Project approach:

Sample user stories:

As a cooking enthusiast who lives in (Country), I want to share and view recipes from my Country/other Countries
As a cooking enthusiast who lives in (Country), I want the ability to edit/develop recipes from my Country/other countries
As a disciplined foodie who lives in(Country), I want to access recipe groups such as vegan recipes from my Country/other countries.
As a busy mom I want the ability to source quick recipes from my country/othercountries
As an allergen sufferer I want the ability to find allergen free recipes from around the world
As an intrepid traveller visiting (country) I want to find indigeneous cuisine

Wireframe presented in the additional info folder:

Database management system:

Mongodb was chosen as the database management system for this application. It was felt that a recipe based application would not necessarily handle a lot of complicated querying, but would be prone to an unstructured format as different users inteact with the application. Mongodb uses the json structure format (bson) which is useful for storing large amounts of unstructured data. JSON format lets you nest records one within the other and also allows for different records to have different fields which is inevitable with different users manipulating data in a web application.
The Database was created first and the application then built around it. A basic DB was set up using the mLab Mongo cloud based application. 2 collections were established, one for the courses(categories) and one for the actual recipes. The category key value pair is added to the recipes collection linking the recipes collection to the category collection akin to a foreign key. The categories is also used to group similar or related recipes for example main dishes, desserts, etc. Several records were added to start off the database.

Python structure:

The various modules were installed (flask, flask-pymongo, os (also pymongo to use constants when sorting) and the app initialised.
The libraries were then imported from these modules(Flask, request object, url_for function, etc).
The application was configured to connect to the database and an instance of pymongo created.
A default route was established, enabling login and access to the application, directing to the home page. Login and index views were established.

A get_recipes route was then set up to find the recipe documents and display in the recipe_html view as as a summary list. (A similar function was estabilished for the categories collection). The summary view displays the recipe name and respective category and contains buttons to call the edit and delete functions(namely url_for edit_recipe and url_for delete_recipe respectively).

A route was established to display an html form (add_recipe) which displays the document key value pairs and in which values can be written into the field. Another route (insert_recipe) inserts the data into the database. The add-recipe route returns a form bound to the categories and this form calls the insert_recipe function to insert data or the returns the home page if the operation is cancelled. When the data is inserted the insert function redirects to the recipe summary list (url_for(get_recipes)).

The add_recipe form initially contained a switch (no yes) for the vegetarian and vegan options. However it was noticed that information is only posted when the field is switched on (yes). As the vegetarian field needs information to be sent when the switch is off (as non vegetarian dishes are returned in the other recipes view when off), it was abandoned for the use of Yes or No for these fields. Also it means either Yes or No will always populate these fields (no blank field which may lead to confusion).

It was also considered to use checkboxes for the allergens field (with multiple options). As this application is used in a global context there may be allergens in other countries not covered in the checkbox list. Also using multiple checkboxes does not guarantee correct entry of data. Thus it was decided to let this be a text input field with some instruction given as how to enter data. Either enter None Known or Contains, Egg, Gluten, Lupin(note: alphabetic sequence). Streamlining this data is important as it is used in charting. Contains Gluten, Egg will give a different answer than Contains Egg, Gluten and render a different pie slice. To get the total recipes with Egg, Gluten the user needs to visually recognise that there are 2, (in this context), as both the illustrated are the same group. One could argue that this is not a major issue as long as the user can pinpoint egg and gluten allergen recipes when exploring the appication, which they can. Also with the ability to quickly edit recipes, an astute observer could easily change Contains Gluten, Egg to Contains Egg, Gluten, thereby rendering 1 pie slice with a value of 2. However more work needs to be done on this fied with future development , incorporating more complex code, outside the scope of this project.

As with the add_recipe route an edit_recipe route was established to present the form data for editing. The recipe _id is targeted to return the data and is a parameter passed into the function. In order to find a match in mongodb, the ObjectId is imported from the bson.objectid library and used to convert the id into a readable format in mongodb. So fundamentally this function fetches the recipe that matches this recipe ID and redirects to the edit_task html template, which in turn calls the update_recipe function and redirects to the recipe summary list on completion. The update_recipe route also passes in the recipe_id as a parameter. A cancel button allows the operation to be cancelled and redirects back to the recipe list.

The delete_recipe route (also passing in recipe_id as a parameter) simply redirects to the recipe list view where a delete (ok or cancel) alert option is presented. Ok deletes and redirects back to the same page (so the user can verify deletion). Cancel keeps the recipe and redirects back to the same page.

The paragraphs above describe the routes and views to enable CRUD operations and the same procedure was used for the category collection, with the category_id being passed into route functions for edit, update and delete.

In the creation of the views above template inheritance was used enabling the principal of DRY to be adhered to. One base html (the parent) extends to all the other child templates using the standard syntax.

The view_recipe route (with recipe_id parameter passed in) allows the full details of a particular recipe to be located and viewed. It renders the view_recipe html which presents an accordion style element. Click on the field name to view the value. At the bottom of the element there is a button to call the upvote function and a button to return to the home page.

The upvote route (with recipe_id parameter passed in) allows a particular recipe to be upvoted. This function fetches the recipe that matches this recipe ID and updates the upvote field by one using the mongodb $inc operator. It then redirects to the home page with a flashed message indicating a successful upvote. The only time a user can upvote is when in full recipe view.

This one was took some time to sort out, as the form was posting a string value to the database field, giving an error on increment, as $inc cannot increment a string. A solution to the posting of the string could not be found. It was discovered that the initial value for the upvote, which was always “0” , could be found and converted to 0 using the $set operator. This solved the problem on adding a recipe. It was then noticed that if the form was edited the readonly integer value was posted back to the database as a string (for example 6 would become “6”). After some time it was discoverd that the jinja inbuilt (|int) function could cast form values to integers. Problem solved.

The index_route function renders the home page. The home view contains the select elements for search by group and by criteria. When an option is clicked a simple jquery function triggers the respective url_for and displays the view.

Routes were then established to search for recipes by groups. These groups allow a user to view all recipes, vegetarian only recipes, vegan only recipes, and other recipes (not vegetarian or vegan). Other groups include starters, main course, etc. The mongo.db documentation (where available the python section) was explored to see what different operators and methods do. A test file (see additional-info folder), shows some functions carried out in the ide to explore the documents. The find().sort(), update_many(), update(_one), insert_one () methods are used extensively. The $inc and $set operators have already been used.

The $in operator was used within the vegan and vegetarian functions, for example; .find({"Suitable_for_Vegetarians": {$”in” : ["Yes", “yes”]}}. However the form now only accepts “Yes” so $in is no longer needed.

The all recipes route finds all recipes, sorts them by name ascending and displays certain attributes in the allRecipes view in an accordion style element, disabled where not required. For these views -U Flash paginate was installed to enable flask pagination (as per the flask documentation). The attributes rendered in this view include recipe name, category name, upvotes, country of origin, date added, allergens, total time and ingredients (collapsible). This view also calls the url_for('draw_chart') which takes a user to the dashboard view. At the bottom of the element a button links to the full recipe and another takes the user back to the home page.

The vegetarian route only returns specific fields using a projection; ({find({"Suitable_for_Vegetarians": "Yes"},{ "_id": 1, "Recipe_name": 1,etc }), which for very large databases would be quicker than returning all fields as was done with all the other groups.

Routes and views (child templates) were established for the other groups as for allRecipes above.

It was noticed that the pagination info bar affects the responsiveness of these pages on smaller screen sizes but the benefit of having simple pagination mitigates some loss of responsiveness.

Other routes and views were established allowing filtration by attribute. This application allows search for recipes based on top 5 upvotes, recipes with no allergens, quick recipes and recently added. Other operators such as $gte, $gt and $lt were used in these filters.

Again find().sort() and the .limit() methods were used.

The top_five route gets all upvotes, sorts them descending and then displays only 5. this is an okay solution for a small database but as it becomes larger the value used with the operator could be increased to omit finding values lower than it. Pagination is not needed as only 5 records are shown. The top_five html only renders recipe name, category name, upvote and country of origin fields. Again button links are wired to urls to get the full recipe or return to the home page.

Similar routes and views were established for the other filter criteria. These have pagination. The quick_recipe html page shows total time instead of country of origin (keeping the field rendered limit to 4), the noAllergens page shows allergens instead of country of origin and the recently_added html shows date added instead of country of origin.

The recently_added function uses the _id object to sort descending (uses id timestamp). At the moment it returns all in descending order but could use a limit when the database grows.

Prior to establishing recipe_book route the json module was imported. This allows the use of the json.dumps method which returns a string from an object. In addition json_util was inported from bson which is a tool to permit the use of python's json module with bson documents.

The recipe_book route was established, which effectively returns selected attributes (using a projection), from the database, in string data format, for processing by javascript into charts, specifically dc.js crossfilter charts.

A script file then takes the data (selected attributes), groups it together and processes it into interactive charts.

For example number boxes are generated showing the total number of recipes, the number of vegan recipes and the number of vegetarian recipes. Another pie chart shows the number of recipes by categories (courses). A bar chart shows the number of recipes by country. Stacked bar charts group recipes by type and allergens and type by category.

The draw_chart function renders the chart.html.

Finally the mongo-datatables module was installed so that the data could be rendered in summary format in a table.

The table_view route renders the table_view.html which returns a summary list of recipes along with some attributes (recipe name, category, country of origin, allergens and date added). Each recipe also has a button link to view the full recipe.
Simple search and pagination functionality is provided with the data table.

The additional info file (see additional-info folder), contains further information on code snippets and data sources.