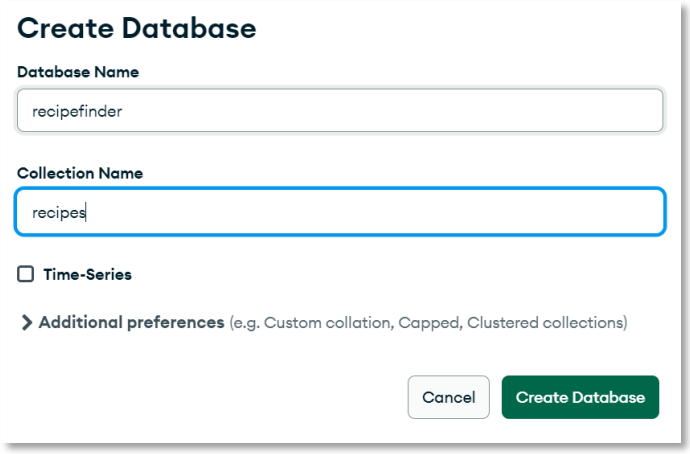
3DM – MongoDB Exercise 1

Imagine: You're hungry and open the fridge but at the sight of its content you have no idea what to prepare from it. Desperate, you sit down on a chair in the kitchen. Then you have an idea. There should be an application that makes suggestions for a suitable recipe based on your supplies! Without hesitation you start programming.

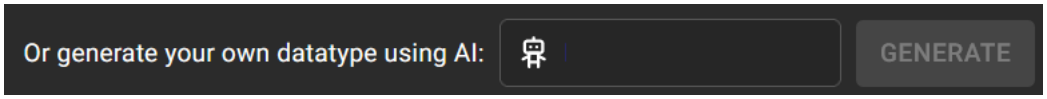
a) Create a new database with Compass named recipefinder containing a collection named recipes.



b) Use Mockaroo to generate 200 recipes with the following structure having 1-8 ingredients:

```
{
  "title": "Musli",
  "ingredients": [
    {
      "name": "Oat Flakes",
      "unit": "g",
      "quantity": 100
    },
    {
      "name": "Apple",
      "unit": "g",
      "quantity": 50
    },
    {
      "name": "Milk",
      "unit": "ml",
      "quantity": 0.25
    }
  ]
}
```

Tipp: use the AI option for generating the fields *name* and *title*.



c) Import the recipes with Compass into the recipes collection.

d) Follow the instructions in the tutorial «MongoDB Java Project Setup» to set up a new Java project. Use «**recipefinder**» as artifact Id in step e) instead of «mongodemo».

e) The sample code from the «MongoDB Java Project Setup» tutorial creates a mongoDB client and performs a ping operation to check the DB connection. Before we can perform any read or write operation, we must **open a specific database and a collection**. Replace the ping code with the code shown below:

```
// Create a new client and connect to the server
try (MongoClient mongoClient = MongoClients.create(settings)) {
  try {
    MongoDatabase recipeDB = mongoClient.getDatabase("recipefinder");
    MongoCollection<Document> recipeCol = recipeDB.getCollection("recipes");
    System.out.println("Found "+recipeCol.countDocuments()+" recipes");
  } catch (MongoException e) {
    e.printStackTrace();
  }
}
```

Explanation:

- getDatabase opens the indicated DB and returns an object that can be used to access the database.
- getCollection opens the indicated collections and returns an object that can be used to access the collection.
- countDocument return the total number of documents within the collection.

Run the app and verify that the document count is printed correctly to the console.

f) **Reading all documents** from a collection can be done with the code shown below. In addition, this code prints all recipe titles to the console.

```
MongoDatabase recipeDB = mongoClient.getDatabase("recipefinder");
MongoCollection<Document> recipeCol = recipeDB.getCollection("recipes");
System.out.println("Found " + recipeCol.countDocuments() + " recipes");

FindIterable<Document> allRecipes = recipeCol.find();

System.out.println("Available recipes: ");
for (Document d : allRecipes) {
  System.out.println("- "+d.get("title"));
}
```

Explanations:

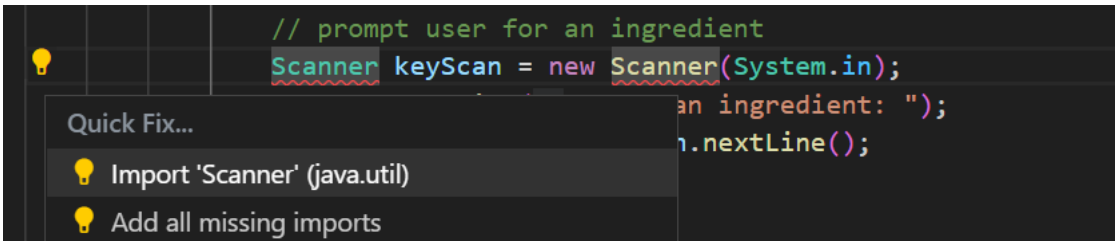
- find() without any parameters will return all documents in the collection.
- The type «Document» is like a HashMap: the get method allows to read fields within the document.
- The type «FindIterable» is a kind of list, but it does not allow you to access individual object, as you could do with an ArrayList using get(index). But you can always process it elementwise with a for-each loop.

Run the app and verify that all recipe titles are printed to the console.

g) Next, we let the user enter an available ingredient.

```
System.out.println("Found " + recipeCol.countDocuments() + " recipes");

// prompt user for an ingredient
Scanner keyScan = new Scanner(System.in);
System.out.print("Enter an ingredient: ");
String ingredient = keyScan.nextLine();

FindIterable<Document> allRecipes = recipeCol.find();

System.out.println("Available recipes: ");
for (Document d : allRecipes) {
  System.out.println("- "+d.get("title"));
}

keyScan.close();
```

Use Quick-Fix to include the Scanner library. Click the marked error to make it appear.



h) Now we can use the entered ingredient to update our find query. First, filters must be imported at the top of the class:

```
import static com.mongodb.client.model.Filters.*;
```

Then the query can be updated:

```
recipeCol.find(eq("ingredients.name", ingredient));
```

The update query only returns recipes containing the entered ingredient. Verify that it works correctly with different ingredients. Example:

```
Found 4 recipes
Enter an ingredient: Potatoes
Available recipes:
- Potato salad
- Mashed potatoes
```

i) Now we want the user to select a recipe. If a valid recipe title is entered, the application should list all required ingredients.

```
Found 4 recipes
Enter an ingredient: Milk
Available recipes:
- Birchermuesli
- Mashed potatoes
Enter the title of your preferred recipe: Birchermuesli
Good choice! For Birchermuesli you need:
- 100g of Oat Flakes
- 50g of Apples
- 0.25dl of Milk
```

If there no valid title was entered, an error message should be shown

```
Found 4 recipes
Enter an ingredient: Milk
Available recipes:
- Birchermuesli
- Mashed potatoes
Enter the title of your preferred recipe: Pizza
Recipe not found. Try again.
```

Tipps: To get only one recipe instead of a list, first() can be added to the find query.

```
Document selectedRecipe = recipeCol.find(..........).first();
```

the list of ingredients can be read as follows:

```
List<Document> ingr = selectedRecipe.get("ingredients", List.class);
```

When reading a list, VS Code show a warning. You can suppress this warning by adding the annotation shown below before the App class begins.

```
@SuppressWarnings("unchecked")
public class App {
    public static void main(String[] args) {
```

j) **Submit** the main class (App.java) of your project on Moodle. Feel free to add additional features to your application 😊.