Passwords - Flask-login and Bcrypt

When a user creates an account that includes personal information, for any platform, they expect both the platform to be secure. To make an account on a platform secure, they need an encrypted password. For password encryption, I used a python module named Bcrypt. Bcrypt is a Flask extension that provides password hashing for an app.

Password hashing is a way of securing a string. It transforms the password into another string called a hashed password. This is done upon submission, generating a new hashed password when the User signs up, and then committing the hashed password to the database. This is a one-way transformation - it is practically impossible to go to the other way due to the several severely mathematically complex hashing algorithms used within the function.

For example, in my project the user entered a password of 'password'. The hash that was generated is '$2b$12$wxezLAXq8evseinySIVzhOtbWF22hppv.oGNA/4L3CsO1f5AUQOOG'

This string is then checked at the login by with a function called 'check_password_hash that compares the form password data, and the hashed password.

Ethical and Moral - considerations for the future:

I followed all ethical and moral requirements for this project, making sure not to create any potentially offensive content. f content, there needs to be some form of filtering to minimise the damage done by individuals. This is something I hope to implement in future versions of my website.

Legal - Credits to Photos and Videos: Pixabay and Pexels

All of the following images and videos were taken from Pixabay and Pexels, with all of them being licenced under a Creative Commons license:

- default.jpg
- defaultService.jpg
- Register_bg.mp4 - self made with four free videos: one, two, three, and four
- Free to use from YouTube - Complete_bg.mp4

I created the logo and hence own the rights to this. .

Select software based on the features of the program(s)

Enables the student to effectively demonstrate skills in creating, editing and integrating media types

Tools and Features: Atom

Atom is the code editor I am using for my project. It's simple and clear UI combined with many features make it the popular program engineers use. I chose to use it for two main features:

1. It has autocomplete - this economises the time it takes to code my website.
2. It has built in Git functionality - the folder structure 'lights up' different colours for unadded, or uncommitted git changes. This makes the project easier to track.

These two features are helpful because they allow me to complete the project in the most efficient way possible, maximising what I can produce in a given amount of time.

Tools and Features: SQLite Studio

Rather than viewing tables through the built in Python IDE, I decided to use the software SQLite Studio. This is a SQLite Database manager. I decided to use SQLite Studio to easily track my database entries, letting me view all of my tables, and creating or updating existing ones. During testing, when I added users, this let me view what item had been committed upon form submission, letting me view the test results quickly and efficiently.

Tools and Features: Coolers

Coolers is an online colour scheming tool. I used this to help with the colour scheme / planning of my website. This is because it easily generates random colour schemes, and lets you select related colours or colours that work well within the chosen colour scheme. It also displays the colour hex code which I can directly copy and paste into my CSS. I mostly used it for a feature that lets me pick colours from an image. I chose the image of a sunset - an experience in its own respect.

Tools and Features: Chrome Inspector

Chrome Inspector is the 'Inspect Element' tool in Google Chrome. This tool is perhaps one of the most useful tools to use for the project. I used this because you can directly see the HTML code being used, but more importantly, see the CSS styling aspects. You can add/remove/change any CSS styling, and see these changes immediately. I also used this because of the easy padding/margin/border styling, where you can view these changes, and the styled element itself, when you hover the mouse over it. Also, there is a feature meant I could change the viewport size, letting me test the responsive features of the CSS.

Purpose End Users:

*Please see here for full documentation of the users of my site.

Complex Tools And Techniques

Producing The Outcome In A Manner That Economises The Use Of Resources

Economising Image Processing: Pillow (PIL)

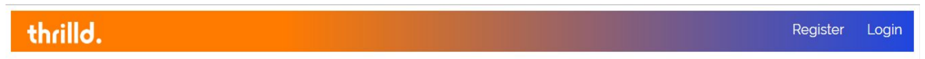
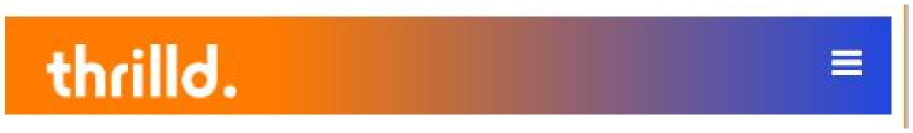
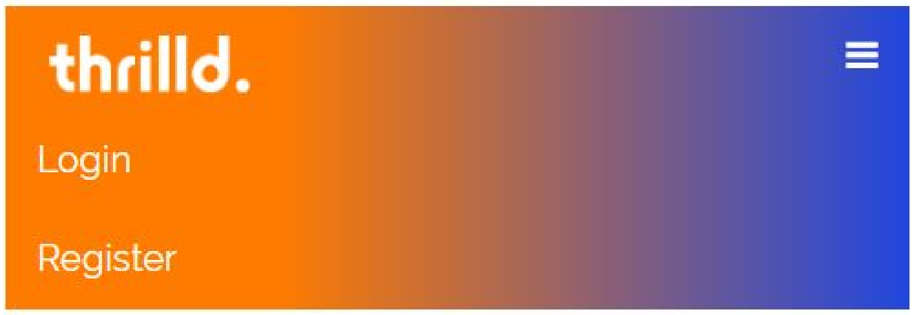
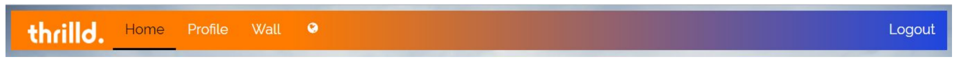
Pillow is an image-processing library I used for my website. Some of these features:

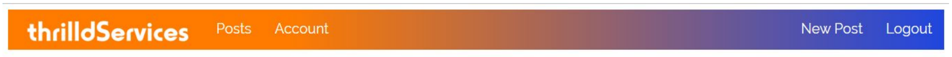
- per-pixel manipulations,
- masking and transparency handling,
- image filtering, such as blurring, contouring, smoothing, or edge finding,
- image enhancing, such as sharpening, adjusting brightness, contrast or color,
- adding text to images and more.

I used pillow for the per-pixel manipulations and ability to save posted form photos. It allowed me to specify a random name, and condense the file into a data type that reduced image size and increased total storage.

Testing Data Integrity - HTML Testing:

NAVBAR

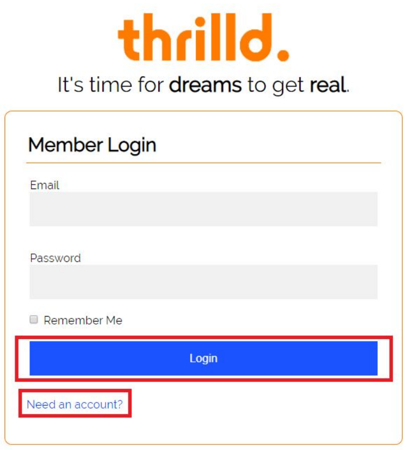
LOGIN	
 Responsive (login) - Closed  Responsive (login) - Open 	**Expected URL:** (image) - .../login Register - .../register Login - .../login **Actual URL:** **(image) - http://127.0.0.1:8080/login** **Register - http://127.0.0.1:8080/register** **Login - http://127.0.0.1:8080/login** PASS OR FAIL: **PASS**
HOME (user)	
	**Expected URL:** (image) - .../ Home - .../ Profile - .../profile Wall - .../wall WorldIcon - .../notifications Logout - .../login (and logs user out) **Actual URL:**

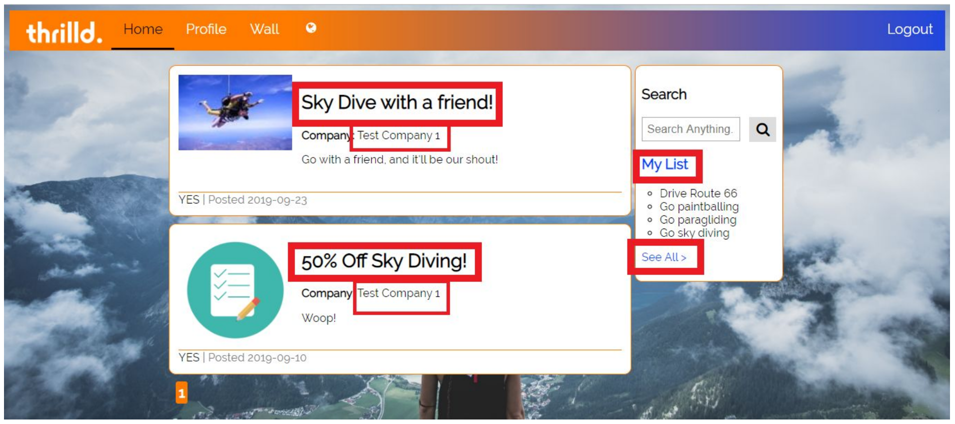
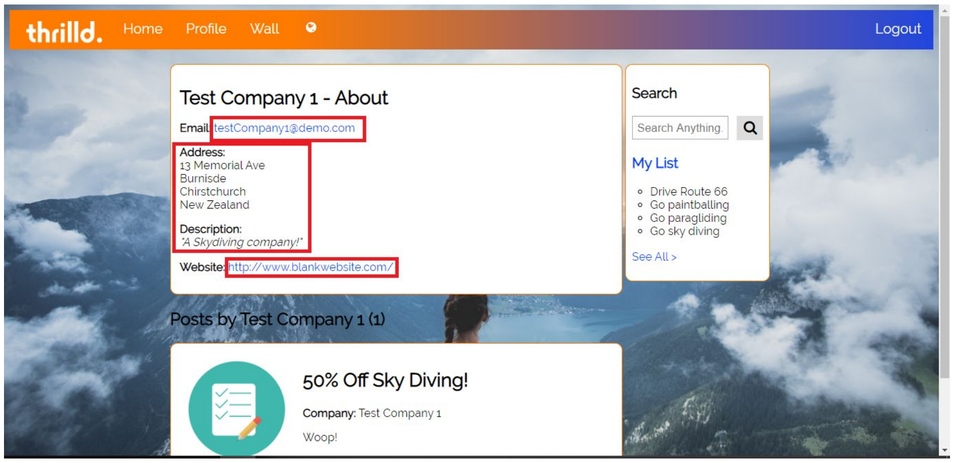
	(image) - http://127.0.0.1:8080/ Home - http://127.0.0.1:8080/ Profile - http://127.0.0.1:8080/profile Wall - http://127.0.0.1:8080/wall WorldIcon - http://127.0.0.1:8080/notifications Logout - http://127.0.0.1:8080/login	
	PASS OR FAIL	**PASS**
HOME (service)		
	**Expected URL:** (image) - .../service Posts - .../service Account - .../service/account New Post - .../post/new Logout - .../login (and logs user out) **Actual URL:** **(image) - http://127.0.0.1:8080/service Posts- http://127.0.0.1:8080/service Account - http://127.0.0.1:8080/service/account**	

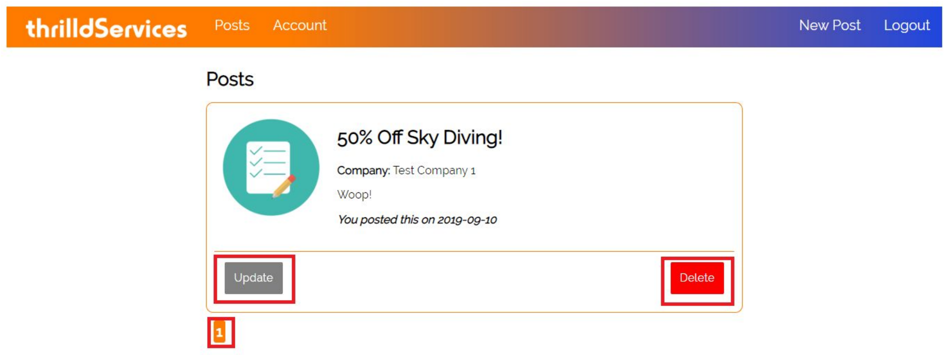
	New Post - http://127.0.0.1:8080/post/new **Logout -** http://127.0.0.1:8080/login	
	PASS OR FAIL	**PASS**

PAGES

*Links are circled in red

LOGIN		
 thrilld. It's time for dreams to get real. Member Login Email Password Remember Me Login Need an account?	**Expected URL:** Login - .../ or .../service (depending on user type) Need an account? - .../register **Actual URL:** **With service member details -** http://127.0.0.1:8080/service **With user member details** - http://127.0.0.1:8080/ **Need an account -** http://127.0.0.1:8080/register	
	PASS OR FAIL	**PASS**
Home		

 thrilld. Home Profile Wall Logout Sky Dive with a friend! Company: Test Company 1 Go with a friend, and it'll be our shout! YES | Posted 2019-09-23 50% Off Sky Diving! Company: Test Company 1 Woop! YES | Posted 2019-09-10 1 Search Search Anything.. My List - Drive Route 66 - Go paintballing - Go paragliding - Go sky diving See All >	**Expected URL:** DYNAMICALLY GENERATED LINKS: - ‘Sky Dive with a friend!’ - .../post/2 - 50% Off Sky Diving - .../post/1 - Test Company 1 - .../Test%20Company%201 My List and See All - .../list **Actual URL (order of appearance above):** **http://127.0.0.1:8080/post/2** **http://127.0.0.1:8080/post/1** **http://127.0.0.1:8080/Test%20Company%201** **http://127.0.0.1:8080/list**	
	PASS OR FAIL	**PASS**
Company		
 thrilld. Home Profile Wall Logout Test Company 1 - About Email: testCompany1@demo.com Address: 13 Memorial Ave Burnisde Chirstchurch New Zealand Description: "A Skydiving company!" Website: http://www.blankwebsite.com/ Posts by Test Company 1 (1) 50% Off Sky Diving! Company: Test Company 1 Woop! Search Search Anything.. My List - Drive Route 66 - Go paintballing - Go paragliding - Go sky diving See All >	**Expected URL:** DYNAMICALLY GENERATED: - Email (LINK) - new email to that company - Address - Description - Website (LINK) - website from database	

	Actual URL (order of appearance above): **https://mail.google.com/mail/u/0/?view=cm&fs=1&tf=1&to=testCompany1@demo.com&su=RE:+Thrilld+Promotion+Query&body=Hi+Test+Company+1, http://www.blankwebsite.com/**	
	PASS OR FAIL	**PASS**
Home - Service		
 thrilldServices Posts Account New Post Logout Update Delete 1 Posts 50% Off Sky Diving! Company: Test Company 1 Woop! *You posted this on 2019-09-10*	**Expected URL:** DYNAMICALLY GENERATED: - Update .../post/1/update - Delete .../service#delete/1 [1] (page number) - Link to same page i.e. no changes **Actual URL (order of appearance above):** **http://127.0.0.1:8080/service#delete/1** **http://127.0.0.1:8080/post/1/update**	
	PASS OR FAIL	**PASS**