Nathan Brathwaite, Samuel Lowe, Zachary Aird, Khareen Proverb

Abstract

For this project, we chose to create an Web application for use on multiple devices.  
The purpose of the application was to provide users with the ability to manage a zoo, as well as the ability to share this information with other users.

seaside developemnt: Web-mobile report writing

Analysis of Product Design and Implementation

Contents

[1. Introduction 2](#_Toc60695531)

[**GITHUB LINK:** 2](#_Toc60695532)

[2. Application Details 3](#_Toc60695533)

[3. Managing Data 4](#_Toc60695534)

[4. Functionality 9](#_Toc60695535)

[5. Future Concerns 15](#_Toc60695536)

[6. Conclusion 15](#_Toc60695537)

# 1. Introduction

We are expected as a group to formulate a working program which provides functionality for the idea in which we conceptualized.

The report aims to look at how the functionality is brought about the appropriate software and toolkits by using spring framework and JAVA JDK.

* 1. **Background**

When the project began there was a general lack of web mobile development among the team. No previous knowledge. Time was dedicated to learning and getting an understanding of how web development functions. This was accomplished from tutorials provided in StackOverflow, as well as various other tutorial sites/videos.

* 1. **Project Brief**

The goal of this project was to create a web mobile application that would allow the user to add Animals and Staff information which would save to a database. Users - essentially creating a table.

The program requires some knowledge of Spring Boot. We were unable to deploy the program and as such must run from the IDE to see the web application in action

# **GITHUB LINK:**

[natebrate/TEST: final application (github.com)](https://github.com/natebrate/TEST)

# 2. Application Details

**2.2 Zoo Management Object**

It is important to know the functionality of the application. The main objects of the program are the database.

The database allows user to store data locally or remotely and is communicate via a special language called SQL (Structured Query Language). We decided to go with SQLite due to its easy and familiarity.

**2.2.1 Member Data Fields**

**Species.java**

private int specId;

The name member acts as an indicator for ID for the. E.g. “4000”

private String FeedingType

* Holding the value for what species feeding type is

private int quantity;

* Integer type that holds the quantity of species type

private String Descriptor;

* A string containing the description of species type, e.g. “Wet and moist”

private int LocationID;

* State to confirm the location for where the species are being held

**Animal.java**

private int animalID;

Provide a unique id for each animal   
private String name;

A string containing the name of the animal given by the management team  
private int trainerID;

* A string containing the trainer (staff) member associate to the animal

private int speciesID;

* An integer containing id for species the animal belongs to

private int age;

* integer to provide the age of the animal from a given date of birth

private String sex

* String data type containing the sex of the specific animal

private float weight;

- Afloat value that gives the animal’s weight. Float is used to represent up to 2-point values e.g. “123.12”

This also done for the Staff member

**2.2.2 Other Features**

One key feature of the program is that allows for

# 3. Managing Data

**3.1 Database**

|  |  |
| --- | --- |
| Figure 1 Database for Web Application | Database and its subsequent tables for the storing of data for the entities of the Application |

For the life of the application, all of the data objects are stored in three separate Database, based on their entity name, as follows:

**species**

**animal**

**Staff**

The software connects to the local SQLite database in the following manner in the application.properties file

*# Spring JPA.database-platform refers to the dialect file created in the previous step*

spring.jpa.database-platform=com.example.test.SQLDialect

*#Following code tells the program to update the tables automatically*

spring.jpa.hibernate.ddl-auto=update

*#says to update the tables whenever they are modified, if not present create them*

*The following code is the location of the source*

spring.datasource.URL = jdbc:sqlite:sqlitesample.db

*#tells spring to look for queries to fire on application startup*

spring.datasource.initialization-mode=always

*#DRIVER to run the database*

spring.datasource.driver-class-name = org.sqlite.JDBC

*#name sources to get in the database*

spring.datasource.username = admin

*#Database password*

spring.datasource.password = admin

To make it easier to pass this information around we created the Repository interfaces, which connects the database to the HTML interface. Additionally, Service classes were used to inject the repositories and access the data between the View pages (HTML) and the Controller classes.

A snippet of code for parsing the HTML data to the database

public class AnimalServices {

//we inject an instance of AnimalRepository via private field using @Autowired

@Autowired

private AnimalRepository repo;

//Retrieve All Elements From the Animal Table

public List<Animal> listAll() {

return repo.findAll();

}

//Save the Animal Member

public void save(Animal animal) {

repo.save(animal);

}

//Search for particular AnimalID

public Animal get(int animalID) {

return repo.findById(animalID).get();

}

//DELETE THE Animal MEMBER

public void delete(int animalID) {

repo.deleteById(animalID);

}

}

***This Service class was duplicate in an equivalent manner for Species and Staff services***

**3.2 HTML**

All of the data is stored in a local database using SQLite, Whenever the main activity of the application is created, all of the data is read from the database, and whenever ID is updated, all of the data is written to the same table. To do this, we created a method that parses HTML data (App Controller) to/from this database.

All Html template (View) or stored in a folder called templates.

A snippet of the code below. This code allows the HTML to display all entities in the table for Animal

*<*tbody*>*<!--Table to show all Staff information-->  
*<*tr th:each="animal : *${*animalList*}*"*>  
 <*td th:text="*${*animal.animalID*}*"*>*ID*</*td*>  
 <*td th:text="*${*animal.name*}*"*>*Name*</*td*>  
 <*td th:text="*${*animal.trainerID*}*"*>*Trainer*</*td*>  
 <*td th:text="*${*animal.speciesID*}*"*>*Species*</*td*>  
 <*td th:text="*${*animal.age*}*"*>*Age*</*td*>  
 <*td th:text="*${*animal.sex*}*"*>*Sex*</*td*>  
 <*td th:text="*${*animal.weight*}*"*>*Weight*</*td*>  
 <*td*>  
 <*a th:href="*@{*'/mod/' + *${*animal.animalID*}}*"*>*Edit*</*a*>* &nbsp;&nbsp;&nbsp;  
 *<*a th:href="*@{*'/remove/' + *${*animal.animalID*}}*"*>*Delete*</*a*>  
 </*td*>  
</*tr*>  
</*tbody*>*

Code Snippet showing the form to edit any Animal entity in the database

*<*form action="#" th:action="*@{*/animalsave*}*" th:object="*${*Animal*}*"  
 method="post"*>  
  
 <*table border="0" cellpadding="10"*>  
 <*tr*>  
 <*td*>*Staff ID:*</*td*>  
 <*td*><*input type="number" th:field="*\*{*animalID*}*" readonly="readonly" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Animal Name:*</*td*>  
 <*td*><*input type="text" th:field="*\*{*name*}*" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Trainer ID:*</*td*>  
 <*td*><*input type="number" th:field="*\*{*trainerID*}*" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Species ID:*</*td*>  
 <*td*>  
 <*select th:field="*\*{*speciesID*}*"*>  
 <*option th:value="1" th:text="Mammals"*></*option*>  
 <*option th:value="2" th:text="Birds"*></*option*>  
 <*option th:value="3" th:text="Reptiles"*></*option*>  
 <*option th:value="4" th:text="Amphibians"*></*option*>  
 <*option th:value="5" th:text="Invertebrates"*></*option*>  
 <*option th:value="6" th:text="Fish"*></*option*>  
 </*select*>  
 </*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Age:*</*td*>  
 <*td*><*input type="number" th:field="*\*{*age*}*" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Sex (Gender):*</*td*>  
 <*td*><*input type="text" th:field="*\*{*sex*}*" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td*>*Weight:*</*td*>  
 <*td*><*input type="number" step="any" th:field="*\*{*weight*}*" */></*td*>  
 </*tr*>  
 <*tr*>  
 <*td colspan="2"*><*button type="submit"*>*Save*</*button*> </*td*>  
 </*tr*>  
 </*table*>  
</*form*>*

This here is an example of user input being take, this is sent to the Repository class to communicate with the database where data is grabbed and compared to

HTML object is used to write the data to file, starting with the header and opening label. Reading the data and manipulating it into the correct structure is more difficult. We need to read all of the data from the file, this why we have a Repository class that can take this data to the database.

The AppController.java is what is used to call webpage pages and get data from the database via the Repositories and Services and display on the View (Html pages)

# 4. Functionality

**5.1 Creating Animal**

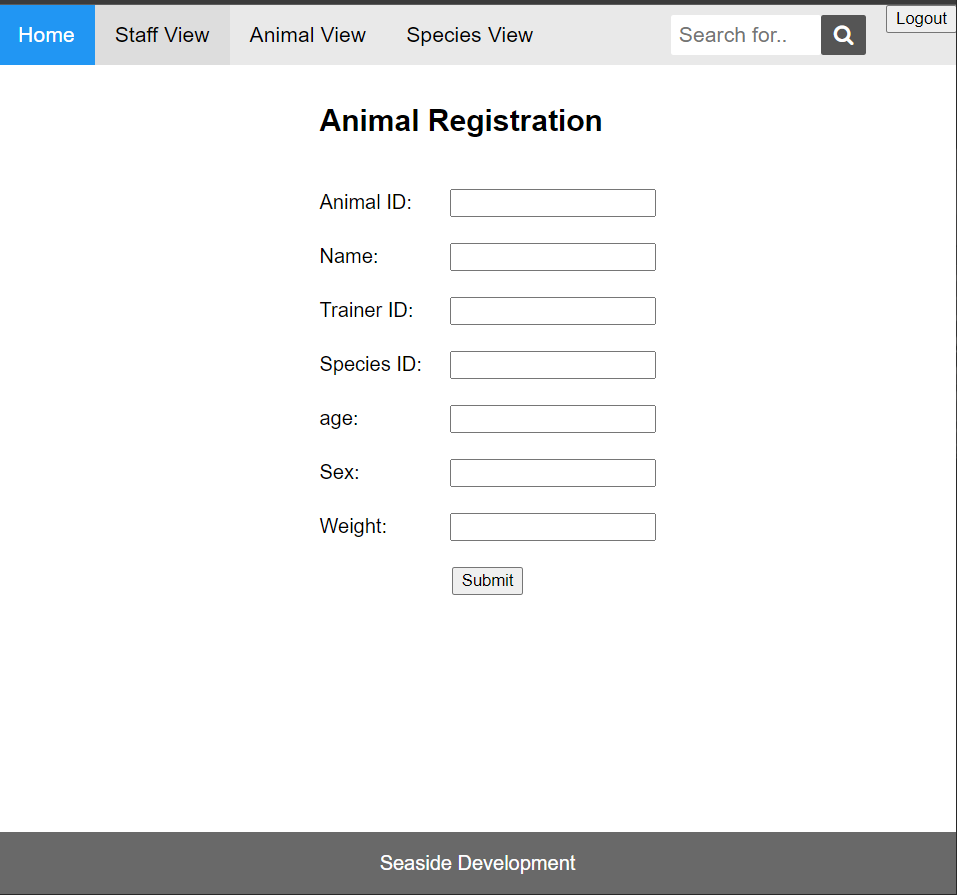
When a user presses "Animal View" from within the main activity, a new page appears allowing them to select create a new animal, giving them the following page

Figure 2 Animal Registration Page

This page was created using an HTML template for the application. The page allows for a new, update or deletes of an Animal. If the user types a unique ID in the ID number field all corresponding information will be display. It is a convenient solution as the animal table is already containing a lot of visual information.

Once a selection has been made and the new details have been entered, it is saved in the animalInfo database

Editing

All the databases can be updated view similar looking pages

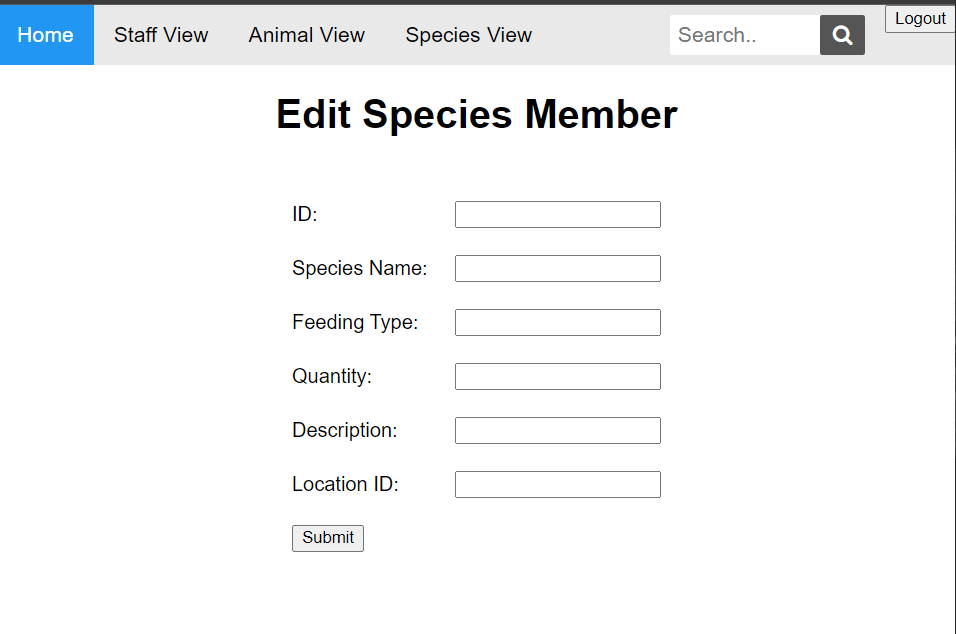


Figure 3 Image showing the layout of all the Edit pages

When a user clicks edit on the view page (refer to figure 5). This user can then edit the entity along with all its attributes

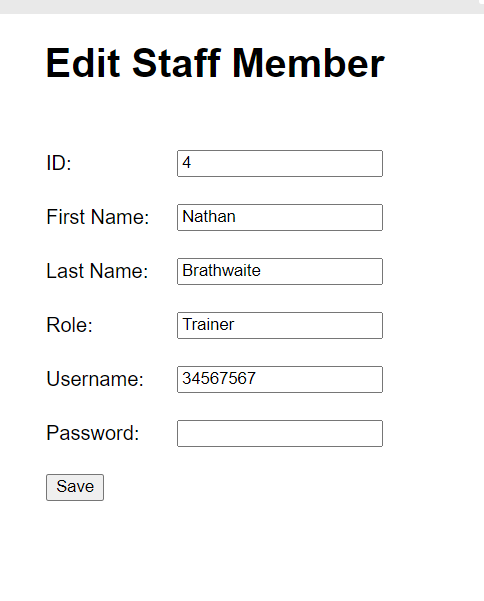


Figure 4 Figure showing what happens when the edit button is selected on an entity

The above table showcases what happens when a user selects a possible entity displayed on StaffView page. The Content belonging to the entity is displayed an editable once permitted.

**5.2 Users (Staff)**

A user can modify all staff on the same screen. This is more flexible than going to another screen. The user (staff) screen does not hold as much data like the animal screen and as such this method was implemented for ease of users.

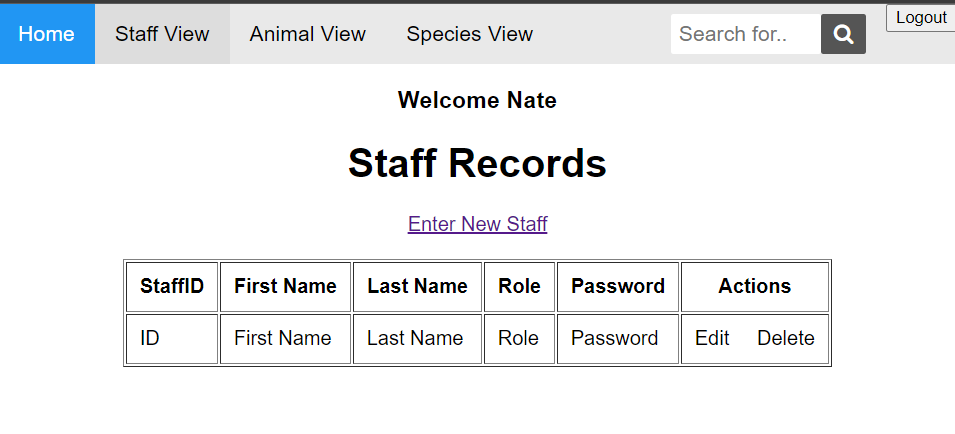


Figure 5 User (Staff) Management Screen (Empty database)

The screen has two buttons in the activity column the screen button allows the user to modify, the red trash can allow for the user to be deleted. The Add new allow for a new user to be added to the console and further saved to the database

The user currently can scroll and find user they wish to find as the size of the database is small and allowed for this method. Capturing the data and sending it to this database is of utmost importance, refer back “3. Managing Data”.

The tables used for animal, species and staff are created by entities in the code itself. Example of this entity is shown below.

Staff.java

import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
@Entity *– Let the program know that this entity and to create a table with the following attributes*

@Table (name = “staff”) *– Gives the table this specific name*  
public class Staff *{…….}*

This snippet of code is performed at runtime and *only if and only if* the table does not exist is this then executed.

Once the user has completed adding User and animals and clicks the appropriate save function the data is sent to corresponding the database.

**4.2 Viewing**

When a user taps Staff and animal pages the user can view the data of the respective database

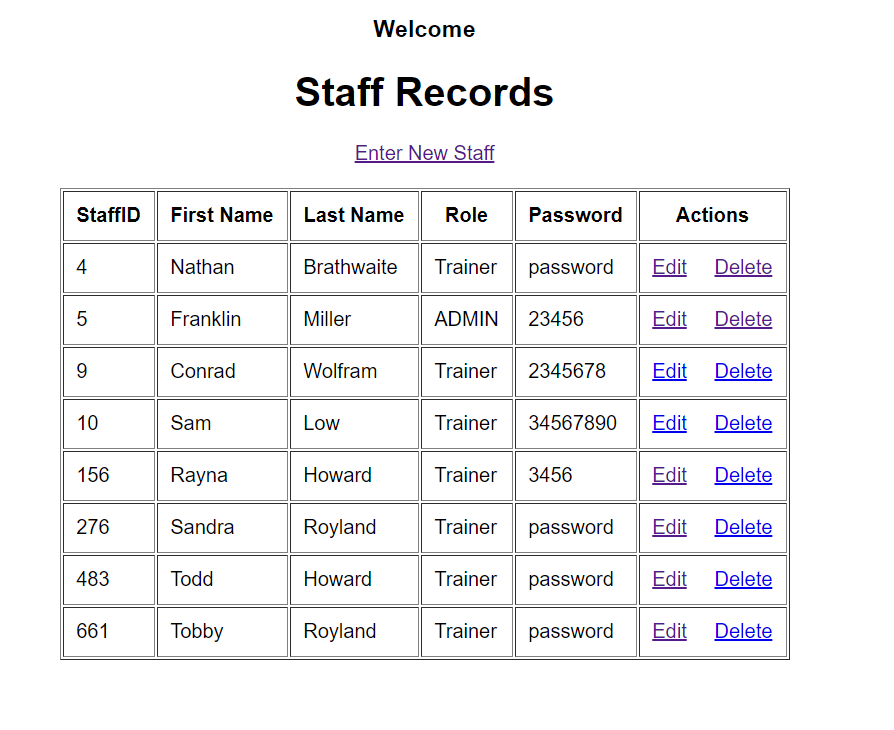


Figure 6 Figure showing database content displayed at the view (Staff page)

This design process is repeated with the other View pages

For staff refer to “5.2 Users (Staff)” for the staff view.

# 5. Future Concerns

5.1 Errors/Bugs

* When creating user or animals there is a glitch that some data is represented properly.
* The pages sometimes failed to load properly and launching the web app was a nightmare. A workaround was found by using Tomcat Apache Server to handle webpage usage.
* The application does not have the most robust system to protect against duplication and as such duplication are highly possible in its current state.
* No alerts are given for certain functions and the system is not a backup event that host device fails
* the device receives updates for the database.
* A virtual machine is a better method of implementation for the future so the remote host can keep the webpages alive 24/7.

**5.2 Data Transfer**

Unfortunately, we were unable to complete this functionality before the end of the project timeframe. It took a significant amount of time trying to understand and implement the process for connecting devices, as well as figuring out how to manipulate data between the mobile and web interface.

Due to the limited capacity of MySQL and the inability to access a virtual machine and/or cloud computing, we were unable to facilitate a remote database, therefore, negating the possibility of the Web App and Android application communicating seamlessly.

**5.3 Archive and not delete**

Currently, there is no functionality to ensure data is an archive and not deleted

# 6. Conclusion

It was an exciting project to work on and there is a lot we learnt from it, above and beyond its original scope. We were able to study and train ourselves on the development in an environment that was almost completely foreign to us. Additionally, we believe we have been reasonably successful with.

Although we were not able to complete some of the proposed functionality, the functionality related to the core purpose of the application is working as desired. We believe there is still a lot of potential for this application, and hopefully will continue development in the future. We will add an IOS companion application to the development process.

It is of hope that we can access a remote database in the future allowing for the seamless communication between the various components.