# Summer 24: Task 1 – Design

This is the design portion of the documents for the product that the client (Riget Zoo Adventures) would like to be produced.

This document contains numerous diagrams and visual representations of different algorithms and other information related to the project. Starting from the top, it begins with an overview of the site: a diagram of the sitemap, which shows all the pages which will be accessible. Additionally, there's a Use Case Diagram.

It then moves on to the visual design of the web application. This includes styling decisions (fonts, colours, spacing, etc) and page mock-ups (wireframes and full-coloured).

After that, the document goes over data requirements, such as the database structure and included tables, as well as regulations we are following as part of data normalisation. Additionally, it shows an ERD (Entity Relationship Diagram) which visually shows the relationship between all the entities in storage. Lastly, it shows example SQL statements that will be used.

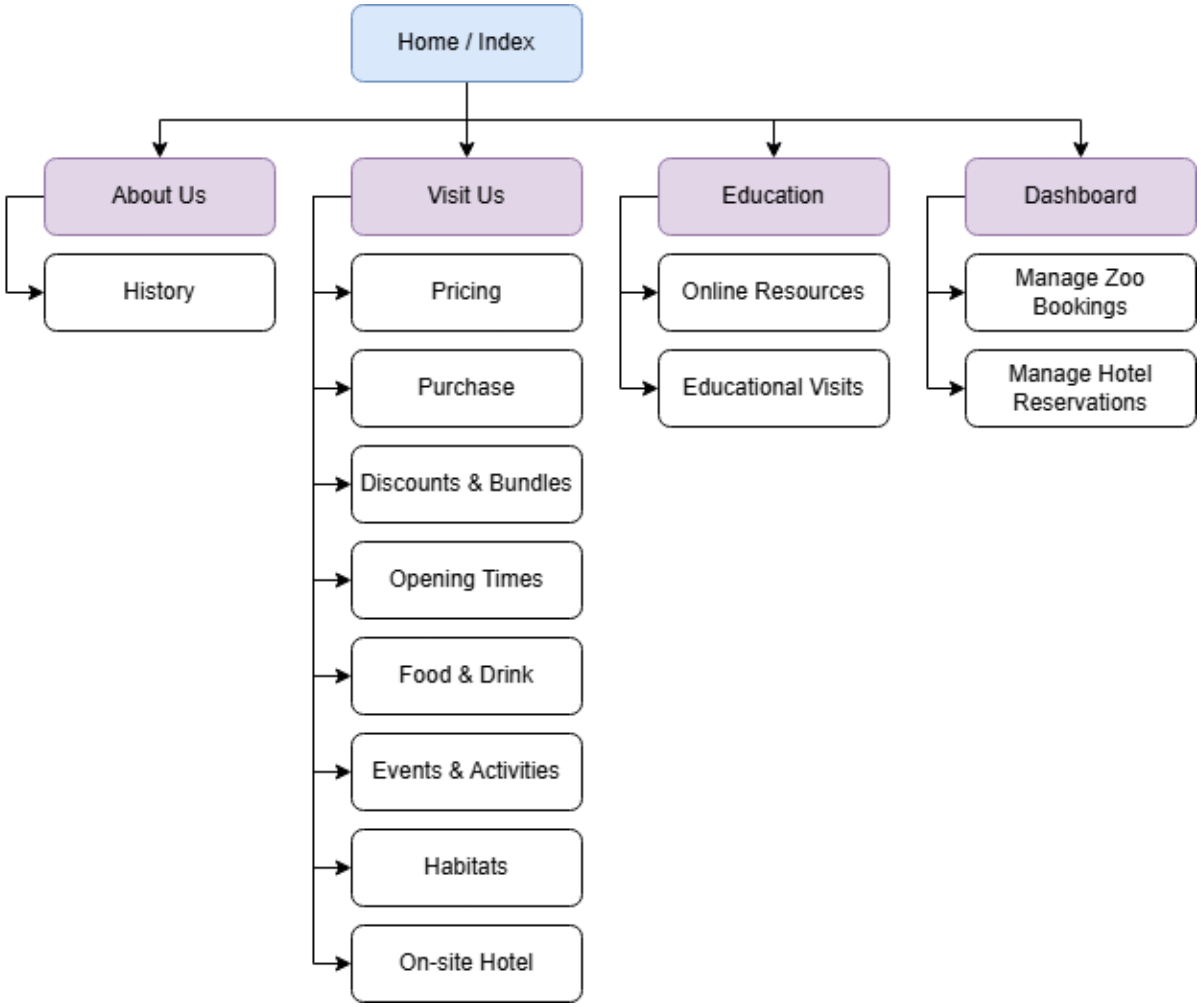
The penultimate section is about algorithms.

## Sitemap

(When I refer to "routes" in this section, I mean destinations that users can access).

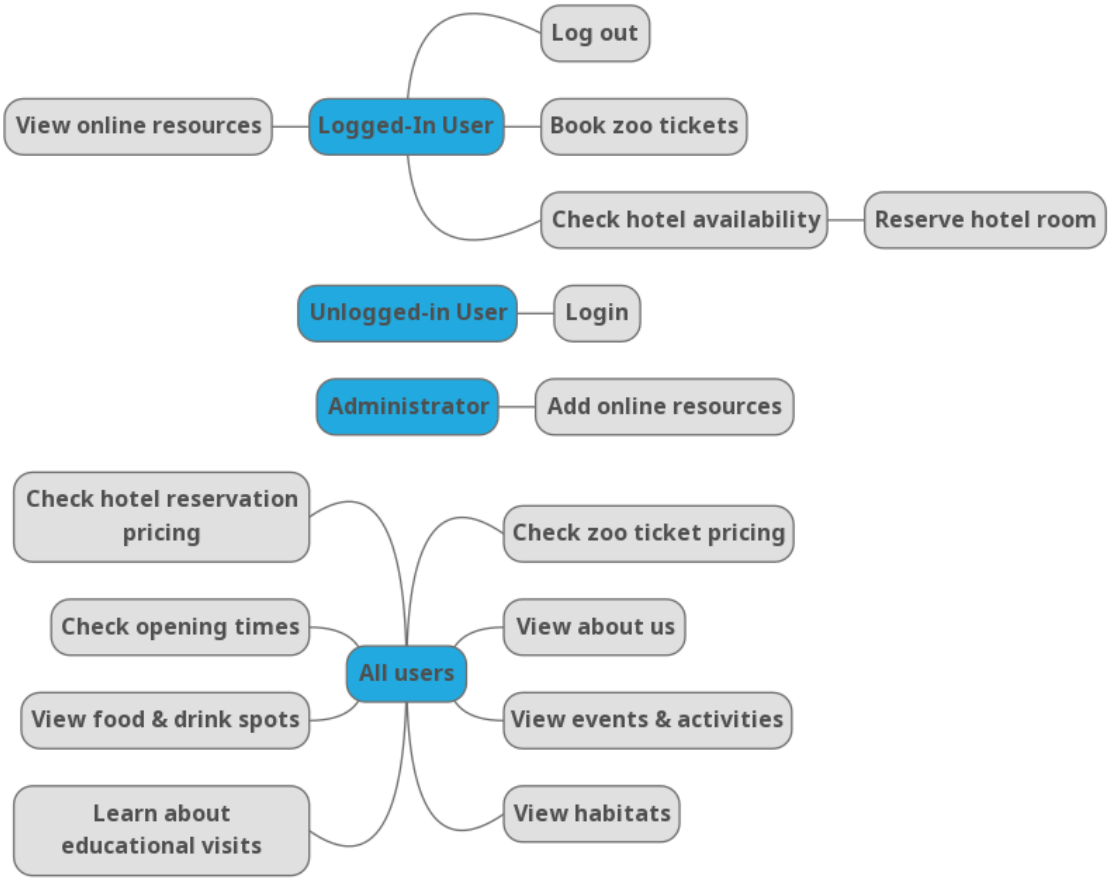
I have drawn up a hierarchical map of all the routes that the site will have available.

Additionally, I have colour coded some of the routes which show the main routes and sub-sections vs individual routes. For example, the "Visit Us" rectangle is a sub-section – like a folder in that it contains the routes under it.




## Use Case Diagram (UCD)

The use case diagram (UCD) shows exactly what users will be able to do whilst on the site.




You can see where I have said “All Users” refers to all types of users, which includes Logged-In, Unlogged-In, and Administrator.

## Visual/Interface designs

In this section, I will be going over the designs for this project and what I decided what the site should look like and why. A lot of the user interfaces have been influenced by the research I completed as part of Task 1 Activity A. You can reference to those documents to see the exact sites I took inspiration from.

### Styles

I have chosen to go for a specific style for this web application. Since zoos are primarily targeted at children, I will be trying to make the website appealing towards that audience (which could include parents and carers, too). Because of this, I will style the website in a friendly, cartoony way, but still include plenty of information so that carers are well-informed of the services Riget Zoo Adventures provides and the services the website provides.

I need to be considerate, however. I must maintain high accessibility and readability, regardless of how I style the site.

Overall, the design I’m going for is minimalist and cartoony.

### *Font*

In the initial designs, I have opted for the “Fredoka” font family. There are several reasons as to why I have come to this decision.

Firstly, the target audience for zoo website will be primarily children, or carers of children (parents, guardians, etc). For this reason, I’m trying to style the site more cartoon-y, which will appeal more to that audience.

To really emphasize the effects of this change, here’s the Fredoka font:


VS the Poppins font-family from Google:


Immediately, you can see how the whole vibe changes. Poppins gives off a much more professional vibe whereas Fredoka is less formal.

Another consideration was font sizes. I decided in the end to have the following styles:

- H1: 80px
- H2: 54px
- H3: 48px

Most regular text within the web application will default to 30px. A bigger font across the website also emphasizes the childish vibe we are aiming for. It also has the added benefit of by default providing extra accessibility for users who would usually use a bigger font size.

Finally, the last consideration to do with the font is the colouring. Generally, I have decided the following colours will be used, primarily:

- White (#FFFFFF)
    - Used whenever is most fitting. Darker background requires white font.
- Black (#000000)
    - Used whenever is most fitting. Lighter background requires black font in important areas such as the navbar.
- Orange (#F29500)
    - Used whenever emphasis is required or for the background of buttons.
- Brown (#523200) *“Saddle Brown”*
    - Used for the main bulk of text when the background is slightly off-white.

Font colouring is very important, as we don't want text to blend in with it's background and become unreadable.

### *Spacing*

Purposefully, throughout all the designs, I make sure to keep spacing and leading consistent. This is to improve readability for our users. The spacing shouldn't be too small, as this causes the text to become hard to read, but also it shouldn't be too large as this causes users to lose where they are reading and become lost.

## Page Mock-ups

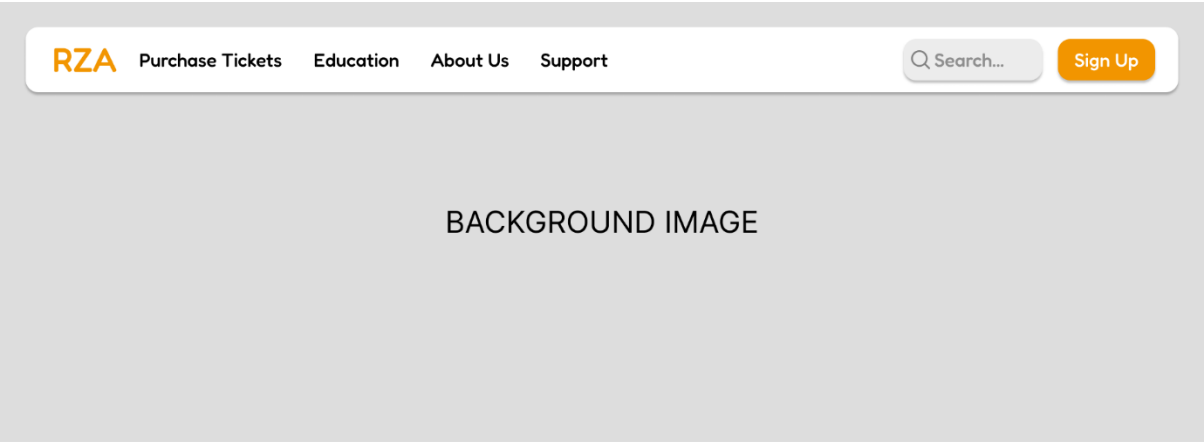
For all the page mock-ups, I've decided to make the default screen size 1920x1080, as this is the most common resolution for desktop devices.

I decided to follow a theme for each page which was to include a hero section at the top with a high-quality and high-resolution image as a background. Because of this, I had to carefully design the text overtop so that it didn't blend with the background and cause it to be unreadable.

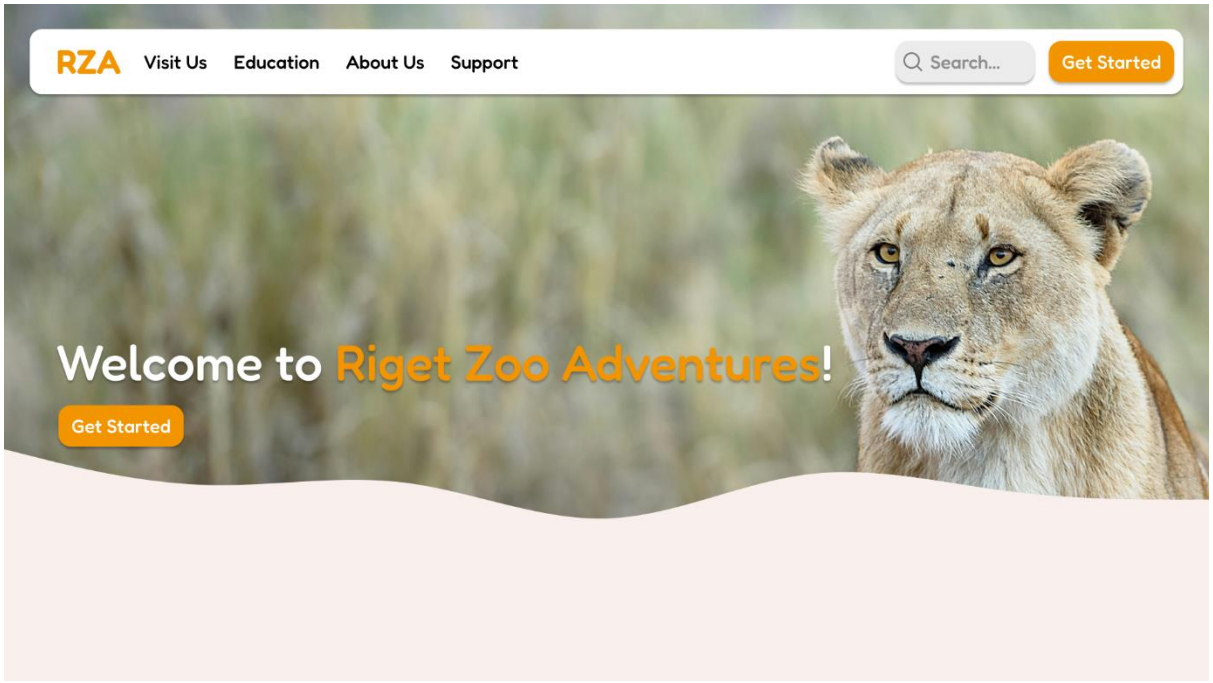
Firstly, I added drop-shadow to all the text that would be over top of an image. This caused it to stand out. In some scenarios, this was enough to make it readable and so I left it at that. For example, this is all I have done for the landing page. Other pages, however, the colours caused it to still blend in a bit. To overcome this problem, I cropped the background image so that only the portion of the image that had the text overtop was blurred. This made the text stand out even more fixing the problem. This can be seen in use on the Visit Us page.

### *Landing Page*

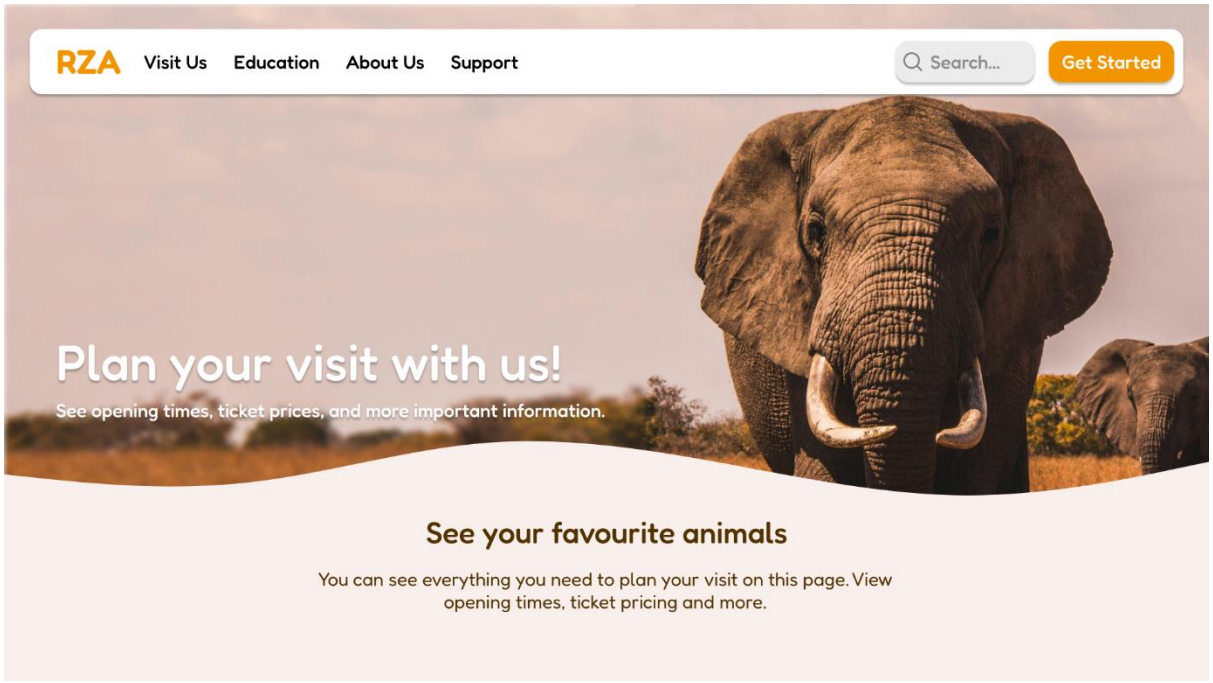
I first constructed a design with bare bones placeholders instead of actual content.



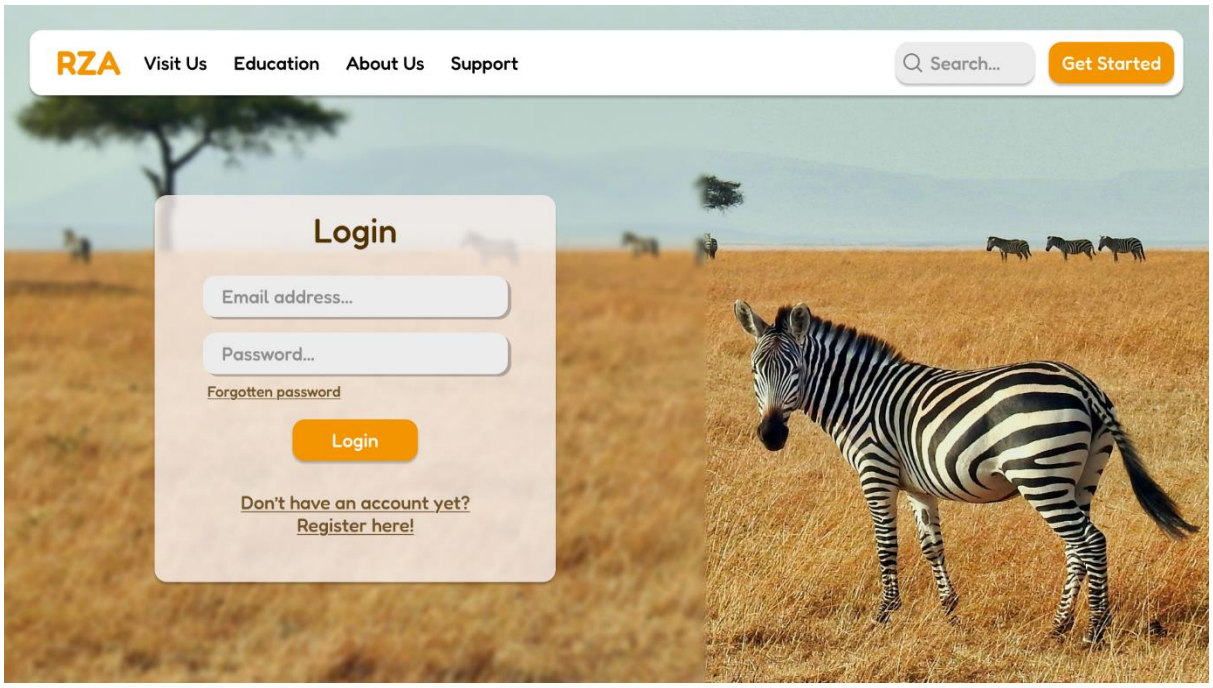

After adding said placeholders, this is the result:




*Visit Us Page*



*Login Page*




# Data requirements

## Data Dictionary

| Term | Definition |
|---|---|
| PK (Primary Key) | The column in the team that should be used to distinguish each row from every other row (a unique identifier). |
| INTEGER | A whole number, positive or negative, without fractions. (1) |
| STRING | A collection of characters, used for text. (“hello”) |
| BOOLEAN | A data type with two possible values (true, false) |
| LONG | A very large whole number, which exceeds the size of an integer. (9223372036854775807) |

## Database Structure

I had planned the design for the database structure. This covers all the data that will need to be stored within the project, including accounts, bookings, etc.

Table columns that have “PK” are marked as the primary key.

All tables must have a primary key as they need to be uniquely identified and linked together in some way. For example, every booking will need to be tied to a user account. This is why each table has an “Id” column.

*Users*

| | Description | Data type |
|---|---|---|
| Id (PK) | A unique identifier for each user | INTEGER |

| | | |
|---|---|---|
| Email | The email address of the account. Will be used for password resets and extra verification. | STRING |
| Username | Chosen by the user per their discretion. Usernames are unique. | STRING |
| Password | Password used to gain access into the account. Will be stored hashed and salted. Doesn't have to be unique. | STRING |

Users must be stored in the database so their accounts can be saved and be logged into later. Additionally, bookings and reservations must be attached to a user so that we know who has made what bookings, etc. This can be further developed and expanded into more complex systems that can be used in physical systems in hotels and park entrances.

Storing an email and username is arguably not required, but I believe that storing the email is an important attribute for each user. Storing an email brings several positives to the system. For example, users can reset their passwords. If a user forgets their password, they can request an email to their address and use a one-time link to reset it.

Also, emails are unique. Storing multiple accounts under the same email would cause many conflicts and problems within the system. Essentially, emails act as an additional unique identifier for each account.

Usernames are a way to display personal information on the users screen without a risk of unintentionally showing private information to others. A warning will be shown to users when they sign up, stating that the username shouldn't include details they wouldn't want to be shared.

Lastly, the password. The password is stored so that the user can login again when they want to. It is stored "hashed" and "salted" which means that even if someone looks at the database, the passwords will be unreadable to the human eye.

All these attributes (apart from the ID) are stored as STRINGs. This is because they are a collection of characters and none of them will be just numbers.

### *Zoo Bookings*

| | Description | Data type |
|---|---|---|
| Id (PK) | A unique identifier for each booking. | INTEGER |
| Secret | A secret for each booking. | STRING |
| UserId | The user that made the booking. | INTEGER |
| DateTime | The date and time the booking is for. | LONG |
| Adults | The number of adults. | INTEGER |
| Children | The number of children. | INTEGER |
| Used | If the ticket has been used yet. | BOOLEAN |

Zoo bookings have an ID which uniquely identifies each booking. This is stored as an integer as the database will automatically increment this value for each new entry that is added.

Zoo bookings also have a "Secret" column. This is what users will use to enter the park. Because of the nature of this value and what it will be used for, we must make sure that it is secure. It acts like a

password. To make it sure, the value that we generate will be a 32-character string consisting of numbers and letters. This way, there will 218 quattuordecillion unique combinations making guessing the secret extremely unlikely- even impossible. Additionally, when storing this value in the database, will be hashing and salting it for added security. The reason we are storing this value in this way is so that we can convert it into a QR code. This QR code will be given to users after purchasing their ticket so they can access the park super easily.

UserId refers to the user that has made the booking. This can be used for when the user decides they would like to see the bookings they've made.

Adults is the number of adults that will be attending the booking. Similarly, Children is the number of children (what counts as children is up to the discretion of the client) that will be attending.

Used is a Boolean that will be updated when the ticket has been used. When the booking is initially made, this value will initially be false. When the ticket is scanned and the user is let into the zoo, this value will be set to true, meaning it can't be used again.

### *Hotel Reservations*

| | Description | Data type |
|---|---|---|
| Id (PK) | A unique identifier for each reservation. | INTEGER |
| Room | The room in which has been reserved. | INTEGER |
| UserId | The user that made the reservation. | INTEGER |
| StartDateTime | The date and time the booking starts. | LONG |
| EndDateTime | The date and time the booking ends. | LONG |
| Adult | The number of adults attending. | INTEGER |
| Children | The number of children attending. | INTEGER |

## Data Normalisation

Whilst designing and developing these SQL table structures; I have kept in mind the rules of data normalisation.

### *First Normal Form (1NF)*

Every table is designed to contain atomic values, and each row can be uniquely identified. As you can see, every table contains an ID field, which automatically increments for each new entry, therefore making it unique. Additionally, they should all contain atomic values only. For example, if a user makes two bookings, each booking will be stored in two separate rows despite the user id being the same.

### *Second Normal Form (2NF)*

Second normal form doesn't really apply to any of our tables as there's nothing that needs to be directly defined, like for example a product of some sort.

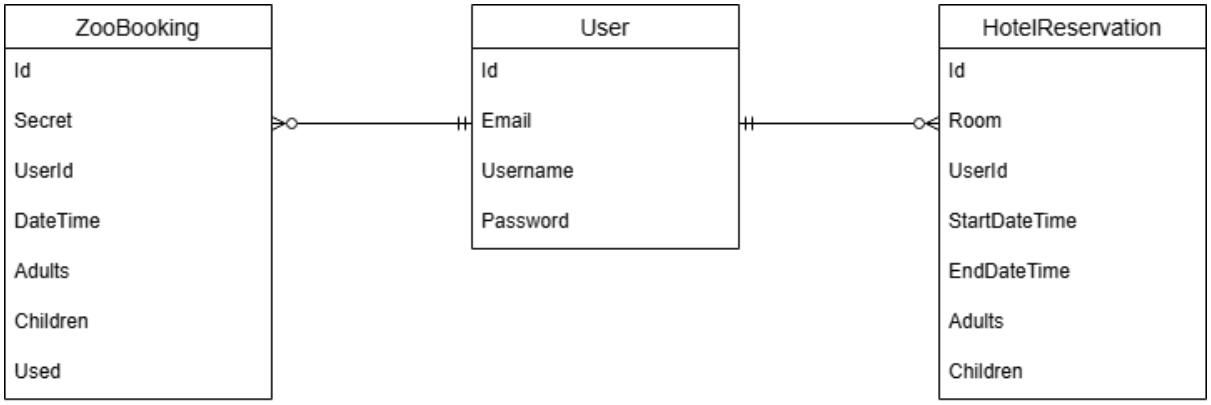
The only requirement that we do fulfil as part of 2NF is being in line with 1NF.

### *Third Normal Form (3NF)*

Again, third normal form doesn't really relate in our case. We do fulfil being in line with 2NF, however, so we do count as being compliant with 3NF as well.

## Entity Relationship Diagram (ERD)

I have designed an ERD (Entity Relationship Diagram) to show the relationship between the entities that will be used in the product.




You can see that for each user, there can be both multiple hotel reservations and zoo bookings. However, on the opposite side of that, there can only be one user for each booking. The same goes for the other; there can only be one user for each reservation.

It's impossible (with the design of the database, too) for a hotel reservation to have multiple users assigned to it, and similarly for a zoo booking to have multiple users assigned to it.

## Sample SQL Statements

In preparation for the development of the product, I have devised some sample SQL statements which will be used in the actual end-product for the client.

### *Adding a new user*

```
INSERT INTO Users (Email, Username, Password) VALUES ("test@gmail.com", "NewUser1", "TestingPass1!");
```

### *Getting an existing user's password*

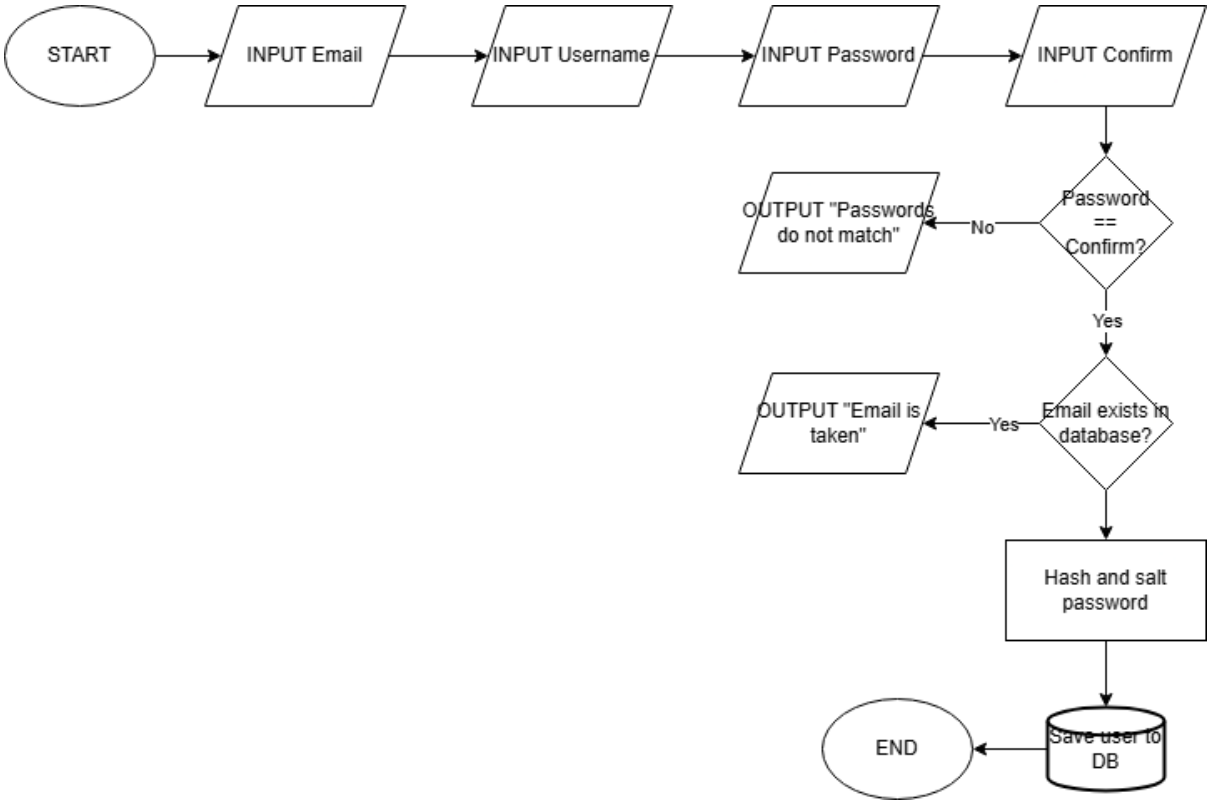
```
SELECT Password FROM Users WHERE Id=0;
```

### *Adding a zoo booking for a user*

```
INSERT INTO Zoo_Bookings (Secret, UserId, DateTime, Adults, Children, Used) VALUES ("42SjNjSHxzydKUsylBmg6fUtzcIIYSSa", 0, 1737134825008, 1, 0, False);
```

# Algorithm designs

## Registration

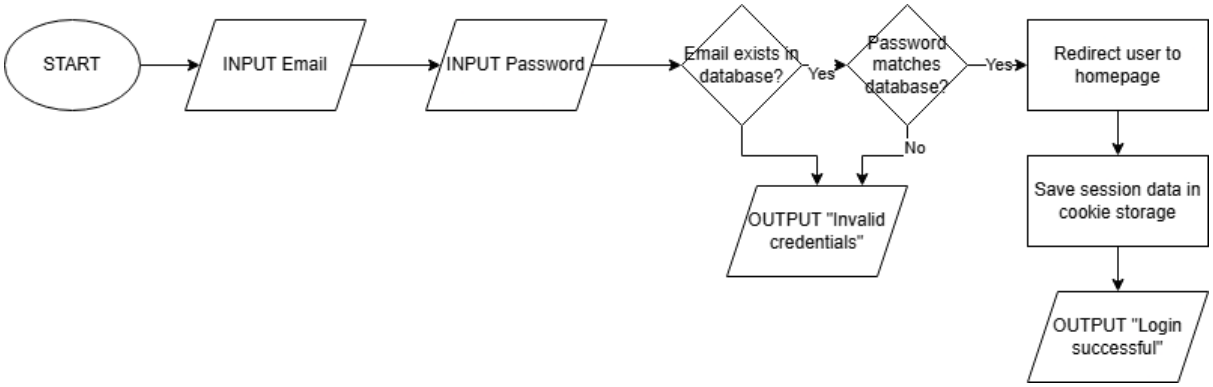



This is the flow chart that represents the process for registering new users. It starts off by asking the user for details. These details are:

- Email
- Username
- Password
- Password confirmation

We ask for password confirmation to ensure that the user knows what their password is once they've signed up. The user would have to misspell it the same way twice in a row for a mistake to be made. After this, we validate that the passwords match. If they don't, they will get an error message. If they do match, we next check if the email entered is already in the database. This would mean that there is already an account that is using that email. An error message is sent if this is the case. If not, the process finishes by hashing and salting the password and saving their details in the database.

## Login

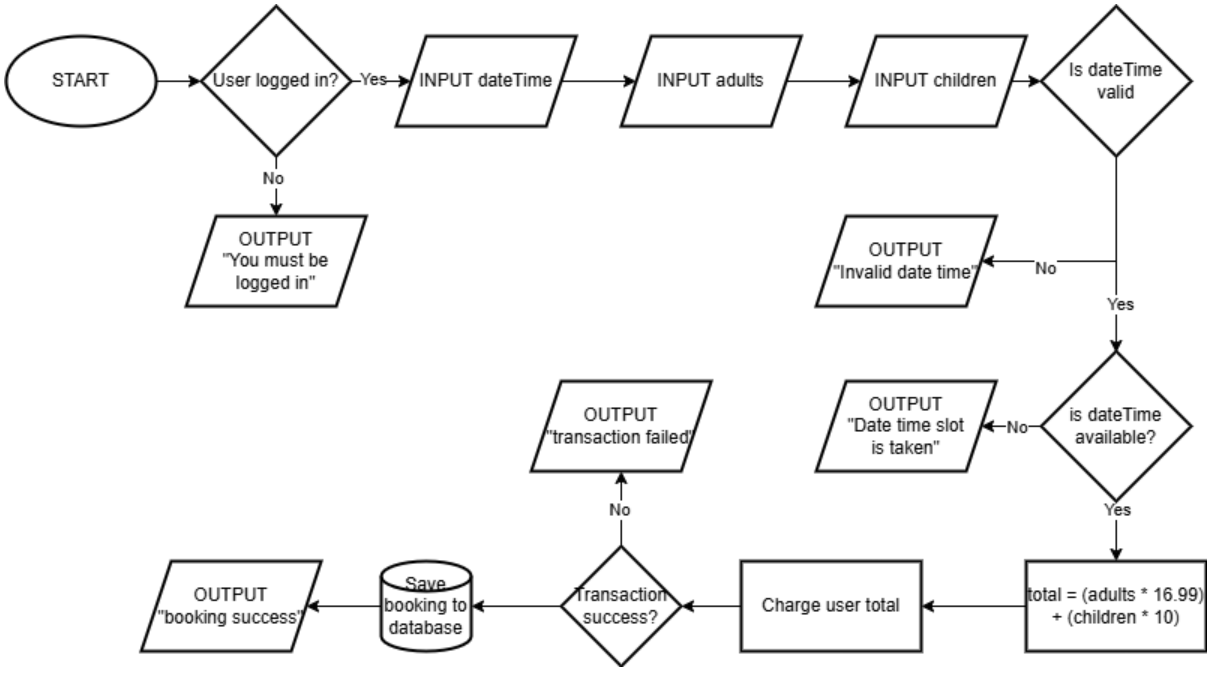



This is the flowchart that represents the steps that are undertaken during the login process. It begins by asking the user to input the following details:

- Email
- Password

After those details are submitted, we check if the email is present in the database. If it isn't, that means an account hasn't been made with that email, so it's not possible for them to sign in. Therefore, we present an error message. If the email does exist, we check the password that is tied to that email for a match with the one that was entered by the user. If they don't match, once again an error message is presented. If they do match, the user is redirected to the homepage and the session data is stored for the user. They will get a success message telling them the result.

## Zoo bookings




This is the flowchart that represents the flow of actions when a user wants to book a slot for the zoo.

Firstly, we check if the user is currently logged in or not. This is so we can track who has made each booking. If the user is logged in, they can continue and input the following details:

- Date & time for booking
- Number of adults attending
- Number of children attending

We then check if the date and time entered is valid. By "valid" we mean is not in the past, is not hundreds of years in the future, etc. If it isn't, the user is presented with an error message. If it is, it is then checked if that date is available. If it isn't, once again, an error message is produced. If it is, we calculate the total cost for the user by adding together the product of adults and cost per adult (16.99), and the product of children and cost per child (10.00). After that, we prompt the user to make the payment. If the payment is successful, we save the booking to the database and give positive user feedback. Upon payment failure, an error message is produced, and the booking is not saved.

## Test strategies

I will use a table to display the individual components that will be tested throughout development. I won't include every single specific test, as there are a lot which fall under each component.

The index column indicates the order in which components should be tested.

| Index | Component | Type of test | Description |
|---|---|---|---|
| 1 | Database initialisation | Unit Test | Tests that the server connects to the database correctly and creates all the required tables. |
| 2 | User registration | Unit Test | Tests that the registration form correctly handles the data that is sent from the front-end. |
| 3 | User registration (database) | Integration test | Tests that the data from the front-end is successfully sent to the database and inserted. |
| 4 | User login | Unit Test | Tests that the login form correctly handles the data that is sent from the front-end. |
| 5 | User login (database) | Integration test | Tests that the data from the front-end is checked against what is stored in the database successfully. |
| 6 | Zoo booking | Unit Test | Tests that the booking form data is handled correctly and validated. |
| 7 | Zoo booking (database) | Integration test | Tests that the data sent from the front-end form is handled and |

| | | | |
|---|---|---|---|
| | | | sent to the database correctly. |
| 8 | Hotel reservations | Unit test | Tests that the data from the form is correctly handled and validated. |
| 9 | Hotel reservations | Integration test | Tests that the data sent from the front-end is sent to the database to be saved correctly. |

Whilst testing these components, a combination of white-box and black-box testing should be used. I have written below the definitions of each.

## Strategy Dictionary

| Strategy | Definition |
|---|---|
| White-box testing | Approach that uses the internal logic, structure, and implementation. Testers use the context of the code and algorithms. |
| Black-box testing | Primarily focuses on the functionality without knowledge of the internal workings and structure. Testers only access the inputs and outputs of the system. |

Now I will go over what components should be tested with what strategy:

| Component | Test strategy | Description |
|---|---|---|
| Database Init | White-box | Test logic for initializing database and structure. |
| User registration | White-box | Test user input validation and registration logic. |
| User registration (DB) | White-box | Test user data insertion and database interaction. |
| User login | White-box | Test login logic (password validation, session handling). |
| User login (DB) | White-box | Test login with database (credential check, session). |
| Zoo booking | White-box | Test booking logic (availability, input validation). |
| Zoo booking (DB) | White-box | Test booking data storage and updates in the database. |
| Hotel reserves | White-box | Test reservation logic (room availability, calculations). |
| Hotel reserves (DB) | White-box | Test reservation data saving and retrieval from DB. |
| User registration | Black-box | Test end-to-end user registration, ensuring correct UI/UX. |

| | | |
|---|---|---|
| User login | Black-box | Test end-to-end login functionality without code knowledge. |
| Zoo booking | Black-box | Test end-to-end booking process as a user, focusing on UX. |
| Hotel reserves | Black-box | Test end-to-end reservation process for room booking. |