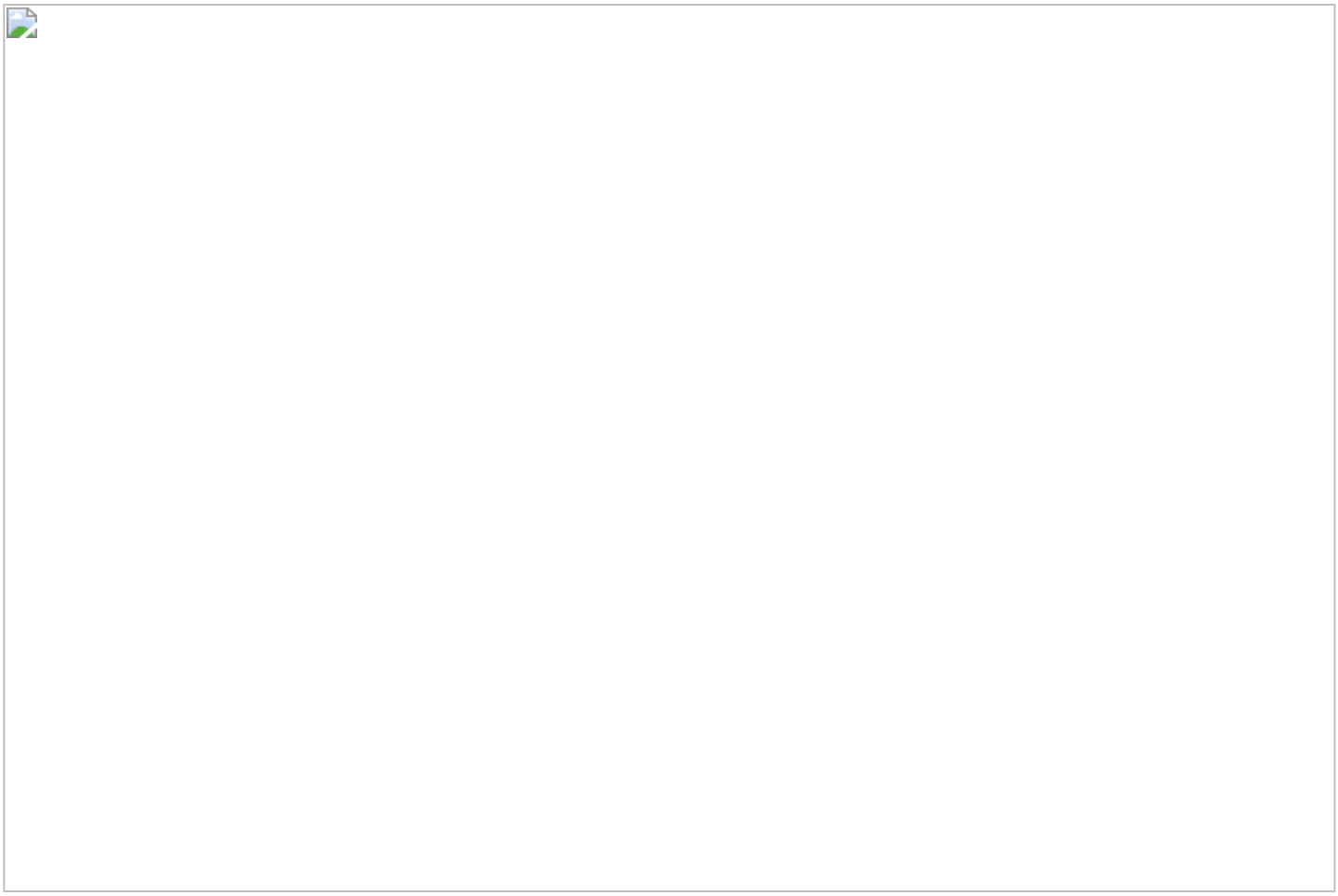
CRUD Operations on MongoDB Tree Data Structure

Nested categories using Node/Express.js, MongoDB and Mongoose

Lamine Bakelli
Apr 24 · 7 min read ★



All our wisdom is stored in the trees — Santosh Kalwar

In a real-world softwares/apps, we often need to store and manage the categories of items/subjects commonly organized in hierarchical structures. In this post I will briefly explain how to use Node/Express, MongoDB, and Mongoose to display, create, read, update and delete categories in a tree data structure. The following points will be addressed:

- Introduction
- Defining schema
- Indexing
- The slug fonction
- Creating the root category
- Creating the sub-category
- Display a category
- Display the descendants of a category
- Change the Ancestry of a Category
- Rename a Category
- Remove a Category

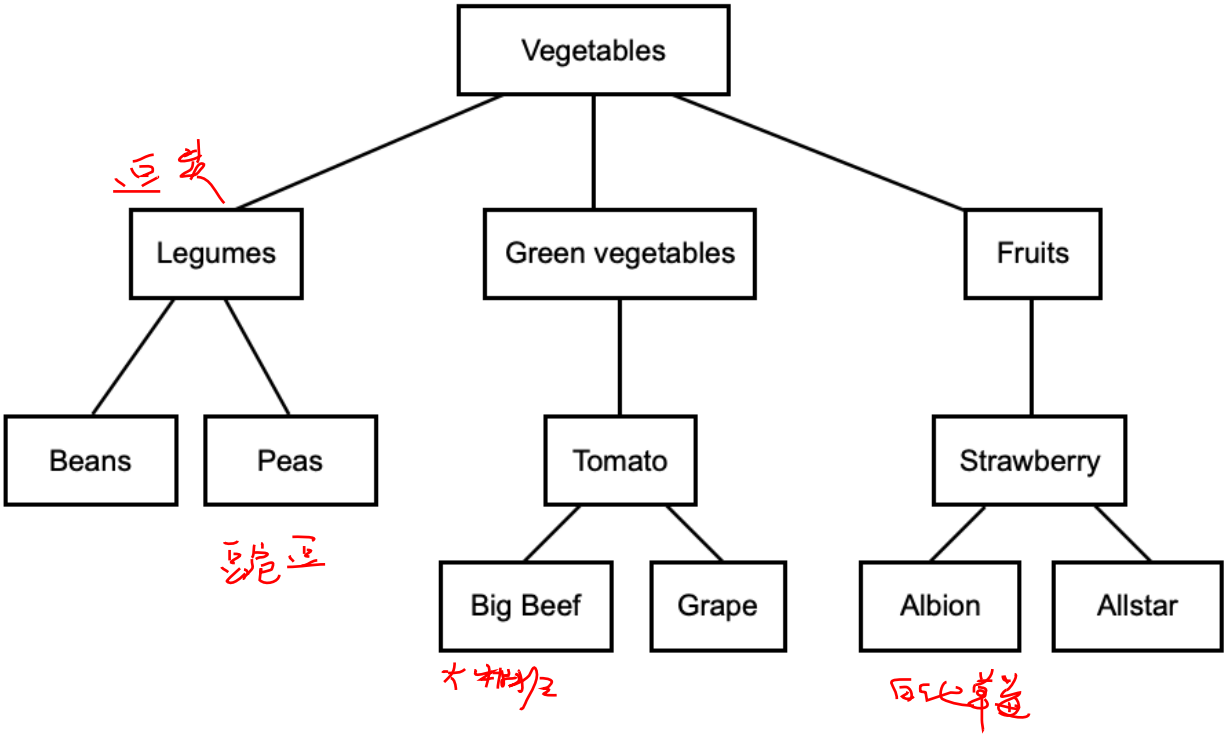
I won't go over Node/Express, MongoDB, and Mongoose installation because it is beyond the scope of this post.

. . .

Introduction

There are several ways to model the nested data or hierarchical structures with MongoDB as it's well explained in the MongoDB official documentation. In this post we will use the *Model "Tree Structures with an Array of Ancestors and Parent References"* to model the nested data relationships, by storing the reference of the parent's category *and* an array of all its ancestors.

The figure below represents an example of nested categories that will be used throughout this post.



Example of vegetables Category Tree

Defining schema

The code below shows how to define the nested categories schema

```
import mongoose from 'mongoose';

const CategorySchema = new mongoose.Schema({

name: String,
slug: String,
parent: {
  type: mongoose.Schema.Types.ObjectId,
  default: null,
  ref: 'Category'
},
ancestors: [{
     _id: {
        type: mongoose.Schema.Types.ObjectId,
        ref: "Category"
     },
     name: String,
```

```
        slug: String
}]
});
```

First we import mongoose, then we use the Schema constructor to create a new schema instance with the following fields inside the constructor's object parameter: The name of the category, the slug, a reference to the parent category and an array of all its ancestors.

Indexing

a. Create a unique index on the slug field by replacing the following line in the schema

```
slug: String
```

by the this line

```
slug: { type: String, index: true }
```

b. Create a unique index on the ancestors _id fields by adding the following line to ancestors array elements

```
index: true
```

The final schema syntax looks like this:

```
const CategorySchema = new mongoose.Schema({

name: String,
slug: { type: String, index: true },
parent: {
  type: mongoose.Schema.Types.ObjectId,
  default: null,
  ref: 'Category'
},
ancestors: [{
     _id: {
        type: mongoose.Schema.Types.ObjectId,
```

```
        ref: "Category",
        index: true
},
    name: String,
    slug: String
}]
});
```

The slug fonction

For the slug fonction I've used Matthias Hagemann's snippet

```
function slugify(string) {
  const a = 'àáâäæãåāăąçćčđďèéêëēėęěğǵḧîïíīįìłḿñńňôöòóœøōõőṕŕřßśšşșťțûüùúūǘůűǔ
  const b = 'aaaaaaaaaacccddeeeeeeeegghiiiiiilmnnnnooooooooprrsssssttuuuuuu
  const p = new RegExp(a.split('').join('|'), 'g')

  return string.toString().toLowerCase()
    .replace(/\s+/g, '-') // Replace spaces with -
    .replace(p, c => b.charAt(a.indexOf(c))) // Replace special characters
    .replace(/&/g, '-and-') // Replace & with 'and'
    .replace(/[^\w\-]+/g, '') // Remove all non-word characters
    .replace(/\-\-+/g, '-') // Replace multiple - with single -
    .replace(/^-+/, '') // Trim - from start of text
    .replace(/-+$/, '') // Trim - from end of text
}
```

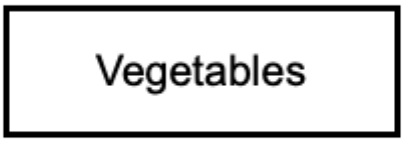
slugify.js hosted with ♡ by GitHub view raw

We define the following "pre" schema middleware to create a slug using the previous *slugify* function with category name as parameter

```
CategorySchema.pre('save', async function (next) {
   this.slug = slugify(this.name);
   next();
});
```

Creating the root Category

We create a root category simply by creating a category that has no parent and with an empty ancestors array.



The root category

To achieve this we're going to import the Express application object, use it to get a Router object and then we define the route using the router.post() method

```
var express = require('express');
var router = express.Router();

router.post('/', async (req, res) => {
  const category = new Category({name: req.body.name})

try {
  let newCategory = await category.save();
  res.status(201).send({ response: `Category ${newCategory._id}` });
} catch (err) {
  res.status(500).send(err);
}
});
```

To test our operations we use Postman , you can use any other REST API client tool.

We post the category name without using any parent ID as shown in the picture below

New Category Query using Postman screenshot

The following document is inserted. As you may noticed the parent field is equal to null and ancestors array is empty.

The parent document

In the same way we create all other parents categories

Creating the sub-categories

By definition a sub-category has a parent and ancestors, so if we want to create one, we start by creating a category referring to its parent, then we build its ancestors array with the *ancestors array builder* function.

Ancestors array builder function

We declare the following helper function that build the ancestors array of a given sub-category

```
const buildAncestors = async (id, parent_id) => {
   let ancest = [];
   try {
       let parent_category = await Category.findOne({ "_id":
parent_id },{ "name": 1, "slug": 1, "ancestors": 1 }).exec();
```

```
if( parent_category ) {
          const { _id, name, slug } = parent_category;
          const ancest = [...parent_category.ancestors];
          ancest.unshift({ _id, name, slug })
          const category = await Category.findByIdAndUpdate(id, {
$set: { "ancestors": ancest } });
        }
     } catch (err) {
         console.log(err.message)
      }
}
```

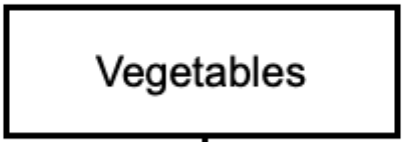
The parameters of the function are id and parent_id, *id* is the id of the the sub-category and *parent_id* is the id of its parent. As you can see the buildAncestors function simply find the ancestors array of the parent category and append the parent category to it.

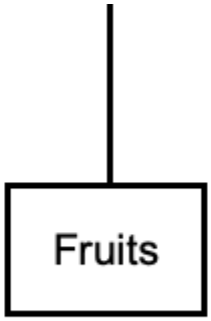
We update our previous router.post() method to support both category and sub-category creation.

```
router.post('/', async (req, res) => {

  let parent = req.body.parent ? req.body.parent : null;
  const category = new Category({name: req.body.name, parent})

try {
  let newCategory = await category.save();
  buildAncestors(newCategory._id, parent)
  res.status(201).send({ response: `Category ${newCategory._id}` });
} catch (err) {
  res.status(500).send(err);
}
});
```

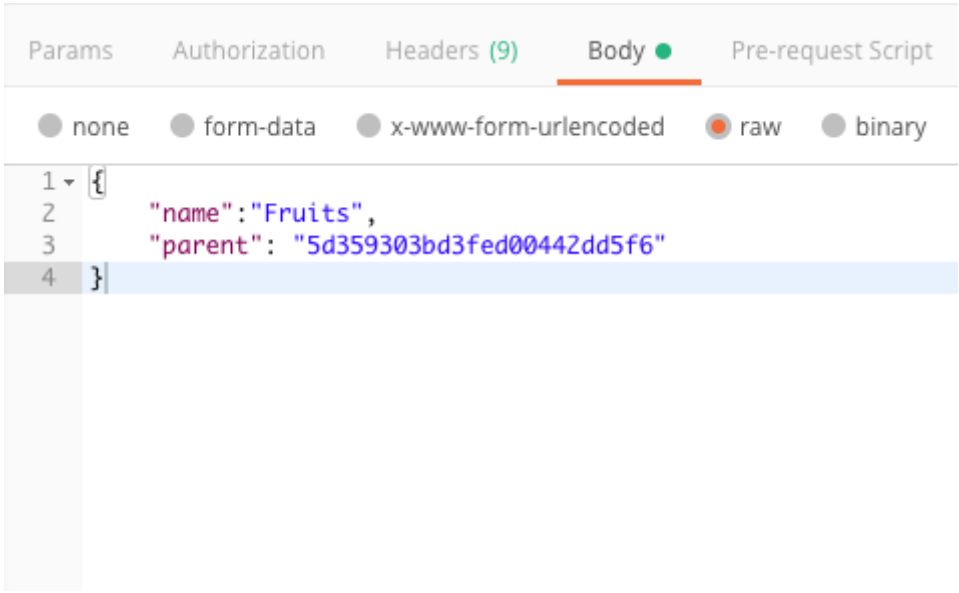
To sum up, we create any category by using its name and its parent category ID if it exists, then we call the build the ancestors array helper function to build its ancestors array.

For example we create *Fruit* category as a child of *Vegetables* category



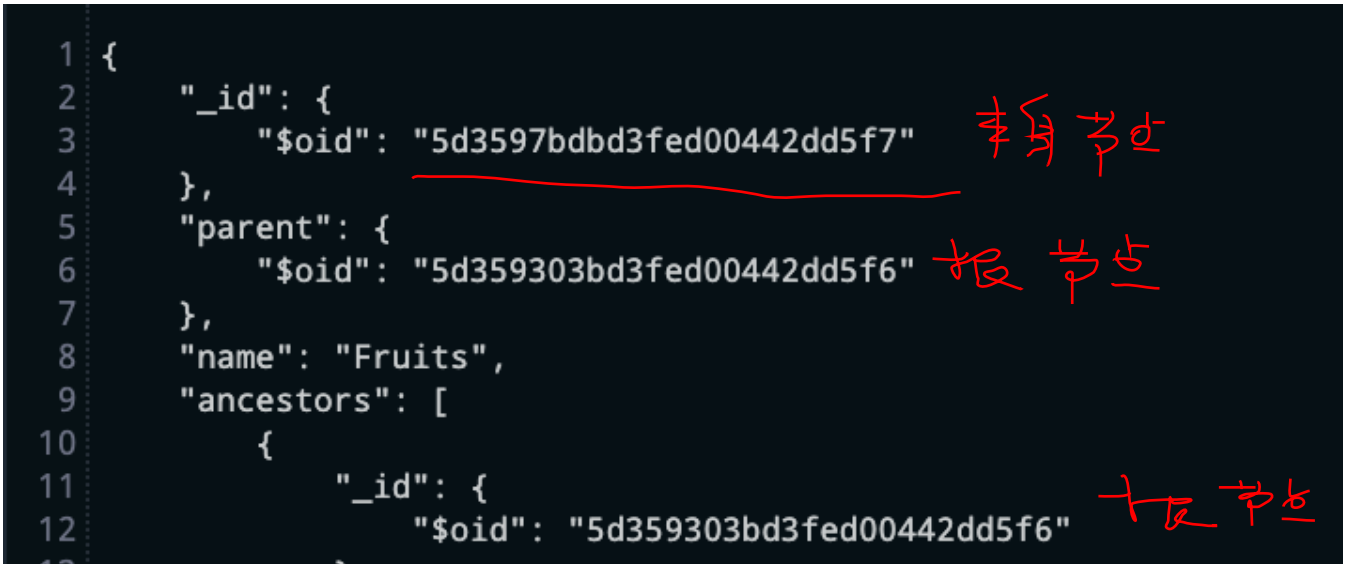


Send a POST request with the category name “Fruits” and its parent ID in the body data as follows



New Child Category Query using Postman screenshot

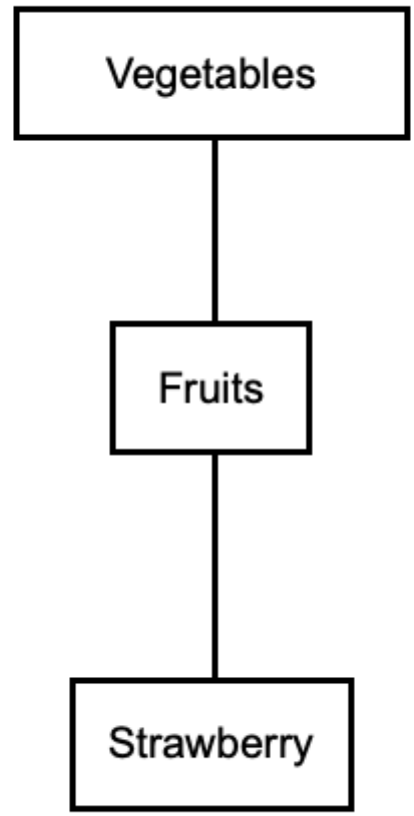
We will get the following document



```
            },
            "name": "Vegetables",
            "slug": "vegetables"
        }
    ],
    "slug": "fruits",
    "__v": 0
}
```

The child document

In the same way we can create Strawberry under Fruits. The result will be:




```
{
    "_id": {
        "$oid": "5d38955c02ad5a004458a989"
    },
    "parent": {
        "$oid": "5d3597bdbd3fed00442dd5f7"
    },
    "name": "Strawberry",
    "ancestors": [
        {
```

In the same way we can create all the categories hierarchy

Display a Category

This routing method uses the slug parameter to return the category name and the ancestors array, which is the “breadcrumb trail” from the current category to the top level category

```
router.get('/', async (req, res) => {

try {
     const result = await Category.find({ slug: req.query.slug })
     .select({
     "_id": false,
     "name": true,
     "ancestors.slug": true,
     "ancestors.name": true }).exec();

     res.status(201).send({ "status": "success", "result": result
});
} catch (err) {
    res.status(500).send(err);
}
});
```

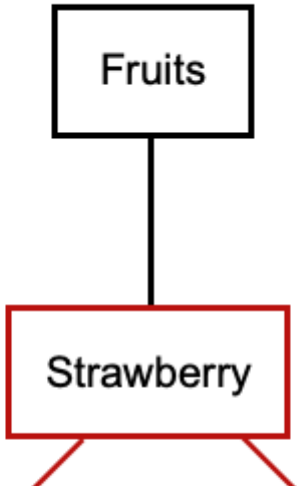
Request Grape category will give us the following result

```
{
    "status": "success",
    "result": [
        {
            "name": "Grape",
            "ancestors": [
                {
                    "name": "Tomato",
                    "slug": "tomato"
                },
                {
                    "name": "Green vegetables",
                    "slug": "green-vegetables"
                },
                {
                    "name": "Vegetables",
                    "slug": "vegetables"
                }
            ]
        }
    ]
}
```


Display the descendants of a Category

The following routing method displays the descendant of a given category ID

```
router.get('/descendants', async (req, res) => {
try {
    const result = await Category.find({ "ancestors._id":
req.query.category_id })
     .select({ "_id": false, "name": true })
     .exec();
   res.status(201).send({ "status": "success", "result": result });
   } catch (err) {
     res.status(500).send(err);
   }
})
```

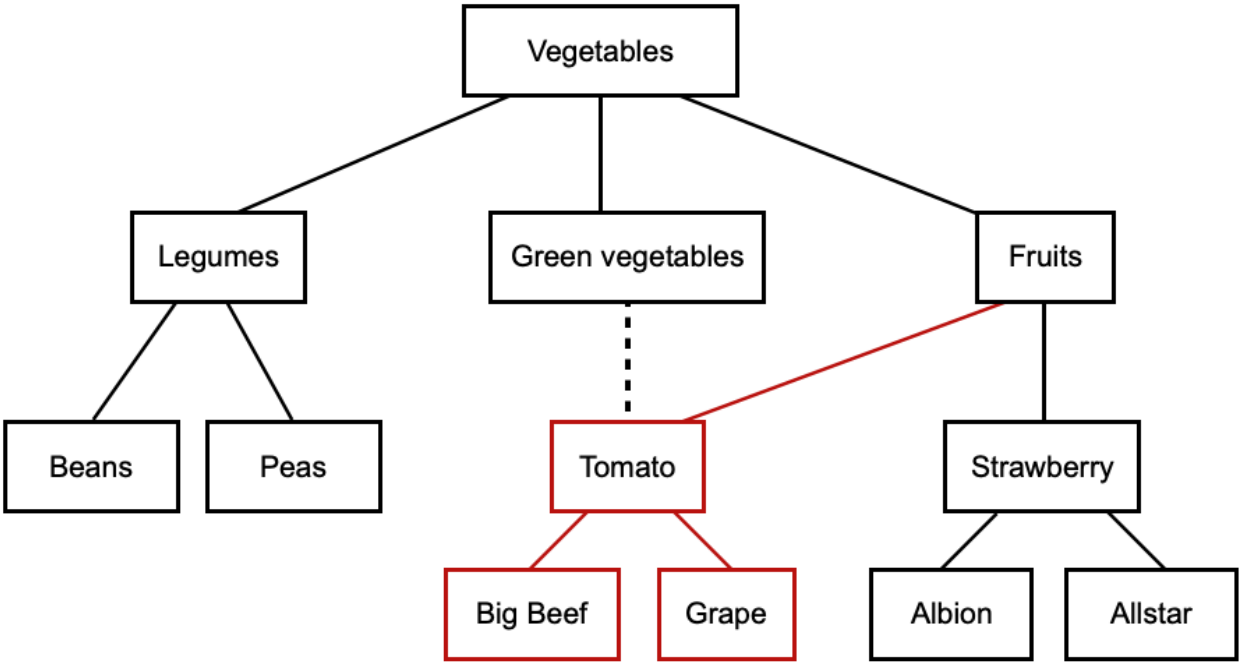


The descendants of Fruits category

We get the following result by sending a GET request with the "*Fruit*" Category ID as parameter to descendants endpoints

```
{
    "status": "success",
    "result": [
        {
            "name": "Strawberry"
        },
        {
            "name": "Albion"
        },
        {
            "name": "Allstar"
        }
    ]
}
```

Change the Ancestry of a Category



1. Update the parent of the category to the new parent as follows

```
const category = await Category.findByIdAndUpdate(category_id, {
$set: { "parent": new_parent_id } });
```

2. Rebuild and update the ancestors array of the category and all its descendants by calling the following helper function

```
buildHierarchyAncestors(category._id, new_parent_id);
```

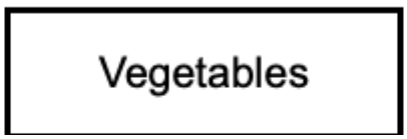
buildHierarchyAncestors is a recursive function that traverse and build the ancestors array of category descendants

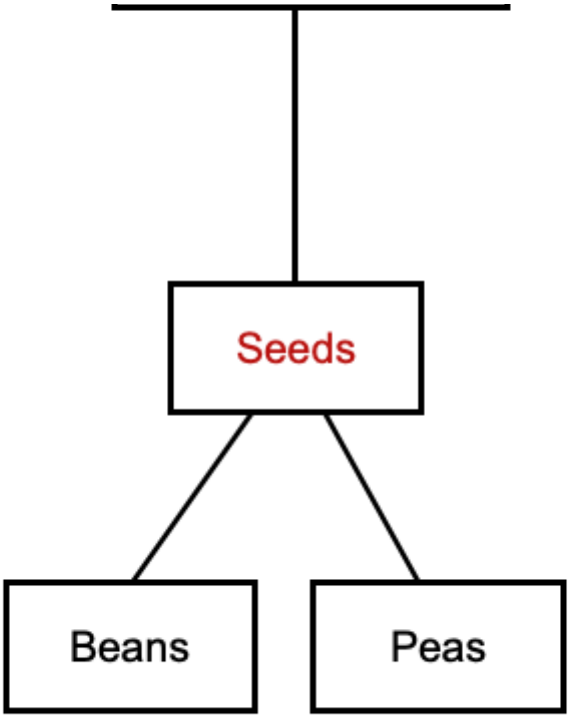
```
export const buildHierarchyAncestors = async ( category_id,
parent_id ) => {

if( category_id && parent_id )
   buildAncestors(category_id, parent_id)
   const result = await Category.find({ 'parent': category_id
}).exec();

if(result)
   result.forEach((doc) => {
      buildHierarchyAncestors(doc._id, category_id)})
   }
}
```

Rename a Category

To rename a category we should update the name field in the category document and also update the category name in the ancestors field of all its descendants documents.

For example to rename the category Legumes to Seeds as in the following figure:





1. First we update the name and the slug of the category with following query:

```
Category.findByIdAndUpdate(category_id, { $set: { "name":
category_name, "slug": slugify(category_name) } });
```

The slugify function is the same function used above in the pre save hook

2. Then we update the category name and the slug in the ancestors array of all the descendants documents using the following query:

```
Category.update({"ancestors._id": category_id},
{"$set": {"ancestors.$.name": category_name, "ancestors.$.slug":
slugify(category_name) }}, {multi: true});
```

Remove a Category

Delete the category and all its descendants

Finally to delete a category we need to delete the category and all its descendants using the following queries

```
err = await Category.findByIdAndRemove(category_id);
if(!err)
  result = await Category.deleteMany({"ancestors._id":
category_id});
```

That's it! I hope you've found this post informative and helpful. I would love to hear your feedback!

Thank you for reading!

Mongodb Mongoose Tree Data Structures Expressjs Nodejs