COM519

Avanced Database Systems (COM519) - Assessment 1: Proof of concept data-driven full stack web-application

Link to github repository: https://https://github.com/dariocruz96/COM519/
Hosted on Cyclic: https://animal-crossing-fish.cyclic.app/
Contributer: **Dario Cruz**

How to Setup the Project

- Download the zip file of the project.
- Unzip it wherever you want.
- Open the folder using Visual Studio Code.
- Open new terminal.
- Run

```
npm install
```

- After that run

```
npm run dev
```

- In the terminal you will see Example app listening at: The URL of the application. Press ctrl and click on the URl to start the application.

Introduction

For this project it was required to create, test, and deploy, a proof of concept data-driven full stack web-application so I wanted to do something that would make sense for me.

I decided to go with a game that is played around the globe and that includes me, Animal Crossing New Horizons for Nintendo Switch. Animal Crossing has around 240 000 players online daily and has sold around 40 million dollars since its release making it one of the most successful games for the console.

While playing the game, there is always something that bothering me, **fishing**! I never know if the fish I just got is worth the space in the inventory or if I can just throw it out and keep the good ones (£££) to sell later and the only way to find out is going to the store and try to sell them, but even then, I would have to sell them individually to know their exact price.

In the end of the day the same problems remained:

- I can't know the price while fishing
- I have to go to the store and them sell individually to see the price
- I can't remember the price later
- I make low money due to carrying worthless fish

There must be a better way!

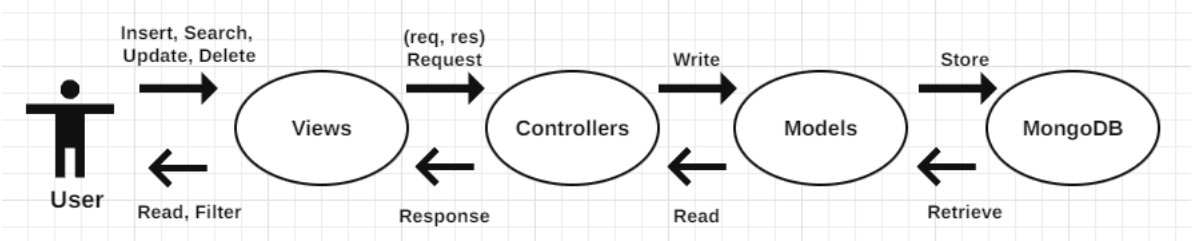
Using the animal crossing new horizons' s dataset from Kaggle.com I created a web-app that allows players to check fish prices while they are fishing so they know if it is worth to keep them.

System Overview

I used mongodb atlas to help me with my non-relational database and fish collection (Kaggle dataset) for data.

This web application structure is based on Model, Views and Controllers (MVC structure), where the user will interact with views (web pages), the view pages will send requests to controllers, the controllers will write in the models and the models will store data in our database (MongoDB), going the other way around, database will retrieve data to the models, the controllers will read from the models sending a response to views, which will be visible to the user.

Diagram



MODELS
To connect to the database used mongoose, reading and writing the properties needed for the app and using some data validation that will make sure the user inputs the correct format to be inserted in the database.

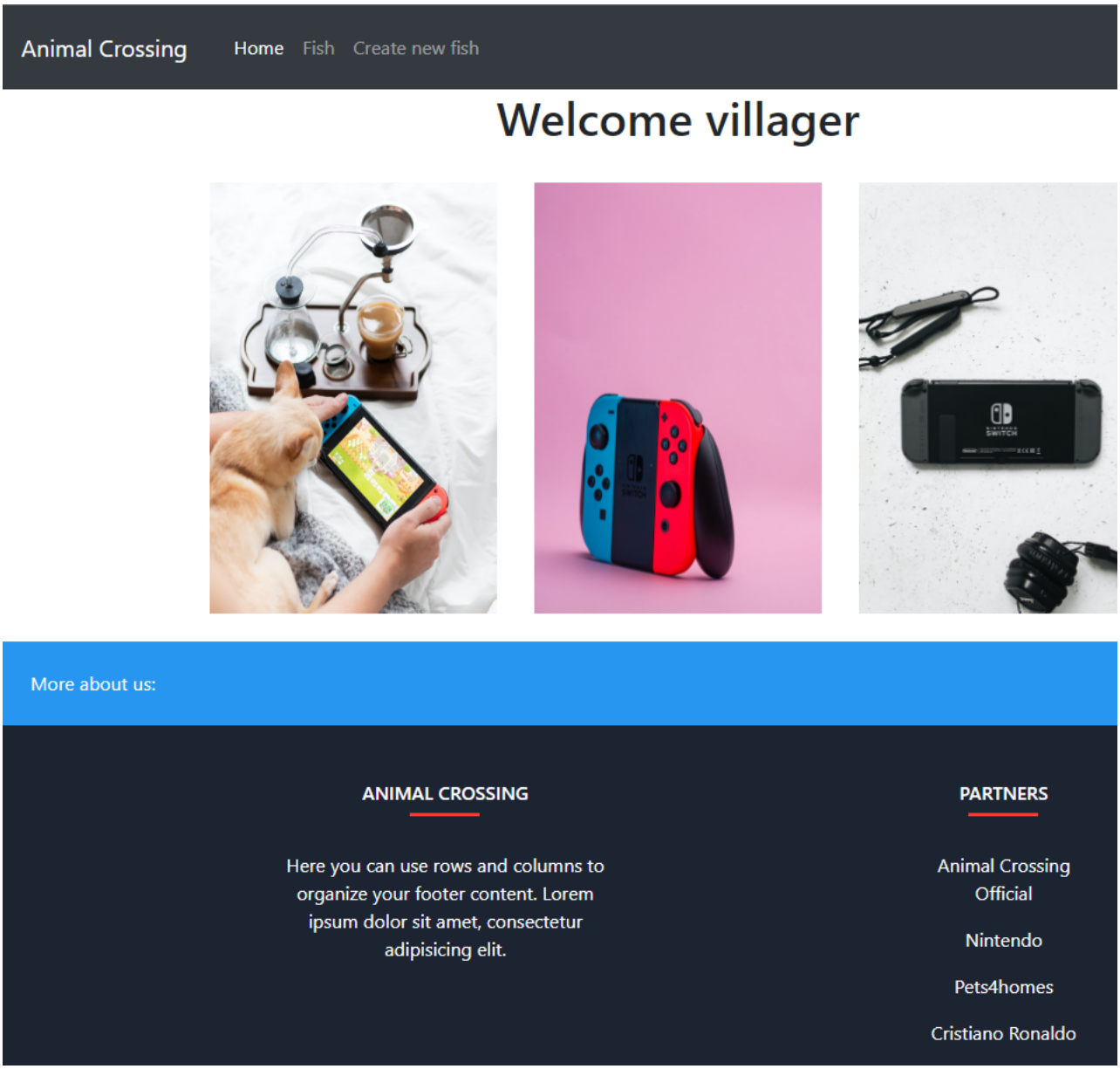
```
const mongoose = require("mongoose");
const { Schema } = mongoose;
```

```
const fishSchema = new Schema(
  {
    Number: {
      type: Number,
    },
    Name: {
      type: String,
      required: [true, "Name is required"],
    },
    Sell: {
      type: Number,
      required: [true, "Price is required"],
    },
    SpawnRates: [String],
    Where: String,
    Shadow: String,
    //"Total Catches to Unlock": String,
  },
  { timestamps: false }
);

module.exports = mongoose.model("Fish", fishSchema);
```

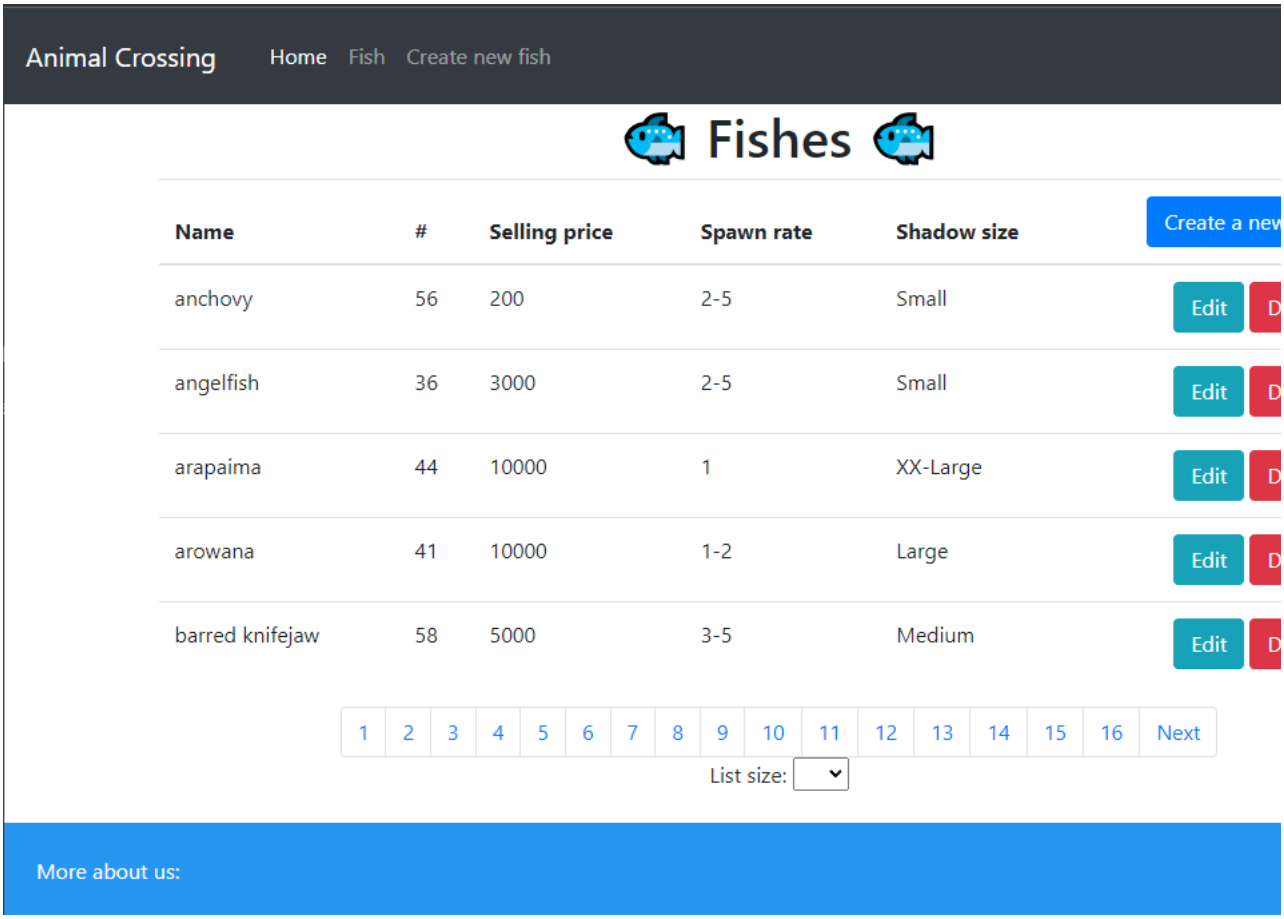
Home page

Simple home page with header menu connections to home, fish and create a new fish pages, three images and a footer with some hyperlinks.



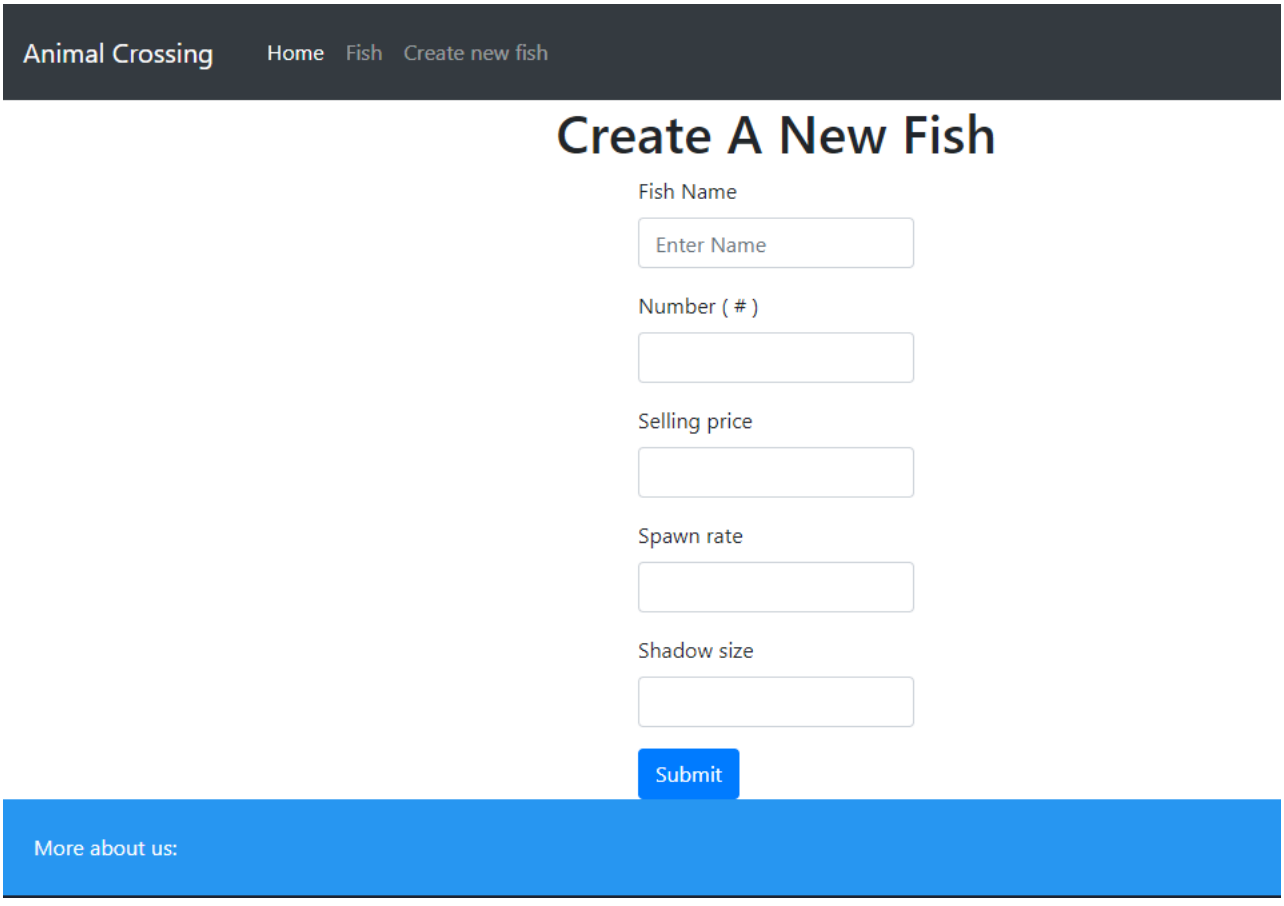
Fish page

Page showing a list of fishes details such as name, selling price, spawn rate and shadow size, have a create, edit and delete fish buttons.

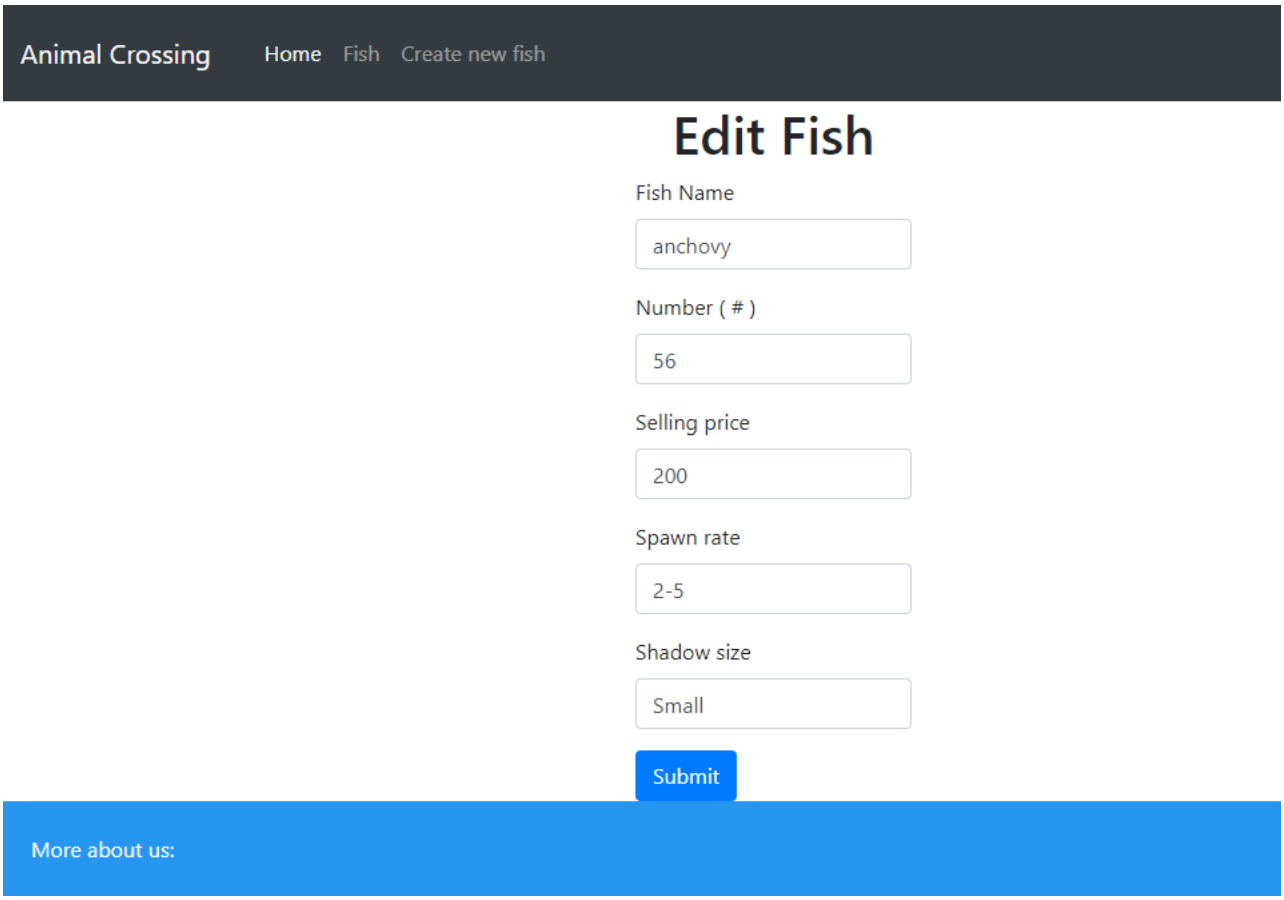


Create fish page

Page with inputs to create a new fish and add it to database.



Edit fish page
Page with inputs to change a fish details, showing the existing details.



CONTROLLERS

To be middle term to the database and the views, the controllers help us receive and sent the information we need. We got some code retrieving, deleting , updating, editing and creating records.

```
exports.list = async (req, res) => {
  const limit = parseInt(req.query.limit); // Make sure to parse the limit to number
  try {
    const page = parseInt(req.query.page); // Make sure to parse the skip to number
    const skip = (page - 1) * limit;
    const totalFishes = await Fish.find({}).count();
    const fishes = await Fish.find({ page })
      .sort({ Name: 1 })
      .skip(skip)
      .limit(limit);

    res.render("fishes", {
      fishes: fishes,
      totalFishes,
      message: req.query?.message,
      limit,
      page,
    });
  } catch (e) {
    res.status(404).send({ message: "could not list fishes" });
  }
};

exports.delete = async (req, res) => {
  const id = req.params.id;
  try {
    await Fish.findByIdAndRemove(id);
    res.redirect("/fishes");
  } catch (e) {
    res.status(404).send({
      message: `could not delete record ${id}.`,
    });
  }
};

exports.update = async (req, res) => {
  const id = req.params.id;
  try {
    await Fish.updateOne({ _id: id }, req.body);
    const fish = await Fish.findById(id);
    res.redirect(`/fishes/?message= ${fish.Name} as been updated.`);
  } catch (e) {
```

```
    if (e.errors) {
      console.log(e.errors);
      return res.render("update-fish", { errors: e.errors });
    }

    res.status(404).send({
      message: `could not update fish: ${id}.`,
    });
  }
};

exports.edit = async (req, res) => {
  const id = req.params.id;
  try {
    const fish = await Fish.findById(id);
    res.render("update-fish", { fish: fish, id: id, errors: {} });
  } catch (e) {
    res.status(404).send({
      message: `could not find fish ${id}.`,
    });
  }
};

exports.create = async (req, res) => {
  let fish = new Fish({
    Name: req.body.name,
    Number: req.body.number,
    Sell: req.body.price,
    SpawnRates: [req.body.spawn],
    Shadow: req.body.shadow,
  });

  try {
    await fish.save();
    res.redirect(`/fishes/?message=${req.body.name} has been created`);
  } catch (e) {
    if (e.errors) {
      return res.render("create-fish", { errors: e.errors });
    }
    return res.status(400).send({ message: JSON.parse(e) });
  }
};
```

Key Design Decisions

I decided to go with a really simple, user-friendly design, not a lot of options or links, not a lot of images, simple to use and no explanation need at all.

Database Design

I used a single collection because it provided me with all the information I needed and even "ignored" a few properties which I didn't find it important to be seen. Mongoose prived a unique ID for each record and so it became very simple to interact with the database.

```
_id: ObjectId('63ca4771ab31aba4345b0794')
Number: 44
Name: "arapaima"
Sell: 10000
> Where: Object
Shadow: "XX-Large"
Total Catches to Unlock: 50
SpawnRates: "1"
> Rain: Object
NH Jan: "NA"
NH Feb: "NA"
NH Mar: "NA"
NH Apr: "NA"
NH May: "NA"
NH Jun: "4 PM - 9 AM"
NH Jul: "4 PM - 9 AM"
NH Aug: "4 PM - 9 AM"
NH Sep: "4 PM - 9 AM"
NH Oct: "NA"
NH Nov: "NA"
NH Dec: "NA"
SH Jan: "4 PM - 9 AM"
SH Feb: "4 PM - 9 AM"
SH Mar: "4 PM - 9 AM"
SH Apr: "NA"
```

Security and Scalability

I didn't provide any security to the web app which is something that can be done if this would be taken forward to a full project or a complete market ready app by creating user authentication and restrict access to some features. Going forward with this proof of concept web application could be creating a full Animal Crossing help tool, using all the collections and properties provided in the Kaggle dataset and provide help with any aspect of the game to players. Doing a better styling would improve the web app a lot as well.

Conclusion and reflection

Deploying this project in Cyclic makes it accessible to everyone and MongoDB/Mongoose allowed me to easly work with database, which following the MVC structure made it really simple and straight forward to create the web application. Working with Git allowed to to go back in work a few times to confirmed to work versions and even work from different PCs and places. While creating the first funcionalities gave a bit of work, the more there was to do the simpler it became, getting used to the structure flow and having some code as base made it possible to quickly add features and funcionalities.